Philosophy of Computer Science

William J. Rapaport

Department of Computer Science and Engineering,
Department of Philosophy, Department of Linguistics,
and Center for Cognitive Science
University at Buffalo, The State University of New York,
Buffalo, NY 14260-2500
rapaport@buffalo.edu
http://www.cse.buffalo.edu/~rapaport/

DRAFT © 2004–2019 by William J. Rapaport

September 6, 2019

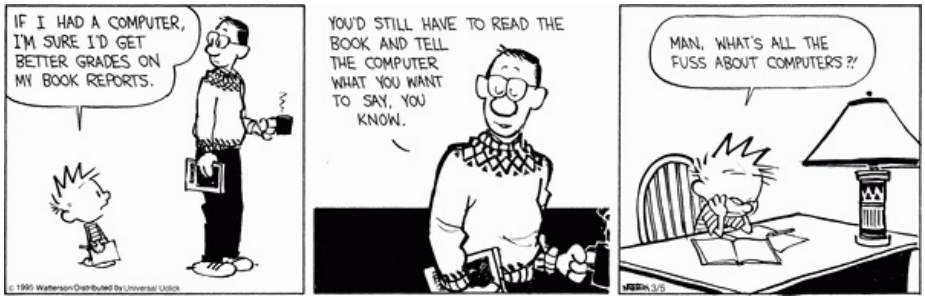


Figure 1: http://www.gocomics.com/calvinandhobbes/2015/3/5, ©1995 Watterson

Contents

I Philosophy and Computer Science **27**

1 What Is Philosophy of Computer Science? **29**
1.1 Recommended Readings: . . . 30
1.2 What This Book Is About . . . 31
1.3 What This Book Is *Not* About . . . 32
1.4 . . . Further Sources of Information . . . 35

2 What Is Philosophy? **39**
2.1 Readings: . . . 40
2.2 Introduction . . . 41
2.3 A Definition of 'Philosophy' . . . 41
2.4 What Is Truth? . . . 43
2.4.1 The Correspondence Theory of Truth . . . 44
2.4.2 The Coherence Theory of Truth . . . 44
2.4.3 Correspondence vs. Coherence . . . 45
2.5 On Searching for the Truth vs. Finding It . . . 46
2.5.1 Asking "Why?" . . . 47
2.5.2 Can There Be Progress in Philosophy? . . . 48
2.5.3 Skepticism . . . 50
2.6 What Is "Rational"? . . . 51
2.6.1 Kinds of Rationality . . . 51
2.6.1.1 Deductive Logical Rationality . . . 52
2.6.1.2 Inductive Logical Rationality . . . 53
2.6.1.3 Abductive Logical Rationality . . . 53
2.6.1.4 Non-Monotonic Logical Rationality . . . 54
2.6.1.5 Computational Rationality . . . 54
2.6.2 Science and Philosophy . . . 55
2.6.2.1 Is Science Philosophy? . . . 55
2.6.2.2 Is Philosophy a Science? . . . 56
2.6.3 Is It Always Rational to Be Rational? . . . 57
2.7 What Is the Import of "Personal Search"? . . . 57
2.8 What Is the Import of "In Any Field"? . . . 58
2.9 Philosophy and Computer Science . . . 64
2.10 Further Sources of Information . . . 65

II Computer Science, Computation, and Computers 69

3 What Is Computer Science? 71
3.1 Readings . . . 72
3.2 Introduction . . . 75
3.3 Preliminary Questions . . . 75
3.3.1 Naming the Discipline . . . 75
3.3.2 Why Ask What CS Is? . . . 76
3.3.2.1 Academic Motivations . . . 76
3.3.2.1.1 Academic Politics. . . . 76
3.3.2.1.2 Academic Pedagogy. . . . 78
3.3.2.1.3 Academic Publicity. . . . 78
3.3.2.2 Intellectual or Philosophical Motivations . . . 79
3.3.3 What Does It Mean to Ask What Something Is? . . . 80
3.3.3.1 Determining Boundaries . . . 80
3.3.3.2 Three Other Controversial Terms . . . 82
3.3.3.2.1 What Is a Planet? . . . 82
3.3.3.2.2 What Is Computation? . . . 84
3.3.3.2.3 What Is Thinking? . . . 84
3.4 Two Kinds of Definition . . . 84
3.4.1 An Extensional Definition of CS . . . 85
3.4.2 Intensional Definitions of CS . . . 87
3.5 CS Is the *Science* of *Computers* . . . 88
3.5.1 Objection to the First Premise . . . 89
3.5.2 Objection: Computers Are Tools, not Phenomena . . . 90
3.5.3 Digression: The Once-upon-a-Time Science of Microscopy . . . 92
3.5.4 Objection: Computer Science Is Just a Branch of 95
3.5.5 Objection: What about Algorithms? . . . 96
3.6 CS Studies *Algorithms* . . . 96
3.6.1 Only Algorithms? . . . 97
3.6.2 Or Computers, Too? . . . 100
3.7 Physical Computers vs. Abstract Algorithms . . . 101
3.8 CS Studies *Information* . . . 103
3.9 CS *Is* a Science . . . 106
3.9.1 Computer Science Is a *Formal* (Mathematical) Science . . . 106
3.9.2 CS Is the Science of *Intellectual* Processes . . . 110
3.9.3 CS *Is* a Natural Science (of *Procedures*) . . . 111
3.9.4 CS Is a Natural Science of *the Artificial* . . . 113
3.9.5 Computer Science Is an *Empirical Study* . . . 114
3.10 CS Is *Engineering* . . . 117
3.11 Science *xor* Engineering? . . . 120
3.12 CS as "Both" . . . 120
3.13 CS as "More" . . . 123
3.13.1 CS Is a *New Kind of Engineering* . . . 123
3.13.1.1 CS Is a *Kind* of Engineering . . . 123
3.13.1.2 CS Is a *New* Kind of Engineering . . . 125

3.13.2 CS Is a New Kind of *Science* 126
3.14 CS as "Neither" 129
3.14.1 CS Has Its Own Paradigm 129
3.14.2 CS Is *Art* 130
3.14.3 CS Is the Study of *Complexity* 132
3.14.4 CS Is the *Philosophy*(!) of Procedures 134
3.14.5 CS Is *Computational Thinking* 135
3.14.6 CS Is *AI* 138
3.14.7 Is CS *Magic*? 139
3.15 So, What Is Computer Science? 144
3.15.1 Computer Science and Elephants 144
3.15.2 Five Central Questions of CS 146
3.15.2.1 Computability 146
3.15.2.1.1 *What* Can Be Computed? 146
3.15.2.1.2 *How* Is It Computable? 148
3.15.2.2 Efficient Computability 150
3.15.2.3 Practical Computability 151
3.15.2.4 Physical Computability 152
3.15.2.5 Ethical Computability 152
3.15.3 Wing's Five Questions 153
3.15.4 Conclusion 155
3.16 A Look Ahead 155
3.17 Questions for the Reader 156
3.18 Further Sources of Information 160

4 What Is Science? 163
4.1 Readings: 164
4.2 Introduction 166
4.3 Science and Non-Science 166
4.4 Early Modern Science 168
4.5 The Goals of Science 169
4.5.1 Description as the Goal of Science 169
4.5.2 Explanation as the Goal of Science 170
4.5.3 Prediction as the Goal of Science 171
4.6 Instrumentalism vs. Realism 172
4.7 What Is a Scientific Theory? 175
4.8 "The" Scientific Method 176
4.9 Alternatives to "The Scientific Method" 177
4.9.1 Falsifiability 177
4.9.1.1 Science as Conjectures and Refutations 177
4.9.1.2 The Logic of Falsifiability 179
4.9.1.3 Problems with Falsifiability 180
4.9.2 Scientific Revolutions 181
4.9.3 Other Alternatives 182
4.10 Is CS a Science? 183
4.11 What Kind of Science Might CS Be? 184

4.12 Questions to Think About . . . 185
4.13 Further Sources of Information . . . 187

5 What Is Engineering? 189
5.1 Readings: . . . 190
5.2 Can We Define 'Engineering'? . . . 191
5.3 Could Engineering Be Science? . . . 193
5.4 A Brief History of Engineering . . . 194
5.5 Conceptions of Engineering . . . 195
5.6 What Do Engineers Do? . . . 196
5.6.1 Engineering as Design . . . 196
5.6.2 Engineering as Building . . . 197
5.7 The Engineering Method . . . 198
5.8 Software Engineering . . . 200
5.9 Closing Remarks . . . 202
5.10 Questions to Think About . . . 203
5.11 Further Sources of Information . . . 205

6 What Is a Computer? A Historical Perspective 207
6.1 Readings: . . . 208
6.2 Introduction . . . 210
6.3 Would You Like to Be a Computer? Some Terminology . . . 211
6.4 Two Histories of Computers . . . 212
6.5 The Engineering History . . . 213
6.5.1 Ancient Greece . . . 213
6.5.2 17th-Century Adding Machines . . . 214
6.5.3 Babbage's Machines . . . 214
6.5.4 Electronic Computers . . . 217
6.5.5 Modern Computers . . . 219
6.6 The Scientific, Mathematical, Logical History . . . 222
6.7 The Histories Converge . . . 227
6.8 What Is a Computer? . . . 228
6.8.1 What Is a Computer, Given the Engineering History? . . . 228
6.8.2 What Is a Computer, Given the Scientific History? . . . 230
6.9 Further Sources of Information . . . 230

7 What Is an Algorithm? 231
7.1 Introduction . . . 233
7.2 What Is 'Computation'? . . . 233
7.2.1 'compute' . . . 233
7.2.2 'reckon' . . . 234
7.2.3 'count', 'calculate', 'figure' . . . 234
7.2.4 'computation' . . . 235
7.3 What Is Computation? . . . 236

7.3.1 What Is a Function? . . . 236
7.3.1.1 Two Meanings . . . 236
7.3.1.2 Functions Described Extensionally . . . 236
7.3.1.3 Interlude: Functions Described as Machines . . . 239
7.3.1.4 Functions Described Intensionally . . . 240
7.3.1.5 Computable Functions . . . 243
7.4 'Algorithm' Made Precise . . . 245
7.4.1 Ancient Algorithms . . . 245
7.4.2 "Effectiveness" . . . 247
7.4.3 Three Attempts at Precision . . . 248
7.4.3.1 Markov . . . 248
7.4.3.2 Kleene . . . 249
7.4.3.3 Knuth . . . 250
7.4.3.4 Summary . . . 255
7.5 Five Great Insights of CS . . . 256
7.5.1 Bacon's, Leibniz's, Morse's, Boole's, Ramsey's, Turing's, and Shannon's *Representational Insight* . . . 256
7.5.2 Turing's *Processing Insight* . . . 258
7.5.3 Böhm & Jacopini's *Structural Insight* . . . 259
7.5.4 The Church-Turing Computability Thesis . . . 266
7.5.5 Turing's, Kay's, Denning's, and Piccinini's *Implementation Insight* . . . 271
7.6 Structured Programming and Recursive Functions . . . 273
7.6.1 Structured Programming . . . 273
7.6.1.1 Structured Programs . . . 273
7.6.1.2 Two Kinds of Structured Programs . . . 275
7.6.2 Recursive Functions . . . 277
7.6.2.1 A Recursive Definition of Natural Numbers . . . 277
7.6.2.2 Recursive Definitions of Recursive Functions . . . 278
7.6.2.3 Classification of Recursive Functions . . . 283
7.7 The Halting Problem . . . 285
7.7.1 Introduction . . . 285
7.7.2 Proof Sketch that H Is Not Computable . . . 288
7.7.2.1 Step 1 . . . 288
7.7.2.2 Step 2 . . . 289
7.7.2.3 Step 3 . . . 290
7.7.2.4 Step 4 . . . 290
7.7.2.5 Final Result . . . 291
7.7.3 Other Non-Computable Functions . . . 291
7.7.3.1 Hilbert's 10th Problem . . . 292
7.7.3.2 The Busy Beaver Problem . . . 292
7.8 Summary . . . 293
7.9 Questions for the Reader . . . 294
7.10 Further Sources of Information . . . 296

8 Turing's Analysis of Computation **297**
8.1 Required Reading: . . . 298
8.2 Introduction . . . 299
8.3 Slow and Active Reading . . . 300
8.4 Title: "The *Entscheidungsproblem*" . . . 300
8.5 Paragraph 1 . . . 301
8.5.1 Paragraph 1, Sentence 1 . . . 301
8.5.1.1 "Computable" . . . 301
8.5.1.2 Real Numbers . . . 301
8.5.1.3 Finitely Calculable . . . 302
8.5.2 Paragraph 1, Last Sentence . . . 302
8.6 Paragraph 2 . . . 303
8.6.1 Paragraph 2, Sentence 1 . . . 303
8.6.2 Paragraph 2, Last Sentence . . . 303
8.7 Section 1, Paragraph 1: "Computing Machines" . . . 304
8.8 Section 9:
"The Extent of the Computable Numbers" . . . 305
8.8.1 Section 9, Paragraphs 1 and 2 . . . 305
8.8.2 Section 9, Subsection I . . . 306
8.8.2.1 Section 9, Subsection I, Paragraph 1 . . . 306
8.8.2.1.1 Is It True? . . . 306
8.8.2.1.2 What About the Paper? . . . 306
8.8.2.1.3 What About the Symbols? . . . 308
8.8.2.2 Section 9, Subsection I, Paragraph 2: States of Mind 310
8.8.2.3 Section 9, Subsection I, Paragraph 3: Operations . . 311
8.8.2.4 Section 9, Subsection I, Paragraph 4: Operations . . 313
8.8.2.5 Section 9, Subsection I, Paragraph 5: More Operations 313
8.8.2.6 Section 9, Subsection I, Paragraph 6: Summary of Operations . . . 314
8.8.2.7 Section 9, Subsection I, Paragraph 7 . . . 314
8.8.2.7.1 Conditions. . . . 314
8.8.2.7.2 States of Mind Clarified. . . . 315
8.8.2.8 Section 9, Subsection I, Paragraph 8 . . . 316
8.8.2.8.1 The Turing Machine. . . . 316
8.8.2.8.2 Turing's (Computability) Thesis. . . . 317
8.8.2.8.3 Turing Machines as AI Programs. . . . 318
8.9 Section 1, continued . . . 319
8.9.1 Section 1, Paragraph 2 . . . 319
8.9.2 Section 1, Paragraph 3 . . . 324
8.10 Section 2: "Definitions" . . . 325
8.10.1 "Automatic Machines" . . . 325
8.10.2 "Computing Machines" . . . 326
8.10.2.1 Paragraph 1 . . . 326
8.10.2.2 Paragraph 2 . . . 327
8.10.3 "Circular and Circle-Free Machines" . . . 327
8.10.3.1 Paragraph 1 . . . 327

8.10.3.2 Paragraph 2 328
8.10.3.3 Coda: A Possible Explanation of 'Circular' 330
8.10.4 "Computable Sequences and Numbers" 331
8.11 Section 3: "Examples of Computing Machines" 331
8.11.1 Section 3, Example I 331
8.11.1.1 Section 3, Example I, Paragraph 1 331
8.11.1.2 Section 3, Example I, Paragraph 2 335
8.11.2 Section 3, Example II 337
8.11.2.1 Section 3, Example II, Paragraph 1 337
8.11.2.2 Section 3, Example II, Paragraph 2 342
8.12 Section 4: "Abbreviated Tables" 345
8.13 Section 5: "Enumeration of Computable Sequences" 346
8.14 Section 6: "The Universal Computing Machine" 352
8.15 The Rest of Turing's Paper 355
8.16 Further Sources of Information 356

9 What Is a Computer? A Philosophical Perspective 359
9.1 Readings: 360
9.2 Introduction 362
9.3 Informal Definitions 364
9.3.1 Reference-Book Definitions 364
9.3.2 Von Neumann's Definition 364
9.3.3 Samuel's Definition 365
9.3.4 Davis's Characterization 366
9.3.5 Discussion 366
9.4 Computers, Turing Machines, and Universal Turing Machines 369
9.4.1 Computers as Turing Machines 369
9.4.2 Stored Program vs. Programmable 372
9.5 John Searle: Anything Is a Computer 374
9.5.1 Searle's Argument 374
9.5.2 Computers Are Described in Terms of 0s and 1s 375
9.5.3 Being a Computer Is a Syntactic Property 376
9.5.4 Being a Computer Is Not an Intrinsic Property of Physical Objects 378
9.5.5 We Can Ascribe the Property of Being a Computer to Any Object 382
9.5.6 Everything Is a Computer 382
9.5.7 Other Views in the Vicinity of Searle's 384
9.6 Patrick Hayes: Computers as Magic Paper 386
9.7 Gualtiero Piccinini: Computers as Digital String Manipulators 390
9.7.1 Definition P1 391
9.7.2 Definition P2 392
9.8 What Else Might Be a Computer? 393

9.8.1 Is a Brain a Computer? . . . 394
9.8.2 Is the Universe a Computer? . . . 398
9.8.2.1 Wolfram's Argument . . . 400
9.8.2.2 Lloyd's Argument . . . 401
9.9 Conclusion . . . 403
9.10 Questions for the Reader . . . 405

III The Church-Turing Computability Thesis 409

10 What Is a Procedure? 413
10.1 Readings: . . . 414
10.2 Introduction . . . 416
10.2.1 The Church-Turing Computability Thesis . . . 417
10.3 What Is a Procedure? . . . 422
10.4 Two Challenges to the Computability Thesis . . . 424
10.4.1 Carol Cleland: Some Effective Procedures Are Not Turing Machines . . . 424
10.4.2 Beth Preston: Recipes, Algorithms, and Specifications . . . 430
10.5 Discussion . . . 434

11 What Is Hypercomputation? 437
11.1 Readings: . . . 438
11.2 Introduction . . . 440
11.3 Hypercomputation . . . 441
11.3.1 Copeland's Theory . . . 441
11.3.2 Questions to Think About . . . 444
11.3.3 Objections to Hypercomputation . . . 444
11.4 Kinds of Hypercomputation and Hypercomputers . . . 445
11.4.1 "Newer Physics" Hypercomputers . . . 445
11.4.2 Analog Recurrent Neural Networks . . . 449
11.4.3 Interactive Computation . . . 449
11.4.3.1 Can a Program Have Zero Inputs? . . . 449
11.4.3.2 Batch vs. Online Processing . . . 449
11.4.3.3 Peter Wegner: Interaction Is Not Turing-Computable 452
11.4.3.4 Can Interaction Be Simulated by a Non-Interactive Turing Machine? . . . 455
11.4.3.4.1 The Power of Interaction. . . . 455
11.4.3.4.2 Simulating a *Halting* Interaction Machine. 455
11.4.3.4.3 Simulating a *Non-Halting* Interaction Machine. . . . 458
11.4.3.4.4 Concurrent Computation. . . . 460
11.4.4 Oracle Computation . . . 462
11.4.5 Trial-and-Error Computation . . . 467
11.4.5.1 Introduction . . . 467
11.4.5.2 Does "Intelligence" Require Trial-and-Error Machines? 470

11.4.5.3 Inductive Inference 476
11.5 Summary 478
11.6 Questions for the Reader 480
11.7 Further Sources of Information 482

IV What Is a Computer Program? 485

12 Algorithms, Programs, Software, and Hardware 487
12.1 Readings: 488
12.2 What Is a Computer Program? 489
12.3 What Is a Program and Its Relation to Algorithms? 490
12.4 What Is Software and Its Relation to Programs and to Hardware? . . . 492
12.4.1 Etymology of 'Software' 492
12.4.2 Software and Music 493
12.4.3 The Dual Nature of Programs 494
12.4.4 Three Theories of Software 495
12.4.4.1 Moor's Theory of the Nature of Software 495
12.4.4.1.1 Levels of Understanding. 495
12.4.4.1.2 Moor's Definition of 'Program' 497
12.4.4.1.2.1 Instructions. 498
12.4.4.1.2.2 "Following Instructions". 499
12.4.4.1.3 Moor's Definitions of Software and Hardware. 500
12.4.5 Suber's Theory of the Nature of Software 503
12.4.6 Colburn's Theory of the Nature of Software 506
12.5 Summary 510
12.6 Questions for the Reader 511

13 Copyright vs. Patent 513
13.1 Readings: 514
13.2 Introduction 515
13.3 Preliminary Considerations 518
13.4 Copyright 520
13.5 Patent 524
13.6 Virtual Machines 526
13.7 Samuelson's Analysis 528
13.8 Allen Newell's Analysis 535

14 What Is Implementation? 537
14.1 Readings: 538
14.2 Introduction 539
14.2.1 Implementation vs. Abstraction 540
14.2.2 Implementations as Role Players 541
14.2.3 Abstract Data Types 542
14.2.4 The Structure of Implementation 543

14.3 Implementation as Semantic Interpretation 544
14.3.1 What Kind of Relation Is Implementation? 544
14.3.2 What Is Semantic Interpretation? 546
14.3.2.1 Formal Systems 546
14.3.2.2 Syntax 547
14.3.2.3 Semantic Interpretation 548
14.3.3 Two Modes of Understanding 551
14.4 Chalmers's Theory of Implementation 554
14.4.1 Introduction 554
14.4.2 An Analysis of Chalmers's Theory 555
14.4.3 Rescorla's Analysis of Chalmers's Theory 559

15 Are Programs Theories? 565
15.1 Readings: 566
15.2 Introduction 567
15.3 Simulations, Theories, and Models 567
15.3.1 Simulations 567
15.3.1.1 Simulation vs. Emulation 567
15.3.1.2 Simulation vs. Imitation 568
15.3.2 Theories 570
15.3.3 Models 571
15.4 Computer Programs as Theories 572
15.4.1 Introduction 572
15.4.2 Simon & Newell's Argument from Analogy 575
15.4.3 Simon's Argument from Prediction 576
15.4.4 Daubert vs. Merrell-Dow 578
15.5 Computer Programs *Aren't* Theories 580
15.5.1 Moor's Objections 580
15.5.2 Thagard's Objections 582
15.6 Questions for the Reader 585

16 Can Computer Programs Be Verified? 587
16.1 Readings: 588
16.2 Introduction 590
16.3 Theorem Verification 594
16.3.1 Theorems and Proofs 594
16.3.1.1 Syntax 594
16.3.1.2 Semantics 595
16.3.2 Programs and Proofs 596
16.3.3 Programs, Proofs, and Formal Systems 598
16.4 Program Verification 600
16.4.1 Introduction and Some History 600
16.4.2 Program Verification by Pre- and Post-Conditions 601
16.4.3 The Value of Program Verification 602
16.5 The Fetzer Controversy 604
16.5.1 Fetzer's Argument against Program Verification 604

16.5.2 The Controversy . . . 607
16.5.3 Barwise's Attempt at Mediation . . . 608
16.6 Summary . . . 609

17 How Do Programs Relate to the World? 615
17.1 Readings: . . . 616
17.2 Introduction . . . 618
17.3 Program Verification, Models, and the World . . . 619
17.3.1 "Being Correct" vs. "Doing What's Intended" . . . 619
17.3.2 Models: Putting the World into Computers . . . 620
17.3.2.1 Creating a Model of the World . . . 620
17.3.2.2 Creating a Computer Representation of the Model . 621
17.3.2.3 Model vs. World . . . 623
17.4 Internal vs. External Behavior: Some Examples . . . 627
17.4.1 Successful Internal Behavior
but Unsuccesful External Behavior . . . 627
17.4.1.1 The Blocks-World Robot . . . 627
17.4.1.2 Cleland's Recipe for Hollandaise Sauce . . . 627
17.4.2 Same Internal Behavior but Different External Behavior . . . 627
17.4.2.1 Fodor's Chess and War Programs . . . 627
17.4.2.2 Rescorla's GCD Computers . . . 629
17.4.2.3 AND-Gates or OR-Gates? . . . 629
17.5 Two Views of Computation . . . 631
17.6 Inputs, Turing Machines, and Outputs . . . 633
17.6.1 Introduction . . . 633
17.6.2 The Turing-Machine Tape as Input-Output Device . . . 633
17.6.3 Are Inputs Needed? . . . 634
17.6.4 Are Outputs Needed? . . . 635
17.6.5 When Are Inputs and Outputs Needed? . . . 636
17.6.6 Must Inputs and Outputs Be Interpreted Alike? . . . 636
17.6.7 Linking the Tape to the External World . . . 639
17.7 Are Programs Intentional? . . . 639
17.7.1 What Is an Algorithm? . . . 640
17.7.2 Do Algorithms Need a Purpose? . . . 640
17.7.3 Marr's Analysis of an Algorithm's Purpose . . . 641
17.7.4 Are Purposes Eliminable? . . . 642
17.7.5 Can Algorithms Have More than One Purpose? . . . 645
17.7.6 What If *G* and *A* Come Apart? . . . 646
17.8 Do We Compute with Symbols
or with Their Meanings? . . . 648
17.8.1 What Is This Turing Machine Doing? . . . 648
17.8.2 Syntactic Semantics . . . 651
17.8.2.1 Syntax vs. Semantics . . . 651
17.8.2.2 Syntactic Semantics . . . 652
17.8.2.3 Syntactic Semantics and Procedural Abstraction . . 653
17.8.2.4 Internalization . . . 655

17.9 Content and Computation 657
17.9.1 Introduction 657
17.9.2 Symbols: Marks vs. Meanings 659
17.9.3 Shagrir's "Master Argument" 663
17.10 Summary 665
17.11 Questions for the Reader 667

V Computer Ethics and Artificial Intelligence 669

18 Computer Ethics I: Decisions 673
18.1 Readings: 674
18.2 Introduction 675
18.3 What Is a Decision? 675
18.4 *Do* Computers Make Decisions? 676
18.5 Are Computer Decisions *Rational*? 678
18.6 Should Computers Make Decisions *for* Us? 679
18.7 Should Computers Make Decisions *with* Us? 679
18.8 Should We *Trust* Decisions Computers Make? 681
18.8.1 The Bias Problem 683
18.8.2 The Black-Box Problem 684
18.9 Are There Decisions Computers *Must* Make for Us? 687
18.10 Are There Decisions Computers *Shouldn't* Make? 688
18.11 Discussion Questions for the Reader 690

19 Artificial Intelligence 693
19.1 Required Readings: 694
19.2 Introduction 695
19.3 What Is AI? 695
19.3.1 Definitions and Goals of AI 695
19.3.2 Artificial Intelligence as Computational Cognition 696
19.4 The Turing Test 698
19.4.1 How Computers Can Think 698
19.4.2 The Imitation Game 699
19.4.3 Thinking vs. "Thinking" 702
19.5 Two Digressions 708
19.5.1 The "Lovelace Objection" 708
19.5.2 Turing on Intelligent Machinery 709
19.6 The Chinese Room Argument 710
19.6.1 Two Chinese Room Arguments 712
19.6.2 The Argument from Biology 713
19.6.2.1 Causal Powers 713
19.6.2.2 The Implementation Counterargument 715
19.6.3 The Argument from Semantics 716
19.6.3.1 The Premises 716
19.6.3.2 Which Premise Is at Fault? 717

19.6.3.3 Semiotics 718
19.6.3.4 Points of View 722
19.6.3.5 A Recursive Theory of Understanding 725
19.7 Leibniz's Mill and Turing's "Strange Inversion" 726
19.8 A Better Way 734
19.9 Questions for Discussion 737

20 Computer Ethics II: AI **739**
20.1 Readings: 740
20.2 Introduction 741
20.3 Is AI Possible in Principle? 741
20.4 What Is the Measure of Success? 743
20.5 Rights 746
20.6 Immoral Experiments 747
20.7 AIs and Morality 749
20.8 NOTES FOR NEXT DRAFT 751
20.9 Further Sources of Information 754

VI Closing Remarks **759**

21 Summary **761**
21.1 Readings: 762
21.2 What Is Philosophy? 763
21.3 What Is Computer Science? 763
21.4 Is Computer Science a Kind of Science or of Engineering? 764
21.4.1 What Is Science? 764
21.4.2 What Is Engineering? 764
21.5 What Does Computer Science Study? 765
21.5.1 What Is a Computer? Historical Answer 765
21.5.2 What Is an Algorithm? Mathematical Answer 765
21.5.3 What Is a Computer? Philosophical Answer 765
21.5.4 What Is an Algorithm? Philosophical Answer 766
21.5.4.1 What Is a Procedure? 766
21.5.4.2 What Is Hypercomputation? 766
21.5.4.3 What Is a Computer Program? 766
21.5.4.3.1 What Is Implementation? 766
21.5.4.3.2 What Is Software? 767
21.5.4.3.3 Can (Should) Software Be Patented, or Copyrighted? 767
21.5.4.3.4 Are Programs Scientific Theories? 767
21.5.4.3.5 What Is the Relation of Programs to the World? 767
21.6 Computer Ethics 768
21.6.1 Are There Decisions Computers Should Never Make? 768
21.6.2 Should We Build an Artificial Intelligence? 768

21.6.3 Philosophy of AI . . . 768
21.7 What Is the Philosophy of Computer Science? . . . 769
21.7.1 The Major Question . . . 769
21.7.2 9 Theses about Computation . . . 769
21.8 NOTE FOR NEXT DRAFT . . . 773

22 Syntax and Semantics **775**

VII Appendices 779

A Argument Analysis and Evaluation **781**
A.1 Readings: . . . 782
A.2 Introduction . . . 783
A.3 A Question-Answer Game . . . 783
A.4 Missing Premises . . . 785
A.5 When Is an Argument a "Good" Argument? . . . 788
A.6 Summary . . . 790
A.7 NOTES FOR NEXT DRAFT . . . 790

B Position-Paper Assignments **795**
B.1 Readings: . . . 796
B.2 Introduction . . . 797
B.3 Position Paper #1: What Is Computer Science? . . . 798
B.3.1 Assignment . . . 798
B.3.1.1 Introduction . . . 798
B.3.1.2 The Argument . . . 798
B.3.2 Suggestions and Guidelines for Peer-Group Editing . . . 801
B.3.3 Grading . . . 803
B.3.3.1 Philosophy of Grading . . . 803
B.3.4 Grading Rubric . . . 803
B.4 Position Paper #2: What Is Computation? . . . 806
B.4.1 Assignment . . . 806
B.4.2 Suggestions and Guidelines for Peer-Group Editing . . . 809
B.4.3 Grading Rubric . . . 811
B.5 Position Paper #3: Is the Brain a Computer? . . . 814
B.5.1 Assignment . . . 814
B.5.2 Suggestions and Guidelines for Peer-Group Editing . . . 815
B.5.3 Grading Rubric . . . 816
B.6 Position Paper #4: What Is a Computer Program? . . . 819
B.6.1 Assignment . . . 819
B.6.1.1 The Argument . . . 819
B.6.1.2 Thinksheet . . . 822
B.6.2 Suggestions and Guidelines for Peer-Group Editing . . . 826
B.6.3 Grading Rubric . . . 827
B.7 Position Paper #5: Can Computers Think? . . . 830

B.7.1 Assignment 830
B.7.2 Suggestions and Guidelines for Peer-Group Editing 832
B.7.3 Grading Rubric 833
B.8 Optional Position Paper: A Competition 838

C Term Paper Topics **839**

D Final Exam **841**
D.1 To the Instructor 841
D.2 A Final Exam 844
D.3 NOTES FOR NEXT DRAFT 846

If you begin with Computer Science, you will end with Philosophy.[1]

[1] "Clicking on the first link in the main text of a Wikipedia article, and then repeating the process for subsequent articles, usually eventually gets you to the Philosophy article. As of May 26, 2011, 94.52% of all articles in Wikipedia lead eventually to the article Philosophy." —"Wikipedia:Getting to Philosophy" http://en.wikipedia.org/wiki/Wikipedia:Getting_to_Philosophy

If you begin with "Computer Science", you will end with "Philosophy" (in 12 links).

Preface

Version of 6 September 2019; DRAFT © 2004–2019 by William J. Rapaport

To readers who have found this document through Brian Leiter's *Leiter Reports* blog[2] or through the FreeTechBooks website[3] (with which I have no affiliation): Welcome!

This document is a continually-being-revised draft of a textbook on the philosophy of computer science. It is based on a course I created for the Department of Computer Science and Engineering and the Department of Philosophy at the State University of New York at Buffalo.

The syllabus, readings, assignments, and website for the last version of the course are online at:

http://www.cse.buffalo.edu/~rapaport/584/

The course is described in:

> Rapaport, William J. (2005), "Philosophy of Computer Science: An Introductory Course", *Teaching Philosophy* 28(4): 319–341, http://www.cse.buffalo.edu/~rapaport/Papers/rapaport_phics.pdf

A video of my Herbert Simon Keynote Address at NACAP-2006 describing the course can be downloaded from:

http://www.hass.rpi.edu/streaming/conferences/cap2006/nacp_8_11_2006_9_1010.asx

The current draft of the book is just that: a draft of a work in progress. Early chapters contain "Notes for the Next Draft" at the end of each chapter. These represent material that I have not yet fully incorporated into the body of the text. There are also one or two unfinished or unwritten chapters.

[2] http://leiterreports.typepad.com/blog/2013/10/a-philosophy-of-computer-science-textbook.html
[3] https://www.freetechbooks.com/philosophy-of-computer-science-t1045.html

A note on Web addresses (URLs): URLs were accurate at the time of writing. Some will change or disappear. Some documents with no public URLs may eventually gain them. When in doubt, try a Google (or other) search for the document; articles can often be found by using a search string consisting of: the author(s) last name(s), followed by: the title of the document enclosed in quotation marks. (For example, to find Rapaport 2005c, search for "rapaport "philosophy of computer science"". Documents that have disappeared can sometimes be found at the Internet Archive's Wayback Machine, https://archive.org/web/

Please send any comments, suggestions for improvement, etc., to me via email at:

rapaport@buffalo.edu

And thanks for looking!

Changes since previous version (dated 29 August 2019):

Major changes to: Ch. 5, and Ch. 20,
Minor changes to: this Preface, Ch. 3, Ch. 17, Ch. 18, and the Bibliography.

Acknowledgments

Thanks especially to Peter Boltuc, Selmer Bringsjord, Jin-Yi Cai, Edgar Daylight, Peter Denning, Eric Dietrich, William D. Duncan, Frank Fedele, Robin K. Hill, Johan Lammens, Nelson Pole, Thomas M. Powers, Michael I. Rapaport, Stuart C. Shapiro, Aaron Sloman, and Matti Tedre, as well as to Russ Abbott, Khaled Alshammari, S.V. Anbazhagan, Roger Derham, Pablo Godoy, David Miguel Gray, Nurbay Irmak, Patrick McComb, Cristina Murta, Alexander Oblovatnyi, Richard M. Rubin, Seth David Schoen, Stephen Selesnick, Mark Staples, Dean Waters, Nick Wiggershaus, and Sen Zhang for comments on, or suggestions for, this early draft or its ancestors.

To Do

Version of 5 July 2019; DRAFT © 2004–2019 by William J. Rapaport

1. Put a real summary section at the end of each chapter.
2. Move some of the appendix exercises to the ends of appropriate chapters. Leave the answerd, comments, and grading rubrics in the appendices.
3. Fix URLs, using BASHurl
4. Either include somewhere, or recommend to instructor to do, a slow reading of (at least part of) one paper.
5. add exercises
    - put exercises at ends of approp chaps as per syllabus?
6. update section on reasoning to include sample valid, invalid args
7. add chap on syntactic semantics
8. write a new closing chapter/article giving *my* phics.
9. add an "other topics" chapter, for things like the philosophy of the web, social interaction, etc.
10. check Further Readings for soundness & completeness w.r.t. online bibs at http://www.cse.buffalo.edu/$rapaport/584/S10/directory.html
11. Incorporate all Further Readings sections at ends of (early) chapters into the bodies of the chapters, omitting any that don't have an obvious reason for being there.
12. Include all relevant emails from /web/faculty/rapaport/584/S07/EMAIL and /web/faculty/rapaport/584/S10/EMAIL as well as anything else relevant from my course online files.
13. At first occurrence of single quotes in text, add an explanatory footnote
14. Try to use **boldface** to highlight important phrases and *italics* for emphasis. And add a note explaining that usage.
15. Incorporate (Rapaport, 2017b) into Ch. 22.
16. Fix use of BASHcite:
    - Epigraphs: Use —FirstName+BASHcitet
    - displayed quotes: Use BASHcitep
    - other in-text uses: Use BASHcitet or citealt, in most cases, rather than citep

17. Might want to have a short section at the end on what the reader should read next if they want to continue the conversation. Two obvious choices are Piccinini 2015 and Turner 2018. On the latter, the section on programming language paradigms is especially useful as covering a topic that I don't.

18. If Ch. 17 ever gets revised, might want to move section on Smith back into Ch. 16, and then reorganize the rest of the chapter in terms of arguments for and against "Do A" vs. "To G, do A":

    - For Do A: Piccinini, Egan, Dennett, Rescorla?, et al.
    - For to G, do A: Hill, Suber, Shagrir, Marr?, Rescorla!, et al.

    Also: Turner 2018, p 201 says:

    > There are two conceptually significant notions of correctness for computational artifacts. The first operates between the function and the structure: it is the correctness of the design. The structural description of the artifact must be in *accord* with the functional demands. This links two abstract structures. A second notion arises from the relationship between the structure and the artifact; it demands the correctness of the implementation. Here the artifact must be in accord with the structural demands, and it is here that the second notion, the one that links the abstract world with the physical, appears. On the face of it, the first notion is an abstract mathematical one, and the second an empirical one.

    Turner's first notion concerns the relation between a specification and a computer program that implements it; it is the subject of the field of program correctness. His second notion concerns the relation between a program and its implementation and execution on a physical computer; it is the subject of Fetzer's objections.

    But there is a third notion: Smith's relation between a real-world problem and a model (that is, a specification) of it. So, where Fetzer extends the correctness issue from the abstract program to the real-world implementation, Smith extends it in the other direction from the abstract specification to the original real-world problem that generated it.

Part I

Philosophy and Computer Science

Chapter 1

What Is Philosophy of Computer Science?

Version of 29 January 2019; DRAFT © 2004–2019 by William J. Rapaport

> Many people "know about modern electronics in general and computers in particular. They know about code and the compilation process that turns source code into binary executable code. *Computation theory is something different. It is an area of mathematics that overlaps with philosophy.*" —"PolR" (2009, my italics)

> There is no way of telling upstream how great an impact any specific bit of research will have. ... Who would have guessed that the arcane *research done by the small set of mathematicians and philosophers* working on formal logic a century ago would lead to the development of computing, and ultimately to completely new industries, and to the reconfiguring of work and life across the globe? —Onora O'Neill (2013, p. 8, my italics)

> There is no such thing as philosophy-free science, just science that has been conducted without any consideration of its underlying philosophical assumptions. —Daniel C. Dennett (2013a, p. 20)

1.1 Recommended Readings:

1. Strongly Recommended:
   - Scheutz, Matthias (2003), "Computation, Philosophical Issues about", *Encyclopedia of Cognitive Science* (London: Macmillan): 604–610.
     - http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/Reading%20Materials/scheutz%20Comp%20Philo%20Issues.pdf
     - http://home.deib.polimi.it/schiaffo/TFI/scheutz_2002.pdf
2. Recommended:
   - Simon, Herbert A. (1977), "What Computers Mean for Man and Society", *Science* 195 (4283, 18 March): 1186–1191, http://posner.library.cmu.edu/Collections/simon/box00083/fld06642/bdl0003/doc0001/01000002.pdf
   - Turner, Raymond; & Eden, Amnon H. (2017), "The Philosophy of Computer Science", in Edward N. Zalta (ed.), *Stanford Encyclopedia of Computer Science*, http://plato.stanford.edu/entries/computer-science/

1.2 What This Book Is About

> My mind does not simply receive impressions. It talks back to the authors, even the wisest of them, a response I'm sure they would warmly welcome. It is not possible, after all, to accept passively everything even the greatest minds have proposed. One naturally has profound respect for Socrates, Plato, Pascal, Augustine, Descartes, Newton, Locke, Voltaire, Paine, and other heroes of the pantheon of Western culture; but each made statements flatly contradicted by views of the others. So I see the literary and philosophical tradition of our culture not so much as a storehouse of facts and ideas but rather as a hopefully endless Great Debate at which one may be not only a privileged listener but even a modest participant. —Steve Allen (1989, p. 2), as cited in Madigan 2014, p. 46.

> As [the logician] Harvey Friedman has suggested, every morning one should wake up and reflect on the conceptual and foundational significance of one's work. — Robert Soare (1999, p. 25, my interpolated brackets)

This book looks at some of the central issues in the philosophy of computer science. It is not designed to answer all (or even any) of the philosophical questions that can be raised about the nature of computing, computers, and computer science. Rather, it is designed to "bring you up to speed" on a conversation about these issues—to give you some background knowledge—so that you can read the literature for yourself and perhaps become part of the conversation by contributing your own views.

This book is intended for readers who might know some philosophy but no computer science, readers who might know some computer science but no philosophy, and readers who know little or nothing about both! So, although most of the book will be concerned with what *computer science* is, we will begin by asking: **What is philosophy?** And, in particular: What is "the philosophy of X?" (where X = things like: science, psychology, history, etc., and, of course, computer science).

Then we will begin our inquiry into the philosophy of computer science by asking: **What is computer science?** To answer this, we will need to consider a series of questions, each of which leads to another: Is computer science a science, a branch of engineering, some combination of them, or something else altogether? And to answer those questions, we will need to ask **what science is** and **what engineering is.**

Whether science or engineering, computer science is surely scien*tific*, so we next ask: **What is computer science a (scientific) study *of*?** *Computers?* If so, then **what is a computer?** Or is computer science a study of *computation*? If so, then **what is computation?** Computations are said to be *algorithms*, so **what is an algorithm?** Algorithms are said to be procedures, or recipes, so **what is a procedure?** What is a recipe? **What is the Church-Turing Computability Thesis** (that our intuitive notion of computation is completely captured by the formal notion of Turing-machine computation—and what is a Turing machine)? **What is "hypercomputation"** (i.e., the claim that the intuitive notion of computation goes beyond Turing-machine computation)?

Computations are expressed in computer programs, which are executed by computers, so **what is a computer program**? Are computer programs "implementations"

of algorithms? If so, then **what is an implementation?** Programs typically have real-world effects, so **how are programs and computation related to the world?** Some programs, especially in the sciences, are designed to model or simulate or explain some real-world phenomenon, so **can programs be considered to be (scientific) theories?** Programs are usually considered to be "software", and computers are usually considered to be "hardware", but **what is the difference between software and hardware?** Programs are texts written in a (programming) language, and linguistic texts are legally copyrightable. But some programs are engraved on CDs and, when installed in a computer, turn the computer into a (special-purpose) machine, which is legally patentable. Yet, legally, nothing can be both copyrightable *and* patentable, so **are programs copyrightable texts, or are they patentable machines?** Computer programs are notorious for having "bugs", which are often only found after the program has been tested, but **can computer programs be logically verified** before testing?

Next, we turn to some of the issues in the **philosophy of artificial intelligence** What is artificial intelligence (AI)? What is the relation of computation to cognition? Can computers think? What are the Turing Test and the Chinese Room Argument? Very briefly: The Turing Test is a test proposed by one of the creators of the field of computation to determine whether a computer can think. The Chinese Room Argument is a thought experiment devised by a philosopher, which is designed to show that the Turing Test won't work.

Finally, we consider two questions in **computer ethics**, which, at the turn of the century, were not much discussed, but are now at the forefront of computational ethical debates: (1) Should we trust decisions made by computers? (Moor, 1979)—a question made urgent by the advent of automated vehicles. And (2) should we build "intelligent" computers? Do we have moral obligations towards robots? Can or should they have moral obligations towards us?

Along the way, we will look at how philosophers reason and evaluate logical arguments, and there will be some suggested writing assignments designed to help focus your thinking about these issues.[1]

1.3 What This Book Is *Not* About

Have I left anything out? Most certainly! I do not claim that the questions raised above and discussed in this book exhaust the philosophy of computer science. They are merely a series of questions that arise naturally from our first question: What is computer science?

But there are many other issues in the philosophy of computer science. Some are included in a topic sometimes called *philosophy of computing*. Here are some examples: Consider the ubiquity of computing—your smartphone is a computer; your car has a computer in it; perhaps someday your refrigerator or toaster or bedroom wall will

[1] Computer science students take note: Computer Science Curricula 2013 covers precisely these sorts of argument-analysis techniques under the headings of Discrete Structures [DS]/Basic Logic, DS/Proof Techniques, Social Issues and Professional Practice [SP] (in general), and SP/Analytical Tools (in particular). Many other CS2013 topics also overlap those in the philosophy of computer science. See http://ai.stanford.edu/users/sahami/CS2013/

contain (or even be) a computer. How will our notion of computing change because of this ubiquity? Will this be a good or a bad thing? Another topic is the role of the Internet. For instance, Tim Berners-Lee, who created the World Wide Web, has argued that "Web science" should be its own discipline (Berners-Lee et al., 2006), (Lohr, 2006)). And there are many issues surrounding the social implications of computers in general and of social media on the Internet (and the World Wide Web) in particular.

Further Reading:
On social implications, see, especially, (Weizenbaum, 1976) and (Simon, 1977), the penultimate section of which ("Man's View of Man") can be viewed as a response to (Weizenbaum, 1976); see also (Dembart, 1977) for a summary and general discussion.

For a discusson of social implications of the use of computers and the Internet, be sure to read E.M. Forster's classic short story "The Machine Stops",
http://archive.ncsa.illinois.edu/prajlich/forster.html:

> It is a chilling ... masterpiece about the role of technology in our lives. Written in 1909, it's as relevant today as the day it was published. Forster has several prescient notions including instant messages (email!) and cinematophotes (machines that project visual images). —Paul Rajlich (from the above-cited website)

Other issues in the philosophy of computer science more properly fall under the heading of the *philosophy of AI*. As noted, we will look at *some* of these in this book, but there are many others that we won't cover, even though the philosophy of AI is a proper subset of the philosophy of computer science.

Another active field of investigation is the *philosophy of information*. As we'll see in Chapter 3, §3.8, computer science is sometimes defined as the study of how to process information, so the philosophy of information is clearly a close cousin of the philosophy of computer science. But I don't think that either is included in the other; they merely have a non-empty intersection. If this is a topic you wish to explore, take a look at some of the books and essays cited in Chapter 3, at the end of §3.8.

Finally, there are a number of philosophers and computer scientists who have discussed topics related to what I am calling the philosophy of computer science whom we will not deal with at all (such as the philosophers Martin Heidegger and Hubert L. Dreyfus (Dreyfus and Dreyfus, 1980), (Dreyfus, 2001), and the computer scientist Terry Winograd (Winograd and Flores, 1987)).[2] An Internet search (for example: "Heidegger "computer science"") will help you track down information on these thinkers and others not mentioned in this book.

But I think that our questions above will keep us busy for a while, as well as prepare you for examining some of these other issues. Think of this book as an extended "infomercial" to bring you up to speed on the computer-science–related aspects of a philosophical conversation that has been going on for over 2500 years, to enable you to join in the conversation.[3]

[2] One philosopher of computer science calls them the "Dark Side philosophers", because they tend not to be sympathetic to computational views of the world.

[3] In 2006, responding to a talk that I gave on the philosophy of computer science, Selmer Bringsjord (a philosopher and cognitive scientist who has written extensively on the philosophy of computer science) said,

So, let's begin ...

... but, before we do, on the next page you will find some ...

"Philosophy of Computer Science... is in its infancy" (Bringsjord, 2006). This may be true as a discipline so called, but there have been philosophical investigations of computer science and computing since at least (Turing, 1936), and James H. Moor's work goes back to the 1970s (e.g., (Moor, 1978)).

1.4 . . . Further Sources of Information

Each chapter will end with a list of essays, books, or websites that discuss some or all of the topics covered—but not explicitly cited—in the chapter (though some may be cited in other chapters). They will also include items that extend the discussion to other topics that were not covered. If the topic of one of these is not obvious from its title, I will add an annotation explaining why I think that it's interesting.

1. Books and Essays:

   - Burkholder, Leslie (ed.) (1992), *Philosophy and the Computer* (Boulder, CO: Westview Press).
   - Bynum, Terrell Ward (2010), "Philosophy in the Information Age", *Metaphilosophy* 41(3) (april): 420–442.
   - Bynum, Terrell Ward; & Moor, James H. (eds.) (2000), *The Digital Phoenix: How Computers Are Changing Philosophy, Revised Edition* (Oxford: Blackwell).
   - Colburn, Timothy (2000), *Philosophy and Computer Science* (Armonk, NY: M.E. Sharpe).
   - Colburn, Timothy (2006), "What Is Philosophy of Computer Science?", extended abstract of paper presented at the European Conference on Computing and Philosophy (ECAP'06, 22–24 June, Trondheim, Norway) http://tinyurl.com/Colburn06.[4]
   - Dodig-Crnkovic, Gordana (2006), "What Is Philosophy of Computer Science? Experience from the Swedish National Course", paper presented at the European Conference on Computing and Philosophy (ECAP'06, 22–24 June, Trondheim, Norway), http://www.cse.buffalo.edu/∼rapaport/584/Dodig-Crnkovic.pdf
   - Eden, Amnon H.; & Turner, Raymond (eds.), Special Issue on the Philosophy of Computer Science, *Minds and Machines* 21(2) (Summer).
     - Table of contents: http://link.springer.com/journal/11023/21/2/page/1
   - Floridi, Luciano (1999), *Philosophy and Computing: An Introduction* (London: Routledge).
   - Floridi, Luciano (ed.) (2004), *The Blackwell Guide to the Philosophy of Computing and Information* (Malden, MA: Blackwell).
   - Longo, Giuseppe (ed.) (1999), *Philosophy of Computer Science*, special issue of *The Monist* 82(1).
   - Magnani, Lorenzo (ed.) (2006), *Computing and Philosophy: Proceedings of the Second European Conference, Computing and Philosophy (ECAP2004-Italy)* (Pavia, Italy: Associated International Academic Publishers).
   - Moor, James H.; & Bynum, Terrell Ward (eds.) (2002), *Cyberphilosophy: The Intersection of Computing and Philosophy* (Malden, MA: Blackwell).
   - Pylyshyn, Zenon W. (1992), "Computers and the Symbolization of Knowledge", in Ralph Morelli, W. Miller Brown, Dina Anselmi, Karl Haberlandt, & Dan Lloyd (eds.), *Minds, Brains & Computers: Perspectives in Cognitive Science and Artificial Intelligence* (Norwood, NJ: Ablex): 82–94; preprint (1987), http://ruccs.rutgers.edu/ftp/pub/papers/suffolk.pdf

[4] https://static.aminer.org/pdf/PDF/000/211/881/some_relationships_between_logics_of_programs_and_complexity_theory_extended.pdf

- Rapaport, William J. (2005), "Philosophy of Computer Science: An Introductory Course", *Teaching Philosophy* 28(4): 319–341.
- Smith, Brian Cantwell (1996), *On the Origin of Objects* (Cambridge, MA: MIT Press).
  - A study of the philosophical foundations of computing.
  - Koepsell, David R. (1998), Review of Smith 1996, *Philosophical Psychology* 11(3): 389–391.
  - Loui, R.P. (1998), Review of Smith 1996, *Artificial Intelligence* 106: 353–358.
- Smith, Brian Cantwell (2002), "The Foundations of Computing", in Scheutz, Matthias (ed.), *Computationalism: New Directions* (Cambridge, MA: MIT Press): 23–58.
  - DeJohn, Jerry; & Dietrich, Eric (2003), "Subvert the Dominant Paradigm! A review of *Computationalism: New Directions*, edited by Matthias Scheutz", *Journal of Experimental & Theoretical Artificial Intelligence* 15(4) (October-December): 375–382, esp. §3.1 (on Smith), pp. 378–379.
  - Piccinini, Gualtiero (2005), Review of Scheutz 2002, in *Philosophical Psychology* 18(3) (June): 387–391, esp. p. 389 (on Smith).
- Tedre, Matti (2007), "Know Your Discipline: Teaching the Philosophy of Computer Science", *Journal of Information Technology Education* 6: 105–122.
- Turner, Raymond; & Eden, Amnon H. (eds.) (2007), *Philosophy of Computer Science*, special issue of *Minds and Machines* 17(2) (Summer): 129–247.
- Turner, Raymond; & Eden, Amnon H. (2008), "The Philosophy of Computer Science", special issue of *Journal of Applied Logic* 6(4) (December): 459–552.

2. Websites:

- Aaronson, Scott (2011), "Philosophy and Theoretical Computer Science", http://stellar.mit.edu/S/course/6/fa11/6.893/
- Brey, Philip; & Søraker, Johnny Hartz (2008), "Philosophy of Computing and Information Technology", http://www.idt.mdh.se/kurser/comphil/2011/PHILO-INFORM-TECHNO-20081023.pdf
- Brown, Curtis (2000, Fall), Seminar: Philosophy and Computers, http://www.trinity.edu/cbrown/philandcomp/
- Commission for the History and Philosophy of Computing, http://www.hapoc.org/
- Dai, Yi; Lamberov, Lev; & Ostermann, Klaus (organizers) (2014), "Discussion Seminar: Philosophy of Programming Languages", Department of Mathematics and Computer Science, University of Marburg (Germany), https://github.com/plmday/DS-PhPL
- Hill, Robin K. (2014), "Teaching the Philosophy of Computer Science" (blog), http://teachingphilofcs.blogspot.com/
- Philosophy of Computer Science (ECAP'06), http://pcs.essex.ac.uk/ecap06/cfp.html
- Philosophy of Computer Science: Online and Offline Resources, http://pcs.essex.ac.uk/
- Philosophy of Computer Science Course (CD5650, 2004), http://www.idt.mdh.se/∼gdc/PI_04/PI-network-course.htm

- Philosophy of Computing and Informatics Network,
  http://www.idt.mdh.se/∼gdc/pi-network.htm
- Price, Colin B. (2007), "Concepts and Philosophy of Computing", University of Worcester (UK), http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp
- Tedre, Matti (2007, Winter-Spring), The Philosophy of Computer Science (175616), University of Joensuu, Finland,
  http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/
  - Contains links to detailed lecture notes.
  - See esp.: "Introduction to the Course",
    http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes1.pdf

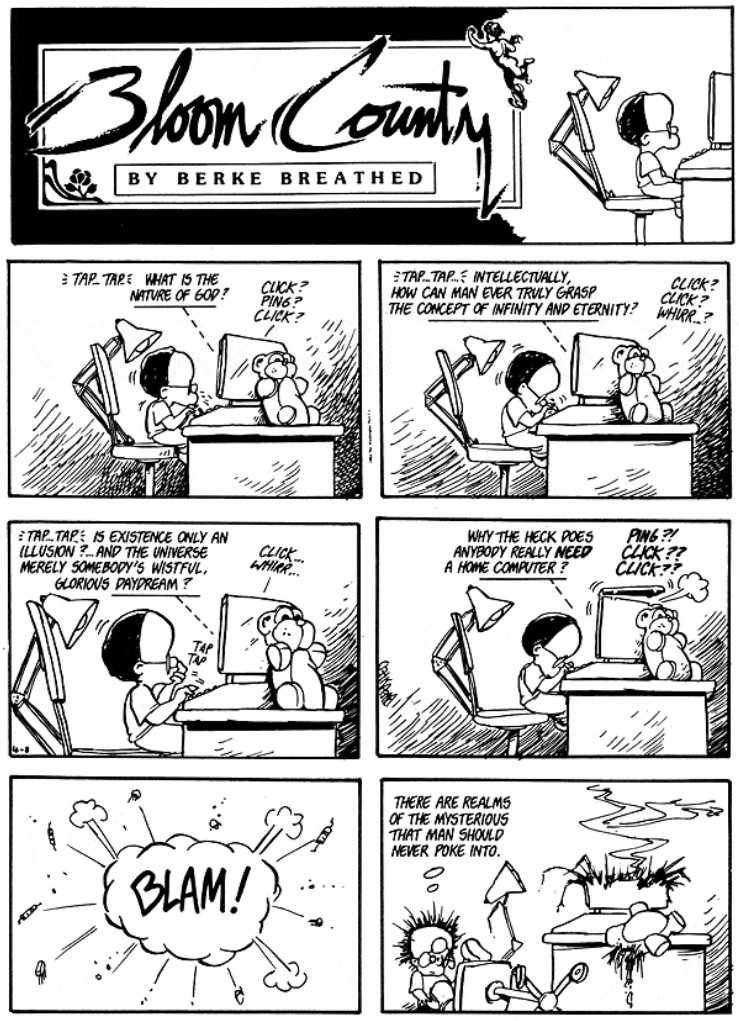


Figure 1.1: © 4 April 1984 by Berkeley Breathed

Chapter 2

What Is Philosophy?

Version of 7 July 2019; DRAFT © 2004–2019 by William J. Rapaport

> 'Two years!' said Dantés. 'Do you think I could learn all this in two years?'
> 'In their application, no; but the principles, yes. Learning does not make one learned: there are those who have knowledge and those who have understanding. The first requires memory, the second philosophy.'
> 'But can't one learn philosophy?'
> 'Philosophy cannot be taught. Philosophy is the union of all acquired knowledge and the genius that applies it. . . '
> (Dumas, 1844, *The Count of Monte Cristo*, Ch. 17, pp. 168–169)

> "Philosophy is the microscope of thought."
> (Hugo, 1862, *Les Misérables*, Vol. 5, Book Two, Ch. II, p. 1262)

> "Philosophy . . . works against confusion"
> —John Cleese (2012), "Twenty-First Century",
> http://www.publicphilosophy.org/resources.html#cleese

> Consider majoring in philosophy. I did. . . . [I]t taught me how to break apart arguments, how to ask the right questions"
> —NPR reporter Scott Simon, quoted in (Keith, 2014)

> "To the person with the right turn of mind, . . . all thought becomes philosophy" (Schwitzgebel, 2012).

> "Philosophy can be any damn thing you want!"
> —John Kearns, personal communication, 7 November 2013

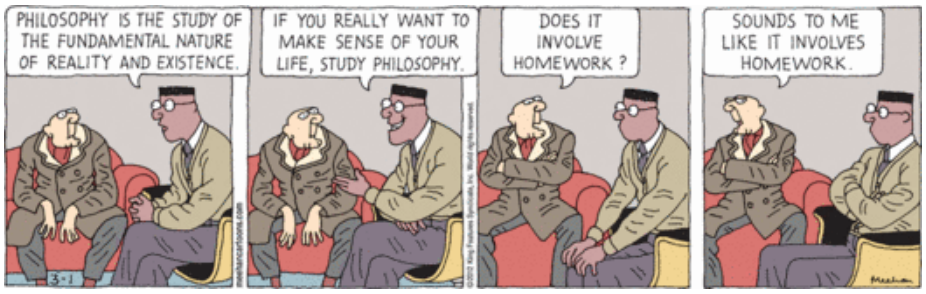


Figure 2.1: © 1 March 2012, King Features Syndicate

2.1 Readings:

1. Very Strongly Recommended:
   - Audi, Robert (1981), "Philosophy: A Brief Guide for Undergraduates" (Newark, DE: American Philosophical Assocation), http://www.apaonline.org/?undergraduates
2. Strongly Recommended:
   - Plato, *The Apology* (various versions are online: search for "Plato Apology")
     - Plato's explanation of what Socrates thought that philosophy was all about; a good introduction to the skeptical, questioning nature of philosophy.
3. Recommended:
   - Colburn, Timothy R. (2000), *Philosophy and Computer Science* (Armonk, NY: M.E. Sharpe):
     (a) Ch. 3: "AI and the History of Philosophy" (pp. 19–40)
     (b) Ch. 4: "AI and the Rise of Contemporary Science and Philosophy" (pp. 41–50)
     - Some of the material may be online at the Google Books (http://books.google.com/) website for this book: http://tinyurl.com/Colburn00

2.2 Introduction

"What is philosophy?" is a question that is not a proper part of the philosophy of computer science. But, because many readers may not be familiar with philosophy, I want to begin our exploration with a brief introduction to how I think of philosophy, and how I would like non-philosophical readers who are primarily interested in computer science to think of it.

So, in this chapter, I will give you *my* definition of 'philosophy'. We will also examine the principal methodology of philosophy: the evaluation of logical arguments (see §2.6.1 and Appendix A).

2.3 A Definition of 'Philosophy'

The word 'philosophy'[1] has a few different meanings. When it is used informally, in everyday conversation, it can mean an "outlook", as when someone asks you what your "philosophy of life" is. The word 'philosophical' can also mean something like "calm", as when we say that someone takes bad news "very philosophically" (that is, very calmly).

But, in this chapter, I want to explicate the *technical* sense of *modern, analytic, Western* philosophy—that is, the kind of philosophy that has been done since at least Socrates's time. 'Modern philosophy' is itself a technical term that usually refers to the kind of philosophy that has been done since René Descartes in the early 17th century (Nagel, 2016). (Descartes lived from 1596 to 1650, almost 400 years ago.) It is "analytic" in the sense that it is primarily concerned with the logical analysis of concepts (rather than literary, poetic, or speculative approaches). And it is "Western" in the sense that it has been done by philosophers working primarily in Europe (especially in Great Britain) and North America—though, of course, there are very many philosophers who do analytic philosophy in other areas of the world (and there are many other kinds of philosophy).

[1] Many philosophers have adopted a convention that *single quotes* are used to form the name of a word or expression. So, when I write this:

'philosophy'

I am not talking about philosophy! Rather, I am talking about the 10-letter word spelled p-h-i-l-o-s-o-p-h-y. This use of single quotes enables us to distinguish between a *thing* that we are talking about and the *name* or *description* that we use to talk about the thing. This is like the difference between a *number* (a thing that mathematicians talk about) and a *numeral* (a word or symbol that we use to talk about numbers). It is called the 'use-mention distinction'; see the *Wikipedia* article at:
http://en.wikipedia.org/wiki/Use-mention_distinction

I will use *double quotes* when I am directly quoting someone. I will also sometimes use double quotes as "scare quotes", to indicate that I am using an expression in a special or perhaps unusual way (as I just did). And I will use double quotes to indicate the *meaning* of a word or other expression.

Further Reading:
On different styles of philosophy, see the University of Michigan Department of Philosophy's website at https://web.archive.org/web/20190617023651/http://lsa.umich.edu/philosophy/undergraduates/graduate-work/styles-of-philosophy.html and commentary on it at https://leiterreports.typepad.com/blog/2007/04/styles_of_philo.html

On non-Western philosophy, consider this observation:

> ... there are good reasons to doubt that Greece, India, and China were the only societies that practiced philosophy, indeed to doubt that philosophy needed to be born or "invented" in the first place. Why not assume that philosophy is just a universal aspet of human culture? To explore this hypothesis, we need some idea of what it means for thoughts to be "philosophical." This is a notoriously difficult question to answer, though most people probably feel that philosophy is like pornography: we know it when we see it. Provisionally, we might agree to apply the term to all abstract reflection on deep questions concering ethics, knowledge, being, language, and so on. If that is what we are looking for, then perhaps we will find philosophy just about everywhere. (Adamson, 2019).

Western philosophy began in ancient Greece. Socrates (470–399 B.C.E.,[2] that is, around 2500 years ago) was opposed to the Sophists, a group of teachers who can be caricaturized as an ancient Greek version of "ambulance-chasing" lawyers, "purveyors of rhetorical tricks" (McGinn, 2012b). The Sophists were willing to teach anything (whether it was true or not) to anyone, or to argue anyone's cause (whether their cause was just or not), for a fee.

Like the Sophists, Socrates also wanted to teach and argue, but only to seek wisdom: truth in any field. In fact, the word 'philosophy' comes from Greek roots meaning "love of [*philo*] wisdom [*sophia*]". The reason that Socrates only *sought* wisdom rather than claiming that he *had* it (like the Sophists did) was that he believed that he *didn't* have it: He claimed that he *knew* that he didn't know anything (and that, therefore, he was actually wiser than those who claimed that they *did* know things but who really didn't). As Victor Hugo put it, "the wise one knows that he is ignorant" ("*Le savant sait qu'il ignore*"; cited in O'Toole 2016), or, as the contemporary philosopher Kwame Anthony Appiah sai, in reply to the question "How do you think Socrates would conduct himself at a panel discussion in Manhattan in 2019?":

> You wouldnt be able to get him to make an opening statement, because he would say, "I dont know anything." But as soon as anybody started saying anything, hed be asking you to make your arguments clearer—he'd be challenging your assumptions. He'd want us see that the standard stories we tell ourselves arent good enough. (Libbey and Appiah, 2019)

Socrates's student Plato (430–347 B.C.E.), in his dialogue *Apology*, describes Socrates as playing the role of a "gadfly", constantly questioning (and annoying!) people about the justifications for, and consistency among, their beliefs, in an effort to find out the truth for himself from those who considered themselves to be wise (but who really weren't). (For a humorous take on this, see Figure 2.2.)

[2] 'B.C.E.' is the abbreviation for 'before the common era'; that is, B.C.E. years are the "negative" years before the year 1, which is known as the year 1 C.E. (for "common era").

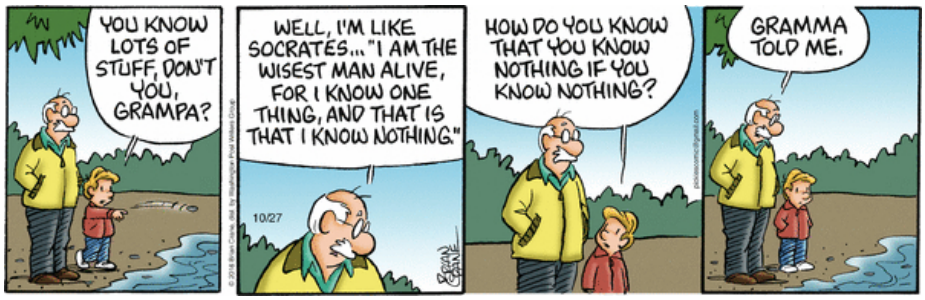


Figure 2.2: ©2016 Brian Crane; http://www.gocomics.com/pickles/2016/10/27

Plato defined 'philosopher' (and, by extension, 'philosophy') in Book V of his *Republic* (line 475c):

> The one who feels no distaste in sampling *every study*, and who attacks the task of learning gladly and cannot get enough of it, we shall justly pronounce the lover of wisdom, the philosopher. (Plato, 1961b, p. 714, my emphasis).

Adapting this, I define 'philosophy' as:

> **the *personal search* for *truth*, in *any* field, by *rational* means.**

This raises several questions:

1. Why only "personal"? (Why not "universal"?)
2. Why is philosophy only the *search* for truth? Can't we succeed in our search?
3. What is "truth"?
4. What does 'any field' mean?
   Is philosophy really the study of anything and everything?
5. What counts as being "rational"?

Let's look at each of these, beginning with the second.

2.4 What Is Truth?

The study of the nature of truth is one of the "Big Questions" of philosophy (along with things like: What is the meaning of life? What is good? What is beauty? and so on).[3] I cannot hope to do justice to it here, but there are two theories of truth that will prove useful to keep in mind on our journey through the philosophy of computer science: the *correspondence* theory of truth and the *coherence* theory of truth.

[3] See Gabriel Segal's response to the question "What is it that is unique to philosophy that distinguishes it from other disciplines?" at http://www.askphilosophers.org/question/5017. We'll return to this in §2.8, below.

2.4.1 The Correspondence Theory of Truth

> The correspondence theory states that a belief is true if and only if that belief corresponds to the facts. ... It captures the idea that truth depends on objective reality—not on us. The problem the correspondence theory has concerns more technical issues such as what a fact is and what the correspondence relation amounts to. (McGinn, 2015a, pp. 148–149)

The word 'true' originally meant "faithful". Such faithfulness requires *two things A* and *B* such that *A* is faithful to *B*. According to the correspondence theory (see (David, 2009)), truth is faithfulness of (A) a *description* of some part of reality to (B) the *reality* that it is a description of. On the one hand, there are *beliefs* (or propositions, or sentences);[4] on the other hand, there is "reality": A belief (or a proposition, or a sentence) is true if and only if ("iff") it corresponds to reality, that is, iff it is faithful to, or "matches", or accurately characterizes or describes reality.

To take a classic example, the three-word English sentence 'Snow is white.' is true iff the stuff in the real world that precipitates in certain winter weather (that is, snow) has the same color as milk (that is, iff it is white). Put somewhat paradoxically (but correctly—see footnote 1 in §2.3) 'Snow is white.' is true iff snow is white.[5]

How do we *determine* whether a sentence (or a belief, or a proposition) is true? On the correspondence theory, in principle, we would have to compare the parts of the sentence (its words plus its grammatical structure, and maybe even the context in which it is thought, uttered, or written) with parts of reality, to see if they correspond. But how do we access "reality"? How can we do the "pattern matching" between our beliefs and reality?

One answer is by sense perception (perhaps together with our beliefs about what we perceive). But sense perception is notoriously unreliable (think about optical illusions, for instance). And one of the issues in deciding whether our *beliefs* are true is deciding whether our *perceptions* are accurate (that is, whether *they* match reality).

So we seem to be back to square one, which gives rise to the coherence theory.

2.4.2 The Coherence Theory of Truth

> The coherence theory states that a proposition is true if and only if that proposition coheres with the other propositions that one believes. ... The problem with the coherence theory is that a belief could be consistent with my other beliefs and yet the whole lot could be false. (McGinn, 2015a, p. 148)

According to the coherence theory of truth (see (Young, 2018)), a set of propositions (or beliefs, or sentences) is true iff:

[4] A "belief", as I am using that term here, is a mental entity, "implemented" (in humans) by certain neuron firings. A "sentence" is a grammatical string of words in some language. And a "proposition" is the meaning of a sentence. These are all rough-and-ready characterizations. Each of these terms has been the subject of much philosophical analysis. For further discussion, see: (Schwitzgebel, 2015), https://en.wikipedia.org/wiki/Sentence_(linguistics), and (King, 2016), respectively.

[5] The standard logical presentation of a correspondence theory of truth is due to Alfred Tarski; see (Hodges, 2014) for an overview and further references, and (Tarski, 1969) for a version aimed at a general audience.

1. they are mutually consistent, and

2. they are supported by, or consistent with, all available evidence;

that is, they "cohere" with each other and with all evidence.

Note that observation statements (that is, descriptions of what we observe in the world around us) are among the claims that must be mutually consistent, so this is *not* (necessarily) a "pie-in-the-sky" theory that doesn't have to relate to the way things really are. It just says that we don't have to have independent access to "reality" in order to determine truth.

2.4.3 Correspondence vs. Coherence

Which theory is correct? Well, for one thing, there are more than two theories: There are several versions of each kind of theory, and there are other theories of truth that don't fall under either category. The most important of the other theories is the "pragmatic" theory of truth (see Glanzberg (2016, §3, http://plato.stanford.edu/entries/truth/#PraThe). Here is one version: "[T]he pragmatic theory of truth ... is that a proposition is true if and only [if] it is useful [that is, "pragmatic", or practical] to believe that proposition" (McGinn, 2015a, p. 148). Another version states that a belief, proposition, or sentence is true iff it continues to be accepted at the limit of inquiry:

> Truth is that to which a belief would tend were it to tend indefinitely to a fixed belief.
>
> —Edwin Martin, Jr., paraphrasing C.S. Peirce; lectures on the theory of knowledge, Indiana University, Spring 1973.[6]

However, "I could have a belief about something that is useful to me but that belief is false" (McGinn, 2015a, p. 149). Similarly, a "fixed" belief that remains "at the limit of inquiry" might still be false.

Fortunately, the answer to which kind of theory is correct (that is, which kind of theory is, if you will excuse the expression, *true*) is beyond our present scope! But note that the propositions that a correspondence theory says are true must be mutually consistent (if "reality" is consistent!), and they must be supported by all available evidence; that is, *a correspondence theory must "cohere"*. Moreover, if you include both propositions and "reality" in one large, highly interconnected network, that network must also "cohere", so the propositions that are true according to *a coherence theory of truth should "correspond to"* (that is, cohere with) *reality*.

Let's return to the question raised in §2.4.1, above: How can we *decide* whether a statement is true? One way that we can determine its truth is *syntactically* (that is, in terms of its grammatical structure only, not in terms of what it means), by trying to *prove* it from axioms via rules of inference.[7] It is important to keep in mind that, when you prove a statement this way, you are not proving that it is true! You are simply proving that it follows logically from certain other statements, that is, that it

[6] For more on Peirce, see footnote 19 in §2.6.1.3, below.

[7] I'll say more about what axioms and rules of inference are in Chs. 14 ("What Is Implementation?") and 22 ("Syntax and Semantics"). For now, just think of proving theorems in geometry or logic.

"coheres" in a certain way with those statements. But, if the starting statements—the axioms—are true (note that I said "*if* they are true; I haven't told you how to determine *their* truth value yet), *and* if the rules of inference "preserve truth", then the statement that you prove by means of them—the "theorem"—will also be true. (Briefly, rules of inference—which tell you how to infer a statement from other statements—are truth-preserving if the inferred statement cannot be false as long as the statements from which it is inferred are true.)

Another way we can determine whether a statement is true is *semantically* (that is, in terms of what it *means*). This, by the way, is the only way to determine whether an *axiom* is true, since, by definition, an axiom cannot be inferred from any *other* statements. (If it could be so inferred, then it would be those other statements that would be the real axioms.)

But to determine the truth of a statement semantically is also to use syntax: We semantically determine the truth value of a complex proposition by syntactic manipulation (truth tables) of its atomic constituents. (We can use truth tables to determine that axioms are true.)[8] How do we determine the truth value of an atomic proposition? By seeing if it corresponds to reality. But how do we do that? By comparing the proposition with reality, that is, by seeing if the proposition coheres with reality.

2.5 On Searching for the Truth vs. Finding It

> Thinking is, or ought to be, a coolness and a calmness. . . .
> (Melville, 1851, *Moby-Dick*, Ch. 135, p. 419)

> Thinking does not guarantee that you will not make mistakes.
> But not thinking guarantees that you will. (Lamport, 2015, p. 41)

> Thinking is the hardest work there is, which is the probable reason why so few engage in it. (Ford, 1928, p. 481)

How does one go about *searching* for the truth, for answering questions? As we'll see below, there are basically two complementary methods: (1) thinking hard and (2) empirical investigation. We'll look at the second of these in §2.6. In the present section, we'll focus on thinking hard.

I have read in many places that philosophy is just thinking really hard about things.[9] Such hard thinking requires "rethinking explicitly what we already believe implicitly" (Baars, 1997, p. 187). In other words, it's more than just expressing one's opinion unthinkingly.

Can we *find* the truth? Not necessarily; that is, we may not be *able* to find it. (See McGinn 1989, 1993 for discussion of this possibility.) But I also believe that finding it is not necessary; that is, we may not *have* to find it: **Philosophy is the *search***

[8] For more on the nature of, and relation between, syntax and semantics, see Ch. 22.

[9] Several people have offered this characterization of philosophy. See the quotes at (Popova, 2012) from Marilyn Adams (originally in (Edmonds and Warburton, 2010, p. xiii)) and, especially, David Papineau: "Philosophy is thinking hard about the most difficult problems that there are. And you might think scientists do that too, but there's a certain kind of question whose difficulty can't be resolved by getting more empirical evidence. It requires an untangling of presuppositions: figuring out that our thinking is being driven by ideas we didn't even realize that we had. And that's what philosophy is" (Edmonds and Warburton, 2010, p. xx).

for truth.[10] Albert Einstein said that "the search for truth is more precious than its possession" (Einstein, 1940, p. 492, quoting G.E. Lessing).[11] In a similar vein, the mathematician Carl Friedrich Gauss said, "It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment."[12]

2.5.1 Asking "Why?"

> Questions, questions. That's the trouble with philosophy: you try and fix a problem to make your theory work, and a whole host of others then come along that you have to fix as well. —(Beebee, 2017)

One reason that this search will never end (which is different from saying that it will not succeed) is that you can always ask "Why?"; that is, you can always continue inquiring. This is

> the way philosophy—and philosophers—are[:] Questions beget questions, and those questions beget another whole generation of questions. It's questions all the way down. (Cathcart and Klein, 2007, p. 4)

You can even ask why "Why?" is the most important question (Everett, 2012, p. 38)! "The main concern of philosophy is to question and understand very common ideas that all of us use every day without thinking about them" (Nagel, 1987, p. 5). This is why, perhaps, the questions that children often ask (especially, "Why?") are often deeply philosophical questions.

In fact, as the physicist John Wheeler has pointed out, the more questions you answer, the more questions you can ask: "We live on an island of knowledge surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance."[13] And "Philosophy patrols the ... [shore], trying to understand how we got there and to conceptualize our next move" (Soames, 2016). The US economist and social philosopher Thorstein Veblen said, "The outcome of any serious research can only be to make two questions grow where only one grew before" (Veblen, 1908, p. 396).

Asking "Why?" is part—perhaps the principal part—of philosophy's "general role of critically evaluating beliefs" (Colburn, 2000, p. 6) and "refusing to accept any platitudes or accepted wisdom without examining it" (Donna Dickenson, in (Popova, 2012). Critical thinking in general, and philosophy in particular, "look... for crack[s] in the wall of doctrinaire [beliefs]—some area of surprise, uncertainty, that might then lead to thought" (Acocella, 2009, p. 71).

---

[10] For more on the importance of search over success, see my website on William Perry's theory of intellectual development [http://www.cse.buffalo.edu/∼rapaport/perry-positions.html] and Rapaport 1982. Perry's theory is also discussed briefly in §2.7, below.

[11] "The true value of a man [sic] is not determined by his possession, supposed or real, of Truth, but rather by his sincere exertion to get to the Truth. It is not possession of the Truth, but rather the pursuit of Truth by which he extends his powers ..." (Lessing, 1778).

[12] From "Letter to Bolyai", 1808; http://blog.gaiam.com/quotes/authors/karl-friedrich-gauss/21863

[13] *Scientific American* 267 (1992);
http://www.todayinsci.com/W/Wheeler_John/WheelerJohn-Quotations.htm.

> [It's] not important to get children to read. Children who wanna read are gonna read. Kids who want to learn to read [are] going to learn to read. *[It's] much more important to teach children to QUESTION what they read. Children should be taught to question everything.*
> —George Carlin, http://www.georgecarlin.net/boguslist.html#question

Whenever you have a question, either because you do not understand something or because you are surprised by it or unsure of it, you should begin to think carefully about it. And one of the best ways to do this is to ask "Why?": Why did the author say that? Why does the author believe it? Why should I believe it? (We can call this "looking backwards".) And a related question is this: What are its implications? What else must be true if that were true? And should I believe those implications? (Call this "looking forwards".)

> ... Plato is the philosopher who teaches us that we should never rest assured that our view, no matter how well argued and reasoned, amounts to the final word on any matter. (Goldstein, 2014, p. 396)

> This is why philosophy must be argumentative. It proceeds by way of arguments, and the arguments are argued over. Everything is aired in the bracing dialectic wind stirred by many clashing viewpoints. Only in this way can intuitions that have their source in societal or personal idiosyncrasies be exposed and questioned. (Goldstein, 2014, p. 39)

The arguments are argued over, typically, by challenging their assumptions. It is rare that a philosophical argument will be found to be invalid. The most interesting arguments are valid ones, so that the only concern is over the truth of its premises. An argument that is found to be invalid is usually a source of disappointment—unless the invalidity points to a missing premise or reveals a flaw in the very nature of logic itself (an even rarer, but not unknown, occurrence).

2.5.2 Can There Be Progress in Philosophy?

If the philosophical search for truth is a never-ending process, can we ever make any progress in philosophy? Mathematics and science, for example, are disciplines that not only search for the truth, but seem to find it; they seem to make progress in the sense that we know more mathematics and more science now than we did in the past. We have well-confirmed scientific theories, and we have well-established mathematical proofs of theorems. (The extent to which this may or may not be exactly the right way to look at things will be considered in Chapter 4: "What Is Science?".) But philosophy doesn't seem to be able to empirically confirm its theories or prove any theorems. So, is there any sense of "progress" in philosophy? Or are the problems that philosophers investigate unsolvable?

I think there can be, and is, progress in philosophy. Solutions to problems are never as neat as they seem to be in mathematics. In fact, they're not even that neat in mathematics! This is because solutions to problems are always *conditional*; they are based on certain assumptions. Most mathematical theorems are expressed as conditional statements: *If* certain assumptions are made, or *if* certain conditions are satisfied, *then*

such-and-such is the case. In mathematics, those assumptions include axioms, but axioms can be challenged and modified: Consider the history of non-Euclidean geometry, which began by challenging and modifying the Euclidean axiom known as the Parallel Postulate.[14] So, solutions are really parts of larger theories, which include the assumptions that the solution depends on, as well as other principles that follow from the solution. Progress can be made in philosophy (as in other disciplines), not only by following out the implications of your beliefs ("forward-looking" progress), but also by becoming aware of the assumptions that underlie your beliefs ("backward-looking" progress) (Rapaport, 1982):

> Progress in philosophy consists, at least in part, in constantly bringing to light the covert presumptions that burrow their way deep down into our thinking, too deep down for us to even be aware of them. ... But whatever the source of these presumptions of which we are oblivious, they must be brought to light and subjected to questioning. Such bringing to light is what philosophical progress often consists of .... (Goldstein, 2014, p. 38)

Philosophy is a "watchdog" (Colburn, 2000, p. 6). This zoological metaphor is related to Socrates's view of the philosopher as "gadfly", investigating the foundations of, or reasons for, beliefs and for the way things are, always asking "What *is X*?" and "Why?". Of course, this got him in trouble: His claims to be ignorant were thought (probably correctly) to be somewhat disingenuous. As a result, he was tried, condemned to death, and executed. (For the details, read Plato's *Apology*.)

One moral is that philosophy can be dangerous. As the philosophers Eric Dietrich and Rebecca Newberger Goldstein put it:

> Thinking about the Big Questions is serious, difficult business. I tell my philosophy students: "If you like sweets and easy living and fun times and happiness, drop this course now. Philosophers are the hazmat handlers of the intellectual world. It is we who stare into the abyss, frequently going down into it to great depths. This isn't a job for people who scare easily or even have a tendency to get nervous." (Eric Dietrich, personal communication, 5 October 2006.)

> And what is it, according to Plato, that philosophy is supposed to do? Nothing less than to render violence to our sense of ourselves and our world, our sense of ourselves in the world. (Goldstein, 2014, p. 40)

It is violent to have one's assumptions challenged:

> [P]hilosophy is difficult because the questions are hard, and the answers are not obvious. We can only arrive at satisfactory answers by thinking as rigorously as we can with the strongest logical and analytical tools at our disposal.
>
> ... I want ... [my students] to care more about things like truth, clear and rigorous thinking, and distinguishing the truly valuable from the specious.
>
> The way to accomplish these goals is not by indoctrination. Indoctrination teaches you what to think; education teaches you how to think. Further, the only

[14] One version of the Parallel Postulate is this: For any line L, and for any point P not on L, there is only one line L' such that P is on L' and L' is parallel to L. For some of the history of non-Euclidean geometries, see http://mathworld.wolfram.com/ParallelPostulate.html or http://en.wikipedia.org/wiki/Parallel_postulate.

> way to teach people how to think is to challenge them with new and often unsettling ideas and arguments.
>
> ... Some people fear that raising such questions and prompting students to think about them is a dangerous thing. They are right. As Socrates noted, once you start asking questions and arguing out the answers, you must follow the argument wherever it leads, and it might lead to answers that disturb people or contradict their ideology. (Parsons, 2015)

So, the whole point of Western philosophy since Socrates has been to get people to think about their beliefs, to question and challenge them. It is not (necessarily) to come up with answers to difficult questions.[15]

Further Reading:
For more on whether there can be progress in philosophy, see (Rapaport, 1982), (Rapaport, 1984a), (Rescher, 1985), (Moody, 1986), (Chalmers, 2015), and (Frances, 2017), as well as the answers to "Have philosophers ever produced anything in the way that scientists have?" and "How is 'philosophical progress' made, assuming it is made at all?", at http://www.askphilosophers.org/question/2249 and http://www.askphilosophers.org/question/4523, respectively.

2.5.3 Skepticism

> Sceptics[16] do not always really intend to prove to us that we cannot know any of the things we naïvely think we know; sometimes they merely wish to demonstrate to us that we are too naïve about how we know them. ... [S]ceptics have an uncanny eye for fundamental principles .... (Katz, 1978, pp. 191–192)

If you can always ask "Why?"—if you can challenge any claims—then you can be skeptical about everything. Does philosophy lead to skepticism?[17]

Skepticism is often denigrated as being irrational. But there are advantages to always asking questions and being skeptical: "A skeptical approach to life leads to advances in all areas of the human condition; while a willingness to accept that which does not fit into the laws of our world represents a departure from the search for knowledge" (Dunning, 2007). Being skeptical doesn't necessarily mean refraining from having any opinions or beliefs. But it does mean being willing to question anything and everything that you read or hear (or think!). Here is another way of putting this: In philosophy, the jury is always out!—see (Polger, 2011, p. 21). But, as we saw above, this does not mean that there can be no progress in philosophy.

Why would you want to question anything and everything? So that you can find reasons for (or against) believing what you read or hear (or think)! And why is it important to have these reasons? For one thing, it can make you feel more confident about your beliefs and the beliefs of others. For another, it can help you try to convince

[15] Very similar comments have been made about science: "The best science often depends on asking the most basic questions, which are often the hardest to ask because they risk exposing fundamental limitations in our knowledge" (Mithen, 2016, p. 42).

[16] That's the British spelling.

[17] See http://www.askphilosophers.org/questions/5572

others about your beliefs—not necessarily to convince them that they should believe what you believe, but to help them understand why you believe what you do.

> I do not pretend that I can refute these two views; but I can challenge them .... —Popper (1978, §4, p. 148)

This is the heart of philosophy: not (necessarily) coming up with answers, but challenging assumptions and forcing you to think about alternatives. My father's favorite admonition was: Never make assumptions. That is, never assume that something is the case or that someone is going to do something; rather, try to find out if it is the case, or ask the person. In other words, challenge all assumptions. Philosophers, as James Baldwin (1962) said about artists, "cannot and must not take anything for granted, but must drive to the heart of every answer and expose the question the answer hides."

This is one way that progress can be made in philosophy: It may be backward-looking progress, because, instead of looking "forwards" to implications of your assumptions, you look "backwards" to see where those assumptions might have come from. (Besides these two directions of progress, there can be a third, which is orthogonal to these two: Considering other issues that might not underlie ("backward") or follow from ("forward") the one that you are considering, but that are "inspired" or "suggested" by it ("sideways" progress?).)

2.6 What Is "Rational"?

> *Active, persistent, and careful consideration of any belief or supposed form of knowledge in the light of the grounds that support it, and the further conclusions to which it tends*, constitutes reflective thought. (Dewey, 1910, p. 6)

Mere statements (that is, opinions) *by themselves* are **not** rational. Rather, **arguments**—reasoned or *supported* statements—are capable of being rational. As the American philosopher John Dewey suggested, it's not enough to merely think something; you must also consider *reasons for* believing it (looking "backwards"), and you must also consider the *consequences of* believing it (looking "forwards"). That is, being rational requires *logic*.

But there are lots of different (kinds of) logics, so there are lots of different kinds of rationality. And there is another kind of rationality, which depends on logics of various kinds, but goes beyond them in at least one way: empirical, or scientific, rationality. Let's look at these two kinds of rationality.

2.6.1 Kinds of Rationality

> "Philosophy: the ungainly attempt to tackle questions that come naturally to children, using methods that come naturally to lawyers" (Hills, 2007).

There are (at least) two *basic* kinds of logical rationality: deductive-logical (or absolutely certain) rationality and scientific (or probabilistic) rationality. There is also, I think, a third kind, which I'll call "psychological" or maybe "economic", and which is at the heart of knowledge representation and reasoning in AI.

2.6.1.1 Deductive Logical Rationality

"Deductive" logic is the main kind of *logical* rationality. Premises $P_1, \ldots, P_n$ deductively support (or "yield", or "entail", or "imply") a conclusion C iff C *must* be true *if* all of the P_i are true; that is, C is true iff (1) the P_i are all true, and (2) the rules of inference that lead from the P_i to C "preserve" truth.

I will use the symbol '$\vdash_D$' to represent this truth-preserving relation between premises and a conclusion that is deductively supported by them. For example, the rule of "modus ponens":

$$P,\ (P \rightarrow C)\ \vdash_D\ C$$

means that the two premises P and $(P \rightarrow C)$ (read this as: "if P, then C") deductively support (that is, they deductively imply) C. For example, let P = "Today is Wednesday." and let C = "We are studying philosophy." So the inference becomes: "Today is Wednesday. If today is Wednesday, then we are studying philosophy. Therefore (deductively), we are studying philosophy."

There are three somewhat surprising things that must be pointed out:

1. **Any or all of the P_i can be false!** A deductive argument is said to be "valid" iff it is impossible for all of the premises to be true but the conclusion false. This can be said in a slightly different way: A deductive argument is valid iff, whenever all of its premises are true, its conclusion cannot be false (or: its conclusion must also be true). But note that "whenever" allows for the possibility that one or more premises are false.

   A deductive argument is said to be "sound" iff it is valid *and* all of its premises are, in fact, true. So, any or all of the premises of a deductively valid argument can be false, as long as—*if* they *were* true, then the conclusion *would also have to be* true.

2. ***C* can be false!** *If* one of the P_i *is* false, even truth-preserving rules can lead to a false C. The conclusion C only has to be true *relative to* the premises (that is, true relative to its context). In other words, C *would be* true *if* all of the P_i *were* true.

3. **The P_i can be *irrelevant* to *C*!** But that's not a good idea, because it wouldn't be a *convincing* argument.

   The classic example of this is that anything follows deductively from a contradiction: From '$2+2=4$' and '$2+2\neq 4$', one can deductively infer that the philosopher Bertrand Russell (a noted atheist) is the Pope.[18]

> **Further Reading:**
> "Relevance" logics are one way of dealing with this problem; see (Anderson and Belnap, 1975) and (Anderson et al., 1992). For applications of relevance logic to AI, see (Shapiro and Wand, 1976) and (Martins and Shapiro, 1988).

[18] Let P and $\neg P$ be the two premises, and let C be the conclusion. From P, we can deductively infer $(P \vee C)$, by the truth-preserving rule of Addition (a form of $\vee$-introduction). Then, from $(P \vee C)$ and $\neg P$, we can deductively infer C, by the truth-preserving rule of Disjunctive Syllogism (a form of $\vee$-elimination).

2.6.1.2 Inductive Logical Rationality

"Inductive" logic is one of the two main kinds of *scientific* rationality. The other is "abductive" logic (to be discussed in the next section). Deductive rationality, which is more characteristic of mathematics than of the experimental sciences, is, however, certainly part of science.

In inductive logic, $P_1, \ \ldots, \ P_n \ \vdash_I \ C$ iff C is *probably* true *if* all of the P_i *are* true. For example, suppose that you have an urn containing over a million ping-pong balls, and suppose that you remove one of them at random and observe that it is red. What do you think the chances are that the next ball will also be red? They are probably not very high. But suppose that the second ball that you examine is also red. And the third. ... And the 999,999th. Now how likely do you think it is that the next ball will also be red? The chances are probably very high, so:

$$\text{Red}(\text{ball}_1), \ldots, \text{Red}(\text{ball}_{999,999}) \vdash_I \text{Red}(\text{ball}_{1,000,000}).$$

Unlike deductive inferences, however, inductive ones do not *guarantee* the truth of their conclusion. Although it is not likely, it is quite possible that the millionth ping-pong ball will be, say, the only blue one in the urn.

2.6.1.3 Abductive Logical Rationality

> Adding a new hypothesis or axiom to a theory for the purpose of explaining already known facts is a process known as "abduction". (Sloman, 2010, slide 56)

"Abductive" logic, sometimes also known as "inference to the best explanation" (Harman, 1965), (Lipton, 2004), (Campos, 2011),[19] is also scientific: From observation O made at time t_1, and from a theory T that deductively or inductively entails O, one can *abductively* infer that T *must have been* the case at earlier time t_0. In other words, T is an *explanation* of why you have observed O. Of course, it is not necessarily a good, much less the best, explanation, but the more observations that T explains, the better a theory it is. (But what is a "theory"? We'll delve into that in §4.7. For now, you can think of a theory as just a set of statements that describe, explain, or predict some phenomenon.)

This kind of argument is deductively *invalid*:

(A) $O, \ (T \rightarrow O) \ \nvdash_D \ T$

Argument (**A**) is called the fallacy of affirming the consequent.[20] (We will come back to this in §4.9.1.1.)

In another form of abduction, from observation O_1 made at time t_1, and from observation O_2 made at a later time t_2, one can abductively infer that O_1 might have caused or logically entailed O_2.

[19] For the origin of the term in the writings of the American philosopher Charles Sanders Peirce (who pronounced his name like the word 'purse'), see
http://www.helsinki.fi/science/commens/terms/abduction.html

[20] O is the "consequent" of the conditional statement $(T \rightarrow O)$ (if T, then O). "Affirming" O as a premise thus "affirms the consequent". But if O is true and T is false, then both premises are true, yet the conclusion (T) is not.

This, too, is deductively invalid: Just because two observations are correlated does not imply that the first causes the second, because the second might have caused the first, or both might have been caused by a third thing.

Like inductive inferences, abductive ones are not deductively valid and do not guarantee the truth of their conclusion. But abductive inferences are at the heart of the scientific method for developing and confirming theories. And they are used in the law, where they are known as "circumstantial evidence".

2.6.1.4 Non-Monotonic Logical Rationality

"Non-monotonic" reasoning is more "psychologically real" than any of the others. It also underlies what the economist and AI researcher Herbert Simon called "satisficing" (or being *satisfi*ed with something that suf*fices* to answer your question rather than having an optimal answer), for which he won the Nobel Prize in Economics. (For a history of the notion, see Brown 2004. We'll return to this topic in §§3.15.2.3, §5.7, and 11.4.5.2.)

In *monotonic* logics (such as deductive logics), once you have proven that a conclusion *C* follows from a premise *P*, then you can be assured that it will always so follow. But in *non*-monotonic logic, you might infer conclusion *C* from premise *P* at time t_0, but, at later time t_1, you might learn that it is not the case that *C*. In that case, you must revise your beliefs. For example, you might believe that birds fly and that Tweety is a bird, from which you might conclude that Tweety flies. But if you then learn that Tweety is a penguin, you will need to revise your beliefs.

Further Reading:
A great deal of work on non-monotonic logics has been done by researchers in the branch of AI called "knowledge representation"; see the bibliography at
http://www.cse.buffalo.edu/∼rapaport/663/F08/nonmono.html.

2.6.1.5 Computational Rationality

In addition to the two kinds of "rationality" discussed in §2.6.1 (logical rationality and scientific rationality), the astronomer Kevin Heng argues that,

> a third, modern way of testing and establishing scientific truth—in addition to theory and experiment—is via simulations, the use of (often large) computers to mimic nature. It is a synthetic universe in a computer. ... If all of the relevant physical laws are faithfully captured [in the computer program] then one ends up with an emulation—a perfect, *The Matrix*-like replication of the physical world in virtual reality. (Heng, 2014, p. 174)

One consideration that this raises is whether this is really a third way, or just a version of logical rationality, perhaps extended to include computation as a kind of "logic". (We'll discuss computer programs and computational simulations in Chapter 15: "Are Computer Programs Theories?")

However, all of the above kinds of rationality seem to have one thing in common: They are all "declarative". That is, they are all concerned with statements (or propositions) that are true or false. But the philosopher Gilbert Ryle has argued that there is another kind of rationality, one that is "procedural" in nature: It has been summarized as "knowing *how*" (to do something), rather than "knowing *that*" (something is the case) (Ryle, 1945, especially p. 9). We will explore this kind of rationality in more detail in §§3.6.1 and 3.14.4.

2.6.2 Science and Philosophy

If philosophy is a search for truth by rational means, what is the difference between philosophy and science? After all, science is also a search for truth by rational means! Is philosophy worth doing? Or can science answer all of our questions?

2.6.2.1 Is Science Philosophy?

Is the experimental or empirical methodology of science "rational"? It is *not* (entirely) deductive. But it yields highly likely conclusions, and is often the best we can get.

I would say that science *is* philosophy, as long as experiments and empirical methods are considered to be "rational" and yield truth. Physics and psychology, in fact, used to be branches of philosophy: Isaac Newton's *Principia*—the book that founded modern physics—was subtitled "Mathematical Principles of *Natural Philosophy*" (italics added), not "Mathematical Principles of *Physics*", and psychology split off from philosophy only at the turn of the 20th century. The philosophers Aristotle and Kant wrote physics books. The physicists Einstein and Mach wrote philosophy. And the "philosophy naturalized" movement in contemporary philosophy (championed by the philosopher Willard Van Orman Quine) sees philosophy as being on a continuum with science. (See §2.6.2.2; we'll come back to this in §2.8.)

But, if experiments don't count as being rational and only logic counts, then science is *not* philosophy. And science is also not philosophy, if philosophy is considered to be the search for *universal* or *necessary* truths, that is, things that would be true no matter what results science came up with or what fundamental assumptions we made.

There might be conflicting world views (e.g., creationism vs. evolution, perhaps). Therefore, the best theory is one that is **consistent**, that is **as complete as possible** (that is, that explains as much as possible), and that is **best-supported** by **good** evidence.

You can't *refute* a theory. You can only point out *problems* with it and then offer a *better* theory. Suppose that you infer a prediction P from a theory T together with a hypothesis H, and then suppose that P doesn't come true (your experiment fails; that is, the experimental evidence is that P is not the case). Then, logically, *either* H is not the case *or* T is not the case (*or* both!). And, since T is probably a complex conjunction of claims $A_1, \ldots, A_n$, then, if T is not the case, then at least one of the A_i is not the case. In other words, you need not *give up* a theory; you only need to *revise* it. That is, if P has been falsified, then you only need to give up one of the A_i or H, not necessarily the whole theory T. However, sometimes you *should* give up an entire theory. This is what happens in the case of "scientific revolutions", such as (most famously) when Copernicus's theory that the Earth revolves around the Sun (and

not vice versa) replaced the Ptolemaic theory, small revisions to which were making it overly complex without significantly improving it. (I'll have a bit more to say about this in Ch. 4.)

2.6.2.2 Is Philosophy a Science?

Could philosophy be more scientific (that is, experimental) than it is? Should it be? The philosopher Colin McGinn takes philosophy to be a science ("a systematically organized body of knowledge" (McGinn, 2012a)), in particular, what he dubs 'ontical science': "the subject consists of the search for the essences of things by means of a priori methods" (McGinn, 2012b). In a later paper, he argues that philosophy is a science, just like physics or mathematics. More precisely, he says that it is the logical science of concepts (McGinn, 2015b, pp. 87–88).

There is a relatively recent movement (with some older antecedents) to have philosophers do scientific (mostly psychological) experiments in order to find out, among other things, what "ordinary" people (for example, people who are not professional philosophers) believe about certain philosophical topics. My feeling is that this is not really philosophy, but rather an (interesting and valuable) branch of cognitive science.

Further Reading:
For more information on this movement, sometimes called 'X-Phi', see (Nahmias et al., 2006), (Appiah, 2007), (Appiah, 2008), (Knobe, 2009), (Beebe, 2011), (Nichols, 2011), and (Roberts and Knobe, 2016). For an argument *against* experimental philosophy, see (Deutsch, 2009).

But there is another way that philosophy can be part of a scientific worldview. This can be done by philosophy being continuous with science, that is, by being aware of, and making philosophical use of, scientific results. Rather than being a passive, "armchair" discipline that merely analyzes what others say and do, philosophy can—and probably should—be a more active discipline, even helping to contribute to science (and other disciplines that it thinks about).

Further Reading:
For a useful discussion of this, which is sometimes called "naturalistic philosophy", see (Thagard, 2012). See also (Williamson, 2007) on why there's nothing wrong with "armchair" philosophy.

Philosophers can also be more "practical" in the public sphere: "The philosophers have only *interpreted* the world in various ways; the point is to *change* it" (Marx, 1845). But an opposing point of view considers that "philosophers ... are ordained as priests to keep alive the sacred fires in the altar of impartial truth" ("Philonous", 1919, p. 19)! (For more on this, see §5.7.)

Further Reading:
For a debate on science vs. philosophy, read (Linker, 2014), (Powell, 2014), and (Pigliucci, 2014), in that order.

For a discussion of whether philosophy or science is "harder", see (Papineau, 2017).

2.6.3 Is It Always Rational to Be Rational?

Is there anything to be said in favor of *not* being rational?

Suppose that you are having trouble deciding between two apparently equal choices. This is similar to a problem from mediaeval philosophy known as "Buridan's Ass" (see (Zupko, 2011)): According to one version, an ass (that is, a donkey) was placed equidistant between two equally tempting bales of hay but died of starvation because it couldn't decide between the two of them. My favorite way out of such a quandary is to imagine tossing a coin and seeing how you feel if it lands heads up: If you would rather that it had landed *tails* up, then you know what you would have preferred, even if you had "rationally" decided that both choices were perfectly equally balanced.

Further Reading:
Look up Andrew N. Carpenter's response to the question "To what extent do philosophers/does philosophy allow for instinct, or gut feelings?" on the *AskPhilosophers* website (http://www.askphilosophers.org/question/2992). An interesting discussion of the role—and limits—of rationality in AI research is (Russell, 1995).

2.7 What Is the Import of "Personal Search"?

> ... I'm not trying to change anyone's mind on this question. I gave that up long ago. I'm simply trying to say what I think is true.
> (Strawson, 2012, p. 146)
>
> And among the philosophers, there are too many Platos to enumerate. All that I can do is try to give you mine. (Goldstein, 2014, p. 396)
>
> [M]y purpose is to put my own intellectual home in order.... (Putnam, 2015)
>
> "The philosophy of every thinker is the more or less unconscious autobiography of its author," Nietzsche observed.... (Martin, 2015)

My major professor, Hector-Neri Castañeda, used to say that philosophy should be done "in the first person, for the first person" (Rapaport, 2005a). So, philosophy is whatever *I* am interested in, as long as I study it in a rational manner and aim at truth (or, at least, aim at the best theory).

There is another way in which philosophy must be a personal search for truth. As one introductory book puts it, "the object here is not to give answers ... but to introduce you to the problems in a very preliminary way *so that you can worry about them yourself*" (Nagel, 1987, pp. 6–7, my italics). The point is not to hope that someone else will tell you the answers to your questions. That would be nice, of course; but

why should you believe them? The point, rather, is for you to figure out answers for yourself.

It may be objected that your first-person view on some topic, no matter how well thought out, is, after all, just your view. "Such an analysis can be of only parochial interest" (Strevens, 2019) or might be seriously misleading (Dennett, 2017, pp. 364–370). Another philosopher, Hilary Kornblith, agrees:

> I believe that the first-person perspective is just one perspective among many, and it is wholly undeserving of the special place which these philosophers would give it. More than this, this perspective is one which fundamentally distorts our view of crucial features of our epistemic situation. Far from lauding the first-person perspective, we should seek to overcome its defects. (Kornblith, 2013, p. 126)

But there is another important feature of philosophy, as I mentioned in §1.3: It is a conversation. And if you want to contribute to that conversation, you will have to take others' views into account, and you will have to allow others to make you think harder about your own views.

The desire for an "Authority" to answer all questions for you has been called the "Dualistic" stance towards knowledge.[21] But the Dualist soon realizes that not all questions have answers that everyone agrees with, and some questions don't seem to have answers at all (at least, not yet).

Rather than stagnating (my term, not Perry's!) in a middle stance of "Multiplism" (because not all questions have answers, multiple opinions—proposed answers—are all equally good), a further stance is that of "Contextual Relativism": All proposed answers or opinions can (should!) be considered—and evaluated!—*relative to* and *in the context of* assumptions, reasons, or evidence that can support them.

Eventually, you "Commit" to one of these answers, and you become responsible for defending your commitment against "Challenges". But that is (just) more thinking and analysis—more philosophizing. Moreover, the commitment that you make is a personal one (one that *you* are responsible for). As the computer scientist Richard W. Hamming warned, "In science and mathematics we do not appeal to authority, but rather *you are responsible for what you believe*" (Hamming, 1998, p. 650).

It is in this way that philosophy is done "in the first person, for the first person", as Castañeda said.

2.8 What Is the Import of "In Any Field"?

> One of the things about philosophy is that you don't have to give up on any other field. Whatever field there is, there's a corresponding field of philosophy. Philosophy of language, philosophy of politics, philosophy of math. All the things I wanted to know about I could still study within a philosophical framework.
> —Rebecca Newberger Goldstein, cited in Reese 2014b

> [He] is a *philosopher*, so he's interested in everything. . . .
> —David Chalmers (describing the philosopher Andy Clark), as cited in Cane 2014.

[21] (Perry, 1970), (Perry, 1981), http://www.cse.buffalo.edu/∼rapaport/perry.positions.html

Figure 2.3: ©Hilary B. Price, http://rhymeswithorange.com/comics/august-31-2007/

> It is not really possible to regret being a philosopher if you have a theoretical (rather than practical or experiential) orientation to the world, because there are no boundaries to the theoretical scope of philosophy. For all X, there is a philosophy of X, which involves the theoretical investigation into the nature of X. There is philosophy of mind, philosophy of literature, of sport, of race, of ethics, of mathematics, of science in general, of specific sciences such as physics, chemistry and biology; there is logic and ethics and aesthetics and philosophy of history and history of philosophy. I can read Plato and Aristotle and Galileo and Newton and Leibniz and Darwin and Einstein and John Bell and just be doing my job. I could get fed up with all that and read Eco and Foucault and Aristophanes and Shakespeare for a change and still do perfectly good philosophy. —Tim Maudlin (2018), https://blogs.scientificamerican.com/cross-check/philosophy-has-made-plenty-of-progress/

Philosophy also studies things that are *not* studied by any *single* discipline; these are sometimes called "the Big Questions": What is truth? What is beauty? What is good (or just, or moral, or right)? What is the meaning of life? What is the nature of mind? (For a humorous take on this, see Fig. 2.3.) Or, as the philosopher Jim Holt put it: "Broadly speaking, philosophy has three concerns: how the world hangs together, how our beliefs can be justified, and how to live" (Holt, 2009). The first of these is metaphysics, the second is epistemology, and the third is ethics. (Similar remarks have been made by (Flanagan, 2012, p. B4), (Schwitzgebel, 2012), and (Weatherson, 2012).)

But the main branches of philosophy go beyond these "big three":

1. **Metaphysics** tries to "understand the nature of reality in the broadest sense: what kinds of things and facts ultimately constitute everything there is" (Nagel, 2016, p. 77). It tries to answer the question "What is there?"[22] (and also the question "Why is there anything at all?"). Some of the things that there might be include: physical objects, properties, relations, individuals, time, God, actions, events, minds, bodies, etc.

   There are major philosophical issues surrounding each of these. Here are just a few examples:

[22] For a computational approach to this metaphysical question, see http://www.cse.buffalo.edu/∼rapaport/663/F06/course-summary.html

- Which physical objects "really" exist? Do rocks and people exist? Or are they "merely" collections of molecules? But molecules are constituted by atoms; and atoms by electrons, protons, and neutrons. And, according to the "standard model", the only really elementary particles are quarks, leptons (which include electrons), and gauge bosons; so maybe those are the only "really existing" physical objects.[23] Here is a computationally relevant version of this kind of question: Do computer programs that deal with, say, student records model students? Or are they just dealing with 0s and 1s? And, on perhaps a more fanciful level, could a computer program model students so well that the "virtual" students in the program believe that they are real?[24]
- Do "socially constructed" things like money, universities, governments, etc., really exist (in the same way that people or rocks do)? (This problem is discussed in (Searle, 1995).)
- Do properties really exist? Or are they just collections of similar (physical) objects; in other words, is there a property—"Redness"—over and above the class of individual red things? Sometimes, this is expressed as the problem of whether properties are "intensional" (like Redness) or "extensional" (like the set of individual red things). (See §3.4 for more about this distinction.)
- Are there any important differences between "accidental" properties (such as my property of being a professor of computer science rather than my being a professor of philosophy) and "essential" properties (such as my property of being a human rather than being a laurel tree)?[25]
- Do "non-existents" (such as Santa Claus, unicorns, Sherlock Holmes, etc.) exist? We can and do think and talk about them. Therefore, whether or not they "exist" in any sense, they do need to be dealt with.

  Further Reading:
  On non-existence, see (Quine, 1948). For a survey of the AI approach to non-existence, see (Hirst, 1991). And for some papers on a fully intensional AI approach to these issues, see (Maida and Shapiro, 1982), (Rapaport, 1986a), (Wiebe and Rapaport, 1986), (Shapiro and Rapaport, 1987), (Shapiro and Rapaport, 1991), and (Rapaport et al., 1997).

- **Ontology** is the branch of metaphysics that is concerned with the objects and the kinds of objects that exists according to one's metaphysical (or even physical) theory, their properties, and their relations to each other (such as whether some of them are "sub-kinds" of others, inheriting their properties and relations from their "super-kinds"). For example, the modern ontology of physics recognizes the existence only of fermions (quarks, leptons, etc.)

[23] For an interesting take on this, see (Unger, 1979b), (Unger, 1979a).

[24] If this sounds like the film *The Matrix*, see Ch. 20: "Computer Ethics II: Should We Build Artificial Intelligences?".

[25] http://www.theoi.com/Nymphe/NympheDaphne.html

and bosons (photons, gluons, etc.); everything else is composed of things (like atoms) that are, in turn, composed of these.[26] Ontology is studied both by philosophers and by computer scientists. In software engineering, "object-oriented" programming languages are more focused on the kinds of objects that a program must deal with than with the instructions that describe their behavior. In AI, ontology is a branch of knowledge representation that tries to categorize the objects that a knowledge-representation theory is concerned with.

Further Reading:
For more information on ontology, see
http://www.cse.buffalo.edu/∼rapaport/563S05/ontology.html.
For the AI version of ontology, see http://aitopics.org/topic/ontologies and http://ontology.buffalo.edu/.

And so on. As William James said:

> Metaphysics means only **an unusually obstinate attempt to think clearly and consistently.** . . . A geologist's purposes fall short of understanding Time itself. A mechanist need not know how action and reaction are possible at all. A psychologist has enough to do without asking how both he and the mind which he studies are able to take cognizance of the same outer world. But it is obvious that problems irrelevant from one standpoint may be essential for another. And as soon as one's purpose is the attainment of the maximum of possible insight into the world as a whole, the metaphysical puzzles become the most urgent ones of all. (James, 1892, "Epilogue: Psychology and Philosophy", p. 427; my boldface)

2. **Epistemology** is the study of knowledge and belief: It is "concerned with whether we humans have the capacity to discover the answers to those questions [of metaphysics], and if not, what limits to our knowledge are imposed by our finite human faculties" (Nagel, 2016, p. 77):

   > Epistemology is concerned with the question of how, since we live, so to speak, inside our heads, we acquire knowledge of what there is outside our heads. (Simon, 1996a, p. 162)

   How do we know what there is? How do we know that there is anything? What is knowledge? Is it justified, true belief, as Plato thought, or are there counterexamples to that analysis (such as those of (Gettier, 1963))? Are there other kinds of knowledge, such as knowing how to do something (see §3.14.4), or knowing someone (either by acquaintance or by description)? What is belief, and how does it relate to knowledge? Can a computer (or a robot) be said to have beliefs or knowledge? In fact, the branch of AI called "knowledge representation" applies philosophical results in epistemology to issues in AI and computer science in general, and it has contributed many results to philosophy as well.

[26] https://en.wikipedia.org/wiki/Elementary_particle

Further Reading:
On knowledge representation, see (Buchanan, 2006), (Shoham, 2016), and the bibliography at http://www.cse.buffalo.edu/∼rapaport/663/F08/krresources.html.

3. **Ethics** tries to answer "What is good?", "What ought we to do?". We'll look at some ethical issues arising from computer science in Chapters 18 and 20.

4. Ethics is closely related to both **social and political philosophy** and to the **philosophy of law**, which try to answer "What are societies?", "What are the relations between societies and the individuals who constitute them?", "What is the nature of law?".

5. **Aesthetics** (or the **philosophy of art**) tries to answer "What is beauty?", "What is art?". (On whether computer programs, like mathematical theorems or proofs, can be "beautiful", see §3.14.2.)

6. **Logic** is the study of good reasoning: What is truth? What is rationality? Which arguments are good ones? Can logic be computationally automated? (Recall our discussion in §2.6.)

7. Philosophy is one of the few disciplines (history is another) in which the history of itself is one of its branches: The **history of philosophy** looks at what famous philosophers of the past believed, and tries to reinterpret their views in the light of contemporary thinking.

8. And of central interest for the philosophy of computer science, there are numerous "philosophies of":

   - **Philosophy of language** tries to answer "What is language?", "What is meaning?". It has large overlaps with linguistics and with cognitive science (including AI and computational linguistics).
   - **Philosophy of mathematics** tries to answer "What is mathematics?", "Is math about numbers, numerals, sets, structures?", "What are numbers?", "Why is mathematics so applicable to the real world?".

     Further Reading:
     On the philosophy of mathematics, see (Benacerraf and Putnam, 1984), (Pincock, 2011), and (Horsten, 2015).

   - **Philosophy of mind** tries to answer "What is 'the' mind?", "How is the mind related to the brain?" (this is known as the "mind-body" problem), Are minds and bodies two different kinds of substances? (This is known as "dualism", initially made famous by the 17th-century French philosopher Descartes.) Or are they two different aspects of some one, underlying substance? (This is a position made famous by the 17th-century Dutch philosopher Baruch Spinoza.) Or are there no minds at all, but only brains? (This is known as "materialism" or "physicalism"; it is the position of most

contemporary philosophers and scientists.) Or are there no independently existing physical objects, but only ideas in our minds? (This is known as "idealism", made famous by the 18th-century Irish philosopher George Berkeley.) It also investigates whether computers can think (or be said to think), and it has close ties with cognitive science and AI:

> [I]f there remain any philosophers who are not familiar with some of the main developments in artificial intelligence, it will be fair to accuse them of professional incompetence, and that to teach courses in philosophy of mind, epistemology, aesthetics, philosophy of science, philosophy of language, ethics, metaphysics, and other main areas of philosophy, without discussing the relevant aspects of artificial intelligence will be as irresponsible as giving a degree course in physics which includes no quantum theory. (Sloman, 1978, §1.2, p. 3)

(In §12.4.6, we'll say more about the mind-body problem and its relation to the software-hardware distinction.)

- **Philosophy of science** tries to answer "What is science?", "What is a *scientific* theory?", "What is a *scientific* explanation?". The philosophy of computer science is part of the philosophy of science. The philosopher Daniel C. Dennett has written that there was a "reform that turned philosophy of science from an armchair fantasy field into a serious partnership with actual science. There came a time when philosophers of science decided that they really had to know a lot of current science from the inside" (Dennett, 2012, p. 12). Although you do not need to know a lot about computer science (or philosophy, for that matter) to learn something from the present book, clearly the more you know about each topic, the more you will be able both to understand what others are saying and to contribute to the conversation. (We will look more closely at the philosophy of science in Ch. 4.)
- In general, **for any X, there can be a philosophy of X**: the philosophical investigation of the fundamental assumptions, methods, and goals of *X* (including metaphysical, epistemological, and ethical issues), where *X* could be: biology, education, history, law, physics, psychology, religion, etc., including, of course, AI and computer science. The possibility of a philosophy of *X* for any *X* is the main reason why philosophy is the rational search for truth in *any* field. "Philosophy is 99 per cent about critical reflection on anything you care to be interested in" (Richard Bradley, in (Popova, 2012)). Philosophy is a "meta-discipline", especially the philosophy of *X*: In *X*, you think about *X* (in mathematics, you think about mathematics); but in the philosophy of *X*, you think about *thinking about X*. Even those subjects that might be purely philosophical (metaphysics, epistemology, and ethics) have strong links to disciplines like physics, psychology, and political science, among others.

  X, by the way, could also be ... philosophy! The philosophy of philosophy, also known as "metaphilosophy", is exemplfied by this very chapter, which is an investigation into what philosophy is and how it can be done.

> Some people might think that the philosophy of philosophy is the height of "gazing at your navel", but it's really what's involved when you think about thinking, and, after all, isn't AI just *computational* thinking about thinking?

Philosophy, besides being interested in any specific topic, also has an overarching or topic-spanning function: It asks questions that don't fall under the aegis of specific topics and that span multiple topics: The philosopher Wilfrid Sellars said, "The aim of philosophy, abstractly formulated, is to understand how things in the broadest possible sense of the term hang together in the broadest possible sense of the term" (Sellars, 1963, p. 1). He went on to say that "the reflective knowing one's way around in the scheme of things ... is the aim of philosophy ... . What is characteristic of philosophy is not a special subject-matter, but the aim of knowing one's way around with respect to the subject-matters of all the special disciplines. ... It is therefore the 'eye on the whole' which distinguishes the philosophical enterprise" (Sellars, 1963, pp. 2–3).

So, for instance, while it is primarily (but not only) mathematicians who are interested in mathematics *per se* and primarily (but not only) scientists who are interested in science *per se*, it is primarily (but not only) philosophers who are interested in how and why mathematics is so useful for science (see (Smith, 2010)).

Are there any topics that philosophy *doesn't* touch on? I'm sure that there are some topics that philosophy *hasn't* touched on. But I'm equally sure that there are no topics that philosophy *couldn't* touch on.[27]

2.9 Philosophy and Computer Science

Philosophy and computer science overlap not only in some topics of common interest (logic, philosophy of mind, philosophy of language, etc.), but also in methodology: the ability to find counterexamples; refining problems into smaller, more manageable ones; seeing implications; methods of formal logic; and so on.

Further Reading:
For further discussion of the value of philosophy for computer science (and vice versa!), see (Arner and Slein, 1984, especially pp. 76–77).

So, let's now begin our philosophical investigation into computer science.

[27] See http://www.askphilosophers.org/question/3001

2.10 Further Sources of Information

1. Standard Reference Works (in print and online):
   (a) Craig, Edward (ed.) (1998), *Routledge Encyclopedia of Philosophy* (London: Routledge).
   (b) Edwards, Paul (ed.) (1967), *Encyclopedia of Philosophy* (New York: Macmillan).
   (c) Fieser, James; & Dowden, Bradley (eds.), *Internet Encyclopedia of Philosophy*, http://www.iep.utm.edu/
   (d) Zalta, Edward N. (ed.), *Stanford Encyclopedia of Philosophy*, http://plato.stanford.edu/
2. What Is Philosophy?
   (a) Books and Essays:
   - Floridi, Luciano (2013), "What Is a Philosophical Question?", *Metaphilosophy* 44(3) (April): 195–221.
     – "Philosophical questions are questions not answerable empirically or mathematically, with observations or calculations. They are *open questions*, that is, questions that remain in principle open to informed, rational, and honest disagreement, even after all the relevant observations and calculations have become available and the answers have been formulated" (pp. 200–201, my italics).
   - McGinn, Colin (2003), "Finding Philosophy", *Prospect*, Issue 92 (November), http://www.prospectmagazine.co.uk/magazine/findingphilosophy/
     – A brief autobiography of how a well-known contemporary philosopher got into the field.
   - **Nagel, Thomas (1987), *What Does It All Mean? A Very Short Introduction to Philosophy* (Oxford: Oxford University Press).**
     – My favorite introduction to philosophy.
   - **Russell, Bertrand (1912), *The Problems of Philosophy***, various editions and publishers.
     – One of my favorite introductions to philosophy.
   - Russell, Bertrand (1946), "Philosophy for Laymen", in his *Unpopular Essays* (London: George Allen & Unwin); http://www.users.drew.edu/∼jlenz/br-lay-philosophy.html
     – Not so much about what philosophy is, but about why studying it is important for everyone, not just professional philosophers. (Warning: The online version has many typographical errors!)
   - Tallis, Raymond (2013), "Philosophy Isn't Dead Yet", *The Guardian* (26 May), http://www.guardian.co.uk/commentisfree/2013/may/27/physics-philosophy-quantum-relativity-einstein
   - Green, Daryn (2014), "A Philosophical Round", *Philosophy Now* 105 (November/December): 22
     – A short poem illustrating the questioning nature of philosophy.

(b) Websites:

- Cline, Austin (2012), "Introduction to Philosophy: Defining, Studying, Doing Philosophy Is Important" (about.com)
  http://atheism.about.com/od/aboutphilosoph1/a/philosophyintro.htm
- Dennett, Daniel; & Steglich-Petersen, Asbjørn (2008), "The Philosophical Lexicon", http://www.philosophicallexicon.com/
  - A satirical dictionary of philosophical jargon.
- Florian, Meghan (2013), "Notes from an Employed Philosopher", *Femmonite* (31 January),
  http://www.femmonite.com/2013/01/notes-from-employed-philosopher.html
  - Interesting comments about how philosophy can make you "feel alive" and prepare you for the "real world".
- Philosophers on Their Conceptions of Philosophy (Leiter Reports: A Philosophy Blog, June 2010),
  http://leiterreports.typepad.com/blog/2010/06/philosophers-on-their-conception-of-philosophy.html
- Philosophical Fun (Department of Philosophy, University of Rochester)
  http://www.rochester.edu/College/PHL/fun.html
- Rosenberg, Alex (2011), "What Exactly Is Philosophy of Science—and Why Does It Matter?",
  http://www.routledge.com/philosophy/articles/what_exactly_is_philosophy_of_science_and_why_does_it_matter/
- Sinnott-Armstrong, Walter (2004), "What Is Philosophy?"
  http://ismailispirit.wordpress.com/apporved-what-is-philosophy/
- Shammas, Michael (2012), "For a Better Society, Teach Philosophy in High Schools", *Huffington Post* (26 December),
  http://www.huffingtonpost.com/mike-shammas/for-a-better-society-teac_b_2356718.html
- Suber, Peter (2003), Guide to Philosophy on the Internet,
  http://www.earlham.edu/∼peters/philinks.htm
- What Is Philosophy? (Department of Philosophy, University of Nevada, Reno),
  http://www.unr.edu/philosophy/2-what_is_philosophy.html
- Why Philosophy? (Department of Philosophy, SUNY Buffalo),
  http://www.buffalo.edu/cas/philosophy/undergrad-study/why.html
- Why Study Philosophy? (Indiana University Department of Philosophy),
  http://www.indiana.edu/∼phil/undergraduate/why.shtml
- AskPhilosophers.org questions:
  i. What do people mean when they speak of "doing" philosophy?
  http://www.askphilosophers.org/question/2915
  ii. Why are philosophers so dodgy when asked a question?
  http://www.askphilosophers.org/question/2941
  iii. Are there false or illegitimate philosophies, and if so, who's to say which ones are valid and which are invalid?
  http://www.askphilosophers.org/question/2994
  iv. What does it take to be a philosopher?
  http://www.askphilosophers.org/question/4609

3. How to Study and Write Philosophy:

   (a) Books:

   - Martinich, A.P. (2005), *Philosophical Writing: An Introduction* (Malden, MA: Blackwell).
   - Woodhouse, Mark B. (2007), *A Preface to Philosophy, 8th edition* (Wadsworth Publishing).

   (b) Websites:

   - Fiala, Andrew (2004), Student Survival Guide, http://www.wadsworth.com/philosophy_d/special_features/popups/survival_guide.html
   - Miller, David Marshall (2011), "Writing Philosophy Papers", http://people.duke.edu/∼dmmiller/Classes/Writing%20Philosophy%20Papers.docx
   - Suber, Peter (2003), Courses, http://www.earlham.edu/ peters/courses.htm
     - See especially the links at the bottom of the page under the heading "Files and links that pertain to more than one course".
   - Do a computer search for: "writing philosophy papers" (include the quotes in the search box)

The Philosopher within You

There's the legend of the fish who swam around asking every sea creature he'd meet, "Where is this great ocean I keep hearing about?"...

We are very much like that fish.

For consider, it's hard to look at a newborn baby without thinking: what an incredible miracle. But when was the last time you looked at an adult and had the same thought? But why not? Every adult *was* a little baby; if the latter is a miracle then so is the former. But it never occurs to us to think this way for one simple reason: we're so used to seeing people that we stop reflecting on them.

Or you drop something, a spoon, and it falls to the floor. But why? Couldn't it, in theory, have remained floating in air or moved upwards? And *how* exactly does it fall to the floor, by "gravity"? There are no strings connecting the earth to the spoon. How can the earth pull on something froma distance, that it's not even attached to? Why don't we pause every time something drops and say: what an incredible miracle!

The most ordinary things contain a whole lifetime of questions, if only we are reminded to start asking them.

Children already know to ask these questions. Every answer you provide to one of their "Why?" questions just generates the next question. But we were all children once. What we need to do now is to let the child still within us—the philosopher within us—re-emerge. ... We need to take a cold wet plunge into the great deep ocean of thought.

It's time to start thinking.

—Andrew Pessin, *The 60-Second Philosopher* (London: Oneworld): 6–7.

Part II

Computer Science, Computation, and Computers

Chapter 3

What Is Computer Science?

Version of 6 September 2019[1] DRAFT © 2004–2019 by William J. Rapaport

> Thanks to those of you who [gave their own] faculty introductions [to the new graduate students]. For those who [weren't able to attend], I described your work and courses myself, and then explained via the Reductionist Thesis how *it all comes down to strings and Turing machines operating on them.*
> — Kenneth Regan, email to University at Buffalo Computer Science & Engineering faculty (27 August 2004); italics added.

> The Holy Grail of computer science is to capture the messy complexity of the natural world and express it algorithmically.
> — Teresa Marrin Nakra, quoted in (Davidson, 2006, p. 66).

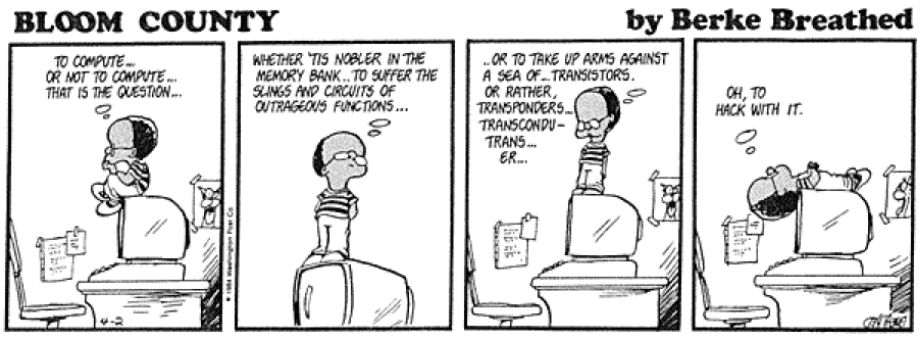


Figure 3.1: ©1984, Washington Post Co.

[1] An earlier version of this chapter appears as Rapaport 2017c.

3.1 Readings

1. Required:

   (a) Newell, Allen; Perlis, Alan J.; & Simon, Herbert A. (1967), "Computer Science", *Science* 157(3795) (22 September): 1373–1374.

   (b) Knuth, Donald (1974), "Computer Science and Its Relation to Mathematics", *American Mathematical Monthly* 81(4) (April): 323–343.

   - required: §§1–3
   - very strongly recommended: §4
   - strongly recommended: readers who are more mathematically inclined may wish to read the whole essay.

   (c) Newell, Allen; & Simon, Herbert A. (1976),
   "Computer Science as Empirical Inquiry: Symbols and Search",
   Communications of the ACM 19(3) (March): 113–126.

   i. For the purposes of this chapter, concentrate especially on what Newell & Simon have to say about what CS is:
   - read the "Introduction" (pp. 113–114)
   - read from "§I. Symbols and Physical Symbol Systems" to the end of the subsection "Physical Symbol Systems" (pp. 114–117)
   - read the "Conclusion" (pp. 125–126)

   ii. For a detailed follow-up, see:
   Newell, Allen (1980), "Physical Symbol Systems", *Cognitive Science* 4: 135–183,
   http://repository.cmu.edu/cgi/viewcontent.cgi?article=3504&context=compsci

   (d) Hartmanis, Juris, & Lin, Herbert (1992), "What Is Computer Science and Engineering?", in Juris Hartmanis & Herbert Lin (eds.), *Computing the Future: A Broader Agenda for Computer Science and Engineering* (Washington, DC: National Academy Press), Ch. 6, pp. 163–216.

   - required: "Computer Science & Engineering", pp. 163–168.
   - required: "Abstractions in Computer Systems", pp. 168–174.
   - very strongly recommended: skim the rest.
   - The book containing this essay was the subject of a petition, sponsored by Bob Boyer, John McCarthy, Jack Minker, John Mitchell, and Nils Nilsson, to withdraw it from publication "because we consider it misleading and even harmful as an agenda for future research" (http://www-formal.stanford.edu/jmc/petition/whysign/whysign.html). Commentaries on it appeared in (Kling et al., 1993).

   (e) Brooks, Frederick P., Jr. (1996), "The Computer Scientist as Toolsmith II", *Communications of the ACM* 39(3) (March): 61–68,
   http://www.cs.unc.edu/∼brooks/Toolsmith-CACM.pdf

   - required: pp. 61–64.
   - very strongly recommended: skim the rest.

   (f) Shapiro, Stuart C. (2001), "Computer Science: The Study of Procedures",
   http://www.cse.buffalo.edu/~shapiro/Papers/whatiscs.pdf

2. Very Strongly Recommended (arranged in chronological order):

   (a) Arden, Bruce W. (1980), "COSERS Overview", in Bruce W. Arden (ed.), *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)* (Cambridge, MA: MIT Press), Ch. 1, pp. 1–31.

   (b) Krantz, Steven G. (1984), Letter to the Editor about the relation of computer science to mathematics, *American Mathematical Monthly* 91(9) (November): 598–600.

   (c) Denning, Peter J. (1985), "What Is Computer Science?", *American Scientist* 73 (January-February): 16–19.

   (d) Abrahams, Paul (1987), "What Is Computer Science?", *Communications of the ACM* 30(6) (June): 472–473.

   (e) Loui, Michael C. (1987), "Computer Science Is an Engineering Discipline", *Engineering Education* 78(3) (December): 175–178.

   (f) Bajcsy, Ruzena K.; Borodin, Allan B.; Liskov, Barbara H.; & Ullman, Jeffrey D. (1992), "Computer Science Statewide Review" (unpublished report), http://www.cse.buffalo.edu/~rapaport/Papers/Papers.by.Others/bajcsyetal92.pdf

   (g) Gal-Ezer, Judith, & Harel, David (1998), "What (Else) Should CS Educators Know?", *Communications of the ACM* 41(9) (September): 77–84.

      - contains a section titled "What Is CS?"
      - contains a "Bibliography for 'What Is CS?' "

   (h) Hartmanis, Juris (1993), "Some Observations about the Nature of Computer Science", in Rudrapatna Shyamasundar (ed.), *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 761 (Berlin: Springer): 1–12, https://www.researchgate.net/publication/221583809_Some_Observations_About_the_Nature_of_Computer_Science

   (i) Hartmanis, Juris (1995), "On Computational Complexity and the Nature of Computer Science", *ACM Computing Surveys* 27(1) (March): 7–16; see footnote 7, below.

   (j) Jacob, Christian (1999), "What Is Computer Science?" http://pages.cpsc.ucalgary.ca/~jacob/Courses/Fall99/CPSC231/03-ComputerScience.pdf

   (k) Shagrir, Oron (1999), "What Is Computer Science About?", *The Monist* 82(1): 131–149.

      - Despite its title, this paper is more about what *computers* are and what *computation* is; Shagrir assumes that computer science is the science of computers.

   (l) Johnson, George (2001), "All Science Is Computer Science", *The New York Times* (25 March): WK1, WK5; http://www.nytimes.com/2001/03/25/weekinreview/the-world-in-silica-fertilization-all-science-is-computer-science.html

   (m) Foley, Jim (2002), "Computing > Computer Science", *Computing Research News* 14(4) (September): 6, http://archive.cra.org/reports/computing/index.html

      - See also:
        Glass, Robert L. (2003), Response to "Computing > Comptuer Science", http://archive.cra.org/reports/computing/glass.html

   (n) Boston University Department of Computer Science (2003), "What Is Computer Science?", http://cs-www.bu.edu/AboutCS/WhatIsCS.pdf

(o) Parlante, Nick (2005), "What Is Computer Science?", *Inroads—The SIGCSE Bulletin* 37(2) (June): 24–25.

(p) Naur, Peter (2007), "Computing Versus Human Thinking", *Communications of the ACM* 50(1) (January): 85–94.

3.2 Introduction

The fundamental question of this book is:

> **What is computer science?**

Almost all of the other questions we will be considering flow from this one. (Is it a *science*? Is it the science *of computers*? What is science? What is a computer? And so on.) In this chapter, we will look at several definitions of the term 'computer science'. Each definition raises issues that we will examine in more detail later, so a final answer (if there is one!) will have to await the end of the book. However, at the end of this chapter, I will give a summary characterization of computer science that, I think, combines important aspects of the various definitions, with the details to be filled in as we go along.

3.3 Preliminary Questions

Before we try to answer the question, it's worth asking some preliminary questions:

> **What should this discipline be called?**
> **Why should we even bother seeking a definition?**
> **What does it mean to give a definition?**

3.3.1 Naming the Discipline

> When our discipline was newborn, there was the usual perplexity as to its proper name. (Brooks, 1996, p. 61)

Should we call the discipline 'computer science' (which seems to assume that it is the *science* of a certain kind of *machine*), or 'computer *engineering*' (which seems to assume that it is *not* a science, but a branch of engineering), or '*computing* science' (which seems to assume that it is the science of what those machines *do*), or 'informatics' (a name more common in Europe), or something else altogether?

Michael Mahoney (a historian of computer science) asks if wondering whether computer science is a science, given its name, is "laboring under a misapprehension rooted in the English use of 'computer science' to denote a subject other languages refer to as 'informatics' " (Mahoney, 2011, p. 195): 'Informatics' is a term that suggests that the discipline is the mathe*matical* study of *infor*mation. But he then goes on to point out that textbooks and courses from "informatics" departments cover exactly the same material as textbooks and courses from "computer science" departments:

> So I must wonder, as Shakespeare's Juliet once did, "What's in a name?" Here too the rose smells the same. (Mahoney, 2011, p. 195)

In this book—but only for convenience—I will call it 'computer science'. However, by doing so, I do not mean to presuppose that it is the science of computers. Worse, it is a standard joke in academe that any discipline that feels the need to call itself a science

(such as political science, library science, exercise science, and others) is therefore not one.[2] Nor do I mean to exclude the other parts of the discipline, such as engineering or the role of information.

So, until we have an answer to our question, think of the subject as being called by a 15-letter word 'computerscience' that may have as little to do with computers or science as 'cattle' has to do with cats. Or, to save space and to suppress presuppositions, I'll often just refer to it as "CS".

Further Reading:
For those who are interested, (Ceruzzi, 1988, esp. pp. 265–270) contains a history of the phrase 'computer science'. And, in a response to a letter that appeared in one of the earliest issues of *Communications of the ACM*, an editor (possibly Alan J. Perlis, whom we will meet again below) listed several, admittedly "facetious", names, including 'turingineering', 'turology', 'applied meta-mathematics', and 'applied epistemology' (DATA-LINK, 1958, p. 6). (The first two are puns on the name of Alan Turing, arguably the founder of the discipline, whom we will discuss in Chapter 8. We'll come back to "applied epistemology" in §3.14.4, below.) In 1966, Peter Naur (a winner of the Turing Award) suggested 'datalogy' (Naur, 2007, p. 86). A useful discussion of these terms can be found in (Arden, 1980, pp. 5–7, "About Names and Labels").

The A.M. Turing Award, given annually by the Association for Computing Machinery, is considered to be the "Nobel Prize" of computer science. See http://amturing.acm.org/, https://en.wikipedia.org/wiki/Turing_Award, and (Vardi, 2017).

3.3.2 Why Ask What CS Is?

With the question of its name put aside, we can now turn to the question of why we might want a definition. There are at least two kinds of motivations for doing so, academic (or political) ones and intellectual (or philosophical) ones.

3.3.2.1 Academic Motivations

Among the academic motivations, there are political, pedagogical, and publicity motivations.

3.3.2.1.1 Academic Politics. Here is an academic political reason for asking what CS is:

> **Where should a "computer science" department be adminstratively housed?**

Intellectually, this might not matter: After all, a small school might not even have academic departments, merely teachers of various subjects. But deciding where to place a CS department can have political repercussions:

[2] This joke has been attributed to the philosopher John Searle, whom we will meet again many times in this book; see http://duncan.hull.name/2011/07/01/but-is-it-science/

> In a purely intellectual sense such jurisdictional questions are sterile and a waste of time. On the other hand, they have great importance within the framework of institutionalized science—e.g., the organization of universities and of the granting arms of foundations and the Federal Government. (Forsythe, 1967b, p. 455)

Sometimes, a department is housed in a particular school or college[3] only because it is hoped that it will get better treatment there (more funding, more resources), or only because it is forced to be there by the administration. It may have very little, if any, academic or intellectual reason for being housed where it is. Some possible locations for CS include:

- a college or school of **arts and sciences**
  - which typically includes other departments in the humanities, social sciences, and natural sciences
- a college or school of **engineering**
  - which typically includes disciplines such as chemical engineering, electrical engineering, mechanical engineering, etc.
- a college or school of **informatics**
  - which might also include disciplines such as communications, library science, etc.

Another possibility is that CS should not be (merely) a department, but an entire school or college itself, with its own dean, and perhaps with its own departments. For example, the School of Computer Science at Carnegie-Mellon University includes a Department of Computer Science, a Department of Computational Biology, and a Department of Machine Learning, among others.

There are examples of each of these, even within a single university system: (1) My own university (State University of New York at Buffalo) currently has a Department of Computer Science and Engineering within a School of Engineering and Applied Sciences. However, when I joined the university, there were both a Department of Computer Science in a Faculty of Natural Sciences and Mathematics and a separate Department of Electrical and Computer Engineering in a Faculty of Engineering and Applied Sciences.[4]

(2) At its sibling institution, State University of New York at Albany, the Department of Computer Science was in the College of Computing and Information in 2012; however, now (2016) there is a Department of Computer Science, a Department of Information Studies, a Department of Informatics, and a Department of Computer Engineering in a College of Engineering and Applied Science.

[3] In the US, colleges and universities are usually administratively divided into smaller units, variously known as 'schools', 'colleges', 'faculties', 'divisions', etc., each typically headed by a "dean" and divided into still smaller units, called 'departments'.

[4] The view of CS as the natural sciences of procedures, which we will look at in §3.9.3, was motivated by our department's move into the engineering school (S.C. Shapiro, letter to Adam Olszewski, 10 May 2018).

(3) And, at my former college, State University of New York College at Fredonia, CS courses were once taught only in the Department of Mathematics; now, there is a Department of Computer and Information Sciences in the College of Liberal Arts and Sciences.

3.3.2.1.2 Academic Pedagogy. Perhaps a more important academic purpose for asking what CS is concerns pedagogy:

What should be taught in an introductory CS course?

- Should it be a **programming** course?
  (That is, is CS the study of programming?)
  - Or, worse, should students be led to *think* that that's what it is? I don't know any computer scientists who think that CS is *just* the study of programming (Denning et al., 2017), but the typical introductory course tends to lead students (and the general public) to think so.
- Should it be a **computer literacy** course?
  (That is, is CS all about how to *use* computers?)
- Should it be a course in the **mathematical theory of computation**?
  (That is, is CS the study of computation?)
- Should it be a course that introduces students to several different branches of CS, including, perhaps, some of its history?

And so on.

3.3.2.1.3 Academic Publicity. A related part of the academic purpose for asking the question concerns **publicity for prospective students and the general public**:

- How should a CS department advertise itself so as to attract good students?
- How should the discipline of CS advertise itself so as to encourage primary- or secondary-school students to consider it as something to study in college or to consider it as an occupation?
  (For more motivations along these lines, see (Denning, 2013b, p. 35).)
- How should the discipline advertise itself so as to attract more women and minorities to the field?
- How should it advertise itself to the public at large, so that ordinary citizens might have a better understanding of what CS is?

Exercise for the Reader:
Many of the definitions of CS that you can find on various academic websites are designed with one or more of these purposes in mind. Link to the websites for various CS departments (including your own school's!), and make a list of the different definitions or characterizations of CS that you find. See if you can figure out whether they were designed with any of these purposes in mind.

3.3.2.2 Intellectual or Philosophical Motivations

Perhaps the academic (and especially political) motivations for asking what CS is are ultimately of little more than practical interest. But there are deep intellectual or philosophical issues that underlie those questions, and this will be the focus of our investigation:

- **What is CS "really"?**
  - Is it like some other academic discipline?

    (For instance, is it like physics, or mathematics, or engineering?)
  - Or is it "*sui generis*"?

    ('*Sui generis*' is a Latin phrase meaning "own kind".)[5]

To illustrate this difference, consider two very different comments by two Turing-award–winning computer scientists (as cited in (Gal-Ezer and Harel, 1998, p. 79)): Marvin Minsky, a co-founder of artificial intelligence, once said:

> Computer science has such *intimate relations* with so many other subjects that *it is hard to see it as a thing in itself.* (Minsky, 1979, my italics)[6]

On the other hand, Juris Hartmanis, a founder of computational complexity theory, has said:

> Computer science *differs* from the known sciences so deeply that it has to be viewed as *a new species among the sciences.*
> (Hartmanis, 1993, p. 1; my italics) (see also (Hartmanis, 1995a, p. 10))[7]

So, is CS like something "old", or is it something "new"? But we have yet another preliminary question to consider ...

---

[5] Here is a simple analogy: A poodle and a pit bull are both kinds of dogs. But a wolf is not a dog; it is its own kind of animal ("*sui generis*"). Some biologists believe that dogs are actually a kind of wolf, but others believe that dogs are *sui generis*.

[6] This echoes an earlier statement by another computer scientist: "Probably a department of computer science belongs in the school of letters and sciences, because of its close ties with departments of mathematics, philosophy, and psychology. But its relations with engineering departments ... should be close" (Forsythe, 1967a, p. 6).

[7] (Hartmanis, 1995a) covers much of the same ground, and in many of the same words, as (Hartmanis, 1993), but is more easily accessible, having been published in a major journal that is widely available online, rather than in a harder-to-find conference proceedings. Moreover, (Hartmanis, 1995a) contains commentaries (including (Denning, 1995), (Loui, 1995), (Plaice, 1995), (Stewart, 1995), (Wulf, 1995)) and a reply by the author (Hartmanis, 1995b).

3.3.3 What Does It Mean to Ask What Something Is?

> It does not make much difference how you divide the Sciences, for they are one continuous body, like the ocean. (Leibniz, 1685, p. 220)

> We will not try to give a few-line definition of computer science since no such definition can capture the richness of this new and dynamic intellectual process, *nor can this be done very well for any other science.* (Hartmanis, 1993, p. 5; my italics)

3.3.3.1 Determining Boundaries

> *We should quell our desire to draw lines.* We don't need to draw lines. —Daniel C. Dennett (2013a, p. 241)

> [O]ne of Darwin's most important contributions to thought was his denial of *essentialism*, the ancient philosophical doctrine that claimed that for each type of thing, each natural kind, there is an *essence*, a set of necessary and sufficient properties for being that kind of thing. Darwin showed that different species are historically connected by a chain of variations that differed so gradually that there was simply no principled way of drawing a line and saying (for instance) dinosaurs to the left, birds to the right. —Dennett (2017, pp. 138–139)

There is a fundamental principle that should be kept in mind whenever you ask what something is, or what kind of thing something is: **There are no sharp boundaries in nature**; there are only continua.

A "continuum" (plural = 'continua') is like a line with no gaps in it, hence no natural places to divide it up. The real-number line is the best example.[8] Another is the color spectrum: Although we can identify the colors red, orange, yellow, green, blue, and so on, there are no sharp (or non-arbitrary) boundaries where red ends and orange begins; in fact, one culture's "blue" might be another's "green" (see (Berlin and Kay, 1969), (Grey, 2016)). Yet a third example is the problem of assigning letter grades to numerical scores. If many of the numerical scores are equally close to each other, there is often no natural (or non-arbitrary) reason why a score of (say) 75 should be assigned a letter grade of (say) 'B−' while a 74 is a 'C+'. (For a history and philosophy of grading, see (Rapaport, 2011a).)

An apparent counterexample to the lack of sharp boundaries in nature might be biological species: Dogs are clearly different from cats, and there are no "intermediary" animals—ones that are not clearly either dogs or else cats. But both dogs and cats evolved from earlier carnivores (it is thought that both evolved from a common ancestor

[8] The *natural* numbers (1, 2, 3, ...) clearly have gaps, because there are non-natural (for example, rational) numbers that separate 1 from 2 (that is, that are *between* 1 and 2), and so on. The *rational* numbers are "dense"; that is, between any two rationals, there is another rational number (for example, their average). Nevertheless, there *are* gaps: irrational numbers (real numbers, such as $\sqrt{2}$ or π) separating any two rationals. (In fact, this separation property underlies Dedekind's definition of real numbers in terms of "cuts". See https://en.wikipedia.org/wiki/Dedekind_cut for an informal presentation and (Rudin, 1964, 3–10) for a more rigorous treatment.)

some 42 million years ago).[9] If we traveled back in time, we would not be able to say whether one of those ancestors was a cat or a dog; in fact, the question wouldn't even make sense. It is also very difficult to give a definition of a "natural kind" (say, dog).

Moreover, although logicians and mathematicians like to define categories in terms of "necessary and sufficient conditions" for membership, this only works for abstract, formal categories. For example, we can define a circle of radius r and center c as the set of *all* and *only* those points that are r units distant from c.

Philosophical Digression:
"All" such points is the "sufficient condition" for being a circle; "only" such points is the "necessary condition": C is a circle of radius r at center c **if and only if** $C = \{p : p \text{ is a point that is } r \text{ units distant from } c\}$. That is, p is r units from c **only if** p is a point on C (that is, **if** p is r units from c, **then** p is a point on C); so, being a point that is r units from c **is a sufficient condition for** being on C. And **if** p is a point on C, **then** p is r units from c; so, being a point that is r units from c **is a necessary condition for** being on C.

However, as philosophers, psychologists, and cognitive scientists have pointed out, non-abstract, non-formal ("real") categories usually don't have such precise, defining characteristics. The most famous example is the philosopher Ludwig Wittgenstein's unmet challenge to give necessary and sufficient defining characteristics for something's being a game (Wittgenstein, 1958, §66ff). Instead, he suggested that games (such as solitaire, basketball, chess, etc.) all share a "family resemblance": The members of a family don't necessarily all have the same features in common (having blue eyes, being tall, etc.), but instead resemble each other (mother and son, but not father and son, might have blue eyes; father and son, but not mother and son, might both be tall, and so on). And the psychologist Eleanor Rosch has pointed out that even precisely definable, mathematical categories can have "blurry" edges: Most people consider 3 to be a "better" example of a prime number than, say, 251, or a robin to be a "better" example of a bird than an ostrich is.

Further Reading:
On Wittngenstein's notion of "game", see Hoyningen-Huene 2015.

On categorization, see Rosch and Mervis 1975; Rosch 1978; Mervis and Rosch 1981; Lakoff 1987; and Hofstadter and Sander 2013, esp. Ch. 7.

In his dialogue *Phaedrus*, Plato suggested that a good definition should "carve nature at its joints" (Plato, 1961a, lines 265e–266a). But, if "nature" is a continuum, then there are no "joints". Hence, we do not "carve nature at *its* joints"; rather, we "carve nature" at "joints" that are usually of our *own* devising: **We impose our own categories on nature.**

[9] See the GreenAnswers.com webpage, http://greenanswers.com/q/95599/animals-wildlife/pets/what-common-ancestor-cats-and-dogs (which cites the *Wikipedia* articles "Feliformia" (http://en.wikipedia.org/wiki/Feliformia) and "Carnivora" (http://en.wikipedia.org/wiki/Carnivora)). Also see the gather.com webpage "Miacids", http://www.gather.com/viewArticle.action?articleId=281474977041872. (A *Ziggy* cartoon (http://www.gocomics.com/ziggy/2013/02/19), however, jokingly suggests that a platypus could be considered as an intermediary animal between birds and mammals!)

But I would not be a good philosopher if I did not immediately point out that, just as Plato's claim is controversial, so is this counter-claim! After all, isn't the point of science to describe and explain a reality that exists independently of us and of our concepts and categories—that is, independently of the "joints" that *we* "carve" *into* nature? (We'll return to the topic of the goal of science in Chapter 4: "What Is Science?".) And aren't there "natural kinds"? Dogs and cats, after all, do seem to be kinds of things that are there in nature, independently of us, no matter how hard it might be to *define* them.

Is CS similar to such a "natural kind"? Here, I think the answer is that it pretty clearly is not. There would be no academic discipline of CS without humans, and there probably wouldn't even be any computers without us, either (though we'll see some reasons to think otherwise, in Chapter 9: "What Is a Computer? A Philosophical Perspective").

Exercise for the Reader:
Take the list of definitions of CS that you found from the exercise at the end of §3.3.2.1.3. Do you agree with them? Do they agree with each other? Are any of them so different from others that you wonder if they are really trying to describe the same discipline?

Perhaps advertising blurbs like the ones you find in this exercise should not be taken too seriously. But the authors of several published essays that try to define 'computer science'—all of whom are well-respected computer scientists—presumably put a lot of thought into them. They *are* worth taking seriously, which is the main purpose of this chapter.

Before turning to those, let's consider a few examples of other familiar terms whose definitions are controversial.

3.3.3.2 Three Other Controversial Terms

> When sharp formulations are offered for concepts that had been vague, they sometimes result in bizarre rulings along the edges, bizarre but harmless. —Willard van Orman Quine (1987, p. 217)

3.3.3.2.1 What Is a Planet? Consider the case of poor Pluto—not Mickey Mouse's dog, but the satellite of the Sun: It used to be considered a planet, but now it's not, because it's too small. I don't mean that it is now *not a planet* because of its size. Rather, I mean that now it is *no longer considered to be* a planet because of its size.

Moreover, if it were to continue being categorized as a planet, then we would have to count as planets many other small bodies that orbit the Sun, eventually having to consider all (non-human-made) objects in orbit around the Sun as planets, which almost makes the term useless, because it would no longer single out some things (but not others) as being of special interest.

Further Reading and Philosophical Digression:
On Pluto, see, for example, (Lemonick, 2015). This is an example of a "slippery-slope" argument: Once you decide to categorize a certain object O in a certain way, you find that you are committed to also categorizing objects that differ only very insignificantly from O in that way, and so on, eventually categorizing *all* objects that way, thus "sliding down a slippery slope". The classic example is 'heap': A pile of, say, 10^6 grains of sand is surely a heap of sand; So, presumably, are a pile of $10^6 - 1$ grains, a pile of $10^6 - 2$ grains, and a pile of $10^6 - 3$ grains. In general, if you have a "heap" H of sand, surely removing 1 grain will not change H from being a heap to no longer being a heap. But removing 1 grain at a time will eventually leave you with a pile of 3 grains, a pile of 2 grains, and a pile of 1 grain. And, clearly, 1 grain of sand is *not* a "heap" of sand! Although not all slippery-slope arguments are unsound, they tend to point out a problem with the term being used, usually that the term is vague.

To make matters even worse, the Moon was once considered to be a planet! When it was realized that it did not orbit the Sun directly, it was "demoted". But, curiously, under a proposed new definition of 'planet' (as having an "orbit-clearing mass") (Battersby, 2015), it might turn out to be (considered as) a planet once more!

Note that, in either case, the *universe* has not changed; only our *descriptions* of it have:

> Exact definitions are undoubtedly necessary but are rarely perfect reflections of reality. Any classification or categorization of reality imposes arbitrary separations on spectra of experience or objects. (Craver, 2007)

So, depending on how we define 'planet', either something that we have always considered to *be* one (Pluto) might turn out *not* to be one, or something that we have (usually) *not* considered to be one (the Moon) might turn out to *be* one! Typically, when trying to define or "formalize" an informal notion, one finds that one has *excluded* some "old" things (that is, things that were informally considered to fall under the notion), and one finds that one has *included* some "new" things (that is, things that one hadn't previously considered to fall under the notion). Philosopher Ned Block has called the former kind of position "chauvinism" and the latter position "liberalism" (Block, 1978, pp. 263, 265–266, 277). When this happens, we can then either reformulate the definition, or else bite the bullet about the inclusions and exclusions. One attitude towards exclusions is often that of sour grapes: Our intuitions were wrong; those things really weren't Xs after all. The attitude towards inclusions is sometimes: Wow! That's right! Those things really *are* Xs! Alternatively, a proposed definition or formalization might be *rejected* because of its chauvinism or its liberalism.

Here is an observation about this point:

> In mathematics, tentative proofs are sometimes 'refuted' not by logical contradiction ..., but instead by a logically-sound consequence ... that is unexpected and unwanted .... Lakatos [1976] argues that in such situations mathematicians should not just implicitly reject the unwelcome result ('monster barring'), but should instead either change their acceptance of the result ('concept stretching' ...), or change the statement of the conjecture either by explicitly incorporating a condition that disallows the unwanted consequence ('lemma incorporation' ...), or by inventing a wholly new conjecture. (Staples, 2015, §3.1)

3.3.3.2.2 What Is Computation? The next two cases will be briefer, because we will discuss them in more detail later in the book. The first is the very notion of 'computation' itself: According to the Church-Turing Computability Thesis, a function is computable if and only if it is computable by a Turing Machine. (If you don't yet know what these terms are, be patient; we will begin discussing them in Chapter 7: "What Is an Algorithm?".)[10] This is neither a definition of 'computable' nor a mathematical theorem; it is a suggestion about what the *informal* notion of "computability" should mean. But some philosophers and computer scientists believe that there are functions that *are* informally computable but *not* computable by a Turing Machine. (We'll discuss these in Chapter 11: "What Is Hypercomputation?".)

3.3.3.2.3 What Is Thinking? Our final case is the term 'thinking': If thinking is categorized as any process of the kind that cognitive scientists study—including such things as believing, consciousness, emotion, language, learning, memory, perception, planning, problem solving, reasoning, representation, sensation, etc. (Rapaport, 2012b, p. 34)—then it is (perhaps!) capable of being carried out by a computer. Some philosophers and computer scientists accept this way of thinking about thinking, and therefore believe that computers will eventually be able to think (even if they do not yet do so). Others believe that if computers can be said to think when you accept this categorization, then there must be something wrong with the categorization. (We'll expore this topic in Chapter 19: "Philosophy of Artificial Intelligence".)

Further Reading:
Related to 'thinking' is 'cognition'. On the difficulty of defining that term (as well as general remarks on defining controversial terms, see Allen (2017).

Another example is the definition of 'life' (Machery, 2012), (Allen, 2017, p. 4239). We'll come back to this example in §10.2, and in our discussion of what Daniel Dennett has called "Turing's Strange Inversion" (Chapter 19).

(Angere, 2017) is another case study, which shows how even 'square' and 'circle' may have counterintuitive definitions, allowing for the (mathematical) existence of square circles (or round squares)!

3.4 Two Kinds of Definition

An "extensional" definition of a term t is given by presenting the *set* of items that are considered to be ts. For example, the (current) extensional definition of 'US President' is {Washington, Adams, Jefferson, ..., Obama, Trump}. For another example, we once might have said that x is a planet (of the Sun) iff $x \in$ {Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto}. Now, however, we say that x is a planet (of

[10] Should 'machine' be capitalized in 'Turing Machine'? Not capitalizing it suggests that a Turing Machine is a machine of some kind. But machines are typically physical objects, and a Turing Machine is an abstract mathematical notion. Capitalizing 'machine' turns it into a proper name, allowing us to ask whether a Turing Machine is or is not a machine without begging any questions.

the Sun) iff $x \in \{$Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune$\}$. Note that these two extensional definitions of 'planet' are *different*.

An "intensional" definition can be given in terms of necessary and sufficient conditions or in terms of a family resemblance. For example, an intensional definition of 'US President' might be given by citing Article II of the US Constitution: Roughly, x is US President iff x has been vested with the executive power of the US. Note that this intensional definition holds even if an extensional definition changes (such as the extensional definitions in the previous paragraph of 'US President', which change roughly every 4 or 8 years).

Two concepts can be said to be "extensionally equivalent" if exactly the same sets of things fall under each concept. Importantly, two extensionally equivalent concepts can be (and usually are) "intensionally distinct"; that is, they really are different concepts. Here is an important example from computability theory (which we'll look at in detail in Chapter 7): Recursive function theory and the theory of Turing machines are extensionally equivalent but intensionally distinct. They are extensionally equivalent because it is mathematically provable that all functions that are recursive are Turing computable, and vice versa. But they are intensionally distinct because the former is concerned with a certain way of defining mathematical functions, while the latter is concerned with algorithms and computation. From the point of view of what facts can be proved about functions, it doesn't matter which formalism is used, because they are extensionally equivalent. Their intensional distinctness comes into play when one formalism might be easier or more illuminating to use in a given situation. We'll come back to this in §3.7.

Further Reading:
For more on extensions and intensions, see Rapaport 2012a.

3.4.1 An Extensional Definition of CS

To the extent that it is we who impose *our* categories on nature, there may be no good answer to the question "What is CS?" beyond something like: "Computer science" *is* what computer scientists *do*. In a similar vein, (Abrahams, 1987, p. 472) says "computer science is that which is taught by computer science departments". Perhaps intended more seriously, (Denning, 2000, p. 1) defines "The discipline of computer science ... [as] the body of knowledge and practices used by computing professionals in their work." But then we can ask: *What is it that computer scientists do?* Of course, one can beg that last question—that is, argue in a circle—by saying that computer scientists do computer science! (Hamming, 1968, p. 4) suggests something like this, citing the (humorous) "definition" of mathematics as "what mathematicians do", but he goes on to point out that "there is often no clear, sharp definition of ... [a] field".[11]

[11] (Bringsjord, 2006, my emphasis) argued that "any answer ... that *includes* some such notion as 'Whatever computer scientists actually do.' is unacceptable." I would replace 'includes' by 'is limited to'; surely, 'includes' has to be included.

Philosophical Digression:
'To beg the question' is a slightly archaic term of art in philosophy and debating. The phrase does *not* mean: "to *ask* a question"—that is, to "beg" in the sense of "to raise or invite" a question. In debating, a "question" is the *topic* being debated. 'To beg the question' means: "to *request* (that is, "to beg") that the *topic being debated* (that is, the "question") be *granted as an assumption* in the debate". That is, it means "to assume as a premise ("to beg") the conclusion ("the question") that you are arguing for". A modern synonymous phrase for 'beg the question' is: 'argue in a circle'.

As with most non-mathematical concepts, there are probably no necessary and sufficient conditions for being CS. At best, the various branches of the discipline share only a family resemblance. If no intensional definition can be given in terms of necessary and sufficient conditions, perhaps an extensional one can: "Computing has no nature. It is what it is because people have made it so" (Mahoney, 2011, p. 109). This is not exactly of the form "CS is what computer scientists do", though it bears a superficial resemblance. But I think Mahoney's point is more subtle: Unlike the other natural sciences (for example, physics, chemistry, biology, and so on), CS only came into existence when its two histories (logical-mathematical and engineering) began to intersect in the 1940s, so its "nature" only came to be what those logicians, mathematicians, and engineers were doing. (We'll look into those twin histories in Chapter 6: "What Is a Computer? A Historical Perspective".)

Nevertheless, it's worth looking briefly at what computer scientists do. It has been said that CS is "a sort of spectrum ... with 'science' on the one end and 'engineering' on the other" (Parlante, 2005, p. 24), perhaps something like this:

abstract, mathematical theory of computations
abstract, mathematical theory of computational complexity
abstract, mathematical theory of program development
software engineering
...
operating systems
...
AI
...
computer architecture
...
VLSI
networks
social uses of computing, etc.

But this is less than satisfactory as a *definition*.

3.4.2 Intensional Definitions of CS

Instead of providing necessary and sufficient conditions, we can try to give an intensional definition by splitting the question of what CS is into two parts:

1. What is its **object**? (*What* does it study or investigate?)
2. What is its **methodology**? (*How* does it go about studying those objects?)

We'll begin with the second.

Is the *methodology* of CS the same as that of some other discipline? Or does it have its own, distinctive methodology. If the latter, is its methodology not only unique, but also something brand new? As for methodology, CS has been said to be (among many other things):

- an *art form*
  ((Knuth, 1974a, p. 670) has said that programs can be beautiful),
- an *art and science*
  ("Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it. ... [T]he process of going from an art to a science means that we learn how to automate something" (Knuth, 1974a, p. 668)),
- a *liberal art* (Perlis, 1962, p. 210), (Lindell, 2001)
  (along the lines of the classical liberal arts of logic, math, or astronomy),
- a branch of *mathematics* (Dijkstra, 1974),
- a *natural science* (McCarthy, 1963), (Newell et al., 1967), (Shapiro, 2001),
- an *empirical study* of the artificial (Simon, 1996b),
- a combination of *science and engineering*
  (Hartmanis, 1993), (Hartmanis, 1995a), (Loui, 1995),
- just *engineering* (Brooks, 1996),
- or—generically—a "study"

But a study (or a science, or an engineering, or an art, or ...) *of what*? Is its *object* the same as that of some other discipline? (Does it study exactly what science, or engineering, or math, or—for that matter—psychology or philosophy studies?) Or does it have its own, distinctive object of study (computers? algorithms? information?) Or does it study something that has never been studied before? The logician Jon Barwise (1989a) suggested that we can understand what CS is in terms of what it "traffics" in So here's an alphabetical list of some of the *objects* that it traffics in:

algorithms
automation
complexity
computers
information
intelligence
numbers (and other mathematical objects)
problem solving
procedures
processes
programming
symbol strings

This is, of course, only a very partial list. One can use computational methods to study pretty much any x. (For some examples of this, see the interview with the computer scientist Mehran Sahami in (Reese, 2014a)).

It is now time to look at some answers to our question in more detail.

3.5 CS Is the *Science* of *Computers*

The first such answer that we will look at comes from three Turing Award winners: Allen Newell, Alan Perlis, and Herbert Simon. (Newell and Simon were also cognitive scientists: Along with J.C. Shaw, they created one of the first AI programs, the Logic Theorist (Newell et al., 1958). And Simon was a winner of the Nobel prize in economics, in part for his work on "bounded rationality", which we'll look at briefly in §3.15.2.3.) Here is their definition, presented as the conclusion of an argument:

> Wherever there are phenomena, there can be a science to describe and explain those phenomena. ... There are computers. Ergo,[12] **computer science is the study of computers**. (Newell et al., 1967, p. 1373, my emphasis)

This argument is actually missing two premises. Their first two premises only imply that there *can* be a science of computers. They do not, by themselves, imply that there *is* such a science *or* that that science is CS rather than some other discipline. So, the missing premises are:

A. There *is* a science of computers.

B. There is no *other* discipline that is the science of computers besides CS.

[12] 'Ergo' is Latin for "therefore".

3.5.1 Objection to the First Premise

Newell, Perlis, & Simon's first premise is that, for any phenomenon[13] *p*, there can be a science of *p*. An objection to this that they consider is that this premise holds, not for *any* phenomenon, but only when *p* is a *natural* phenomenon. For example, the computer engineer Michael Loui (Loui, 1987, p. 175) notes that there are toasters, but no *science* of toasters.

The objection goes on to point out that computers aren't natural; they are *artifacts*. So, it doesn't follow that there can be (much less that there *is*) a *science* of computers. (It might still be the case that there is some other kind of discipline that studies computers (and toasters!), such as engineering.)

For example, computer scientist Bruce W. Arden argues that neither math nor CS are sciences, because their objects are not natural phenomena (Arden, 1980, p. 6). He says that the object of math is "human-produced systems ... concerned with the development of deductive structures". (We'll return to the relationship between CS and math in §3.9.1.) And he says that the object of CS is "man-made" [sic].[14] But what is the object? Computers? Yes, they're clearly human-made, and this leads us back to Newell, Perlis, & Simon's arguments. Algorithms? They're only human-made in whatever sense mathematical structures are. But, in §3.9.3, we'll look at a claim that algorithms are a special case of a *natural* entity ("procedures").

(Mahoney, 2011, pp. 159–161) discusses the objection that CS is not a natural science because "the computer is an artifact, not a natural phenomenon, and science is about natural phenomena". Mahoney rejects this, but not because he thinks that computers *are* natural phenomena. Rather, he rejects it because he thinks that there is no sharp dividing line "between nature and artifact" for two reasons: (1) because we *use* artifacts to study nature—"[w]e know about nature through the models we build of it"—and (2) because "[a]rtifacts work by the laws of nature, and by working reveal those laws". In other words, artifacts are *part* of nature. Philosopher Timothy Williamson makes a similar point about scientific instruments: "The scientific investigation of [a] physical quantity widens to include the scientific investigation of its interaction with our experimental equipment. After all, our apparatus is part of the same natural world as the primary topic of our inquiry" (Williamson, 2007, p. 43). The same could be said about computers and computation: We use computers and computational methods to study both computers and computation themselves. Mahoney even goes on to suggest that nature itself might be ultimately computational in nature (so to speak). (We will explore that idea in Chapter 9, when we consider whether the universe might be a computer.)

Newell, Perlis, & Simon's reply to the objection is to deny the premise that the phenomenon that a science studies must be natural. They point out that there *are* sciences of artifacts; for example, botanists study hybrid corn.[15] In fact, in 1969, Simon wrote

[13] By the way, 'phenomen**on**' is the correct singular term. If you have two or more of them, you have two or more phenomen**a**.

[14] The annotation '[sic]' (which is Latin for "thus" or "so") is used when an apparent error or odd usage of a word or phrase is to be blamed on the original author and not on the person (in this case, me!) who is quoting the author. For example, here I want to indicate that it is Arden who said "*man*-made", where I would have said "*human*-made".

[15] Curiously, they say that it is *zoologists* who study hybrid corn!

a book called *The Sciences of the Artificial* (Simon, 1996b), and computer scientist Donald Knuth has called CS "an *unnatural* science [because] [c]omputer science deals with artificial things, not bound by the constraints of nature" (Knuth, 2001, p. 167).

The objector might respond that the fact that Simon had to write an entire book to argue that there could be sciences of artifacts shows that the premise—that science only studies natural phenomena—is not *obviously* false. Moreover, botanists study mostly *natural* plants: Hybrid corn is not only not studied by all botanists, it is certainly not the only thing that botanists study (that is, botany is not defined as the science of hybrid corn). Are there any *natural* phenomena that computer scientists study? As I have already hinted, we will see a positive answer to this question in §3.9.3.

But let's not be unfair. There certainly are sciences that study artifacts *in addition* to natural phenomena: Ornithologists study both birds (which are natural) and their nests (which are artifacts); apiologists study both bees (natural) and their hives (artifacts). On the other hand, one might argue (a) that beehives and birds' nests are not *human*-made phenomena, and (b) that 'artifact' should be used to refer, not to any *manufactured* thing (as opposed to living things), but only to things that are manufactured by *humans*, that is, to things that are not "found in nature", so to speak. The obvious objection to this claim is that it unreasonably singles out humans as being apart from nature. (For a commentary on this, see the *Abstruse Goose* cartoon in Figure 3.2.)

Further Reading:
For more on the nature of artifacts in general, see (Dipert, 1993), (Hilpinen, 2011).
On artifacts in CS, see (Mizoguchi and Kitamura, 2009).

3.5.2 Objection: Computers Are Tools, not Phenomena

A related objection has to do with the observation that it is wrong to define a subject by its *tools*: (Fellows and Parberry, 1993) say that CS is not about computers, because "Science is not about tools; it is about how we use them and what we find out when we do". And, as (Hammond, 2003) notes, "Theoretical Computer Science doesn't even use computers, just pencil and paper".

> The debate over the appropriate place of computing in grade schools and high schools echoes the debate in universities decades ago, when computers and software were initially seen as mere plumbing. And certainly not something worthy of study in its own right. A department of computer science? Why not a department of slide rules? (Lohr, 2008)

Newell, Perlis, & Simon also say that astronomy is the science of stars (Newell et al., 1967, p. 1373). And, of course, telescopes are used to study the stars. But, as the computer scientist Edsger W. Dijkstra is alleged to have said, "Computer Science is no more about computers than astronomy is about telescopes."[16] Dikjstra (1987, cited in Tedre & Sutinen 2008) also said that calling the discipline 'computer science' "is like referring to surgery as 'knife science'." This may be true, but the problem, of

[16] https://en.wikiquote.org/wiki/Computer_science#Disputed

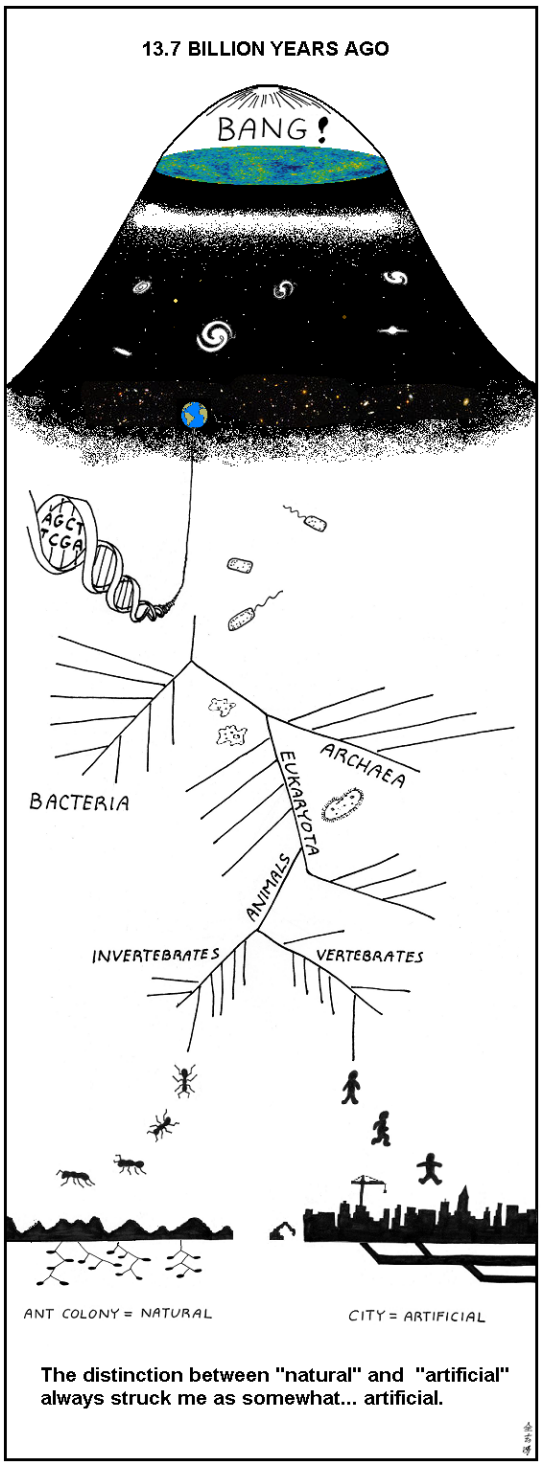


Figure 3.2: http://abstrusegoose.com/215

course, is that the closest term that computer scientists have corresponding to 'surgery' is probably 'computing', and defining 'computer science' as the science of computing may be legitimate but not very clarifying (at least, not without a further description of computing, preferably not in terms of computers!). Newell, Perlis, & Simon address this in their Objection 4: "The computer is such a novel and complex instrument that its behavior is subsumed under no other science" (Newell et al., 1967, p. 1374). (We'll look at this issue in §3.14.1.) This is also a reply to one of the missing premises.

But it is also wrong to define a subject without saying what its tools enable. Even if what Newell, Perlis, & Simon say about the novelty of computers is true, it can be argued that a new tool can open up a new science or, at least, a new scientific paradigm:[17] "Paradigm shifts have often been preceded by 'a technological or conceptual invention that gave us a novel ability to see things that could not be seen before' " (Mertens, 2004, p. 196), quoting (Robertson, 2003). Although CS may not be *about* computers any more than astronomy is *about* telescopes, "The computer is to the naked mind what the telescope is to the naked eye, and it may well be that future generations will consider all precomputer science to be as primitive as pretelescopic astronomy" (Mertens, 2004, p. 196).

But there once *was* a science that only studied a particular artifact, a particular tool—microscopes![18] It is worth a short digression to look at this case.[19]

3.5.3 Digression: The Once-upon-a-Time Science of Microscopy

> ... Marcello Malpighi (1628–1694), was a great scientist whose work had no dogmatic unity.[20] He was one of the first of a new breed of explorers who defined their mission neither by the doctrine of their master nor by the subject that they studied. They were no longer 'Aristotelians' or 'Galenists.' Their eponym, their mechanical godparent, was some device that extended their senses and widened their vistas. What gave his researches coherence was a new instrument. Malpighi was to be a 'microscopist,' and his science was 'microscopy' ... . His scientific career was held together not by what he was trying to confirm or to prove, but by the vehicle which carried him on his voyages of observation.
> (Boorstin, 1983, p. 376)

In a similar fashion, surely computers are "device[s] that [have] extended [our] senses and widened [our] vistas", and the science of computer scientists is, well, *computer* science. After all, one of the two principal professional associations is the Association for Computing *Machinery* (ACM). What "holds" computer scientists "together ... [is] the vehicle which carrie[s them] on [their] voyages of observation".

But this is not necessarily a positive analogy.

---

[17] In the sense of (Kuhn, 1962); we'll discuss Kuhn in §4.9.2.

[18] Another "science" of an artifact might be bicycle science (Wilson and Papadopoulos, 2004). But it's really not clear if this is a science or a branch of engineering.

[19] Thanks to Stuart C. Shapiro for suggesting this.

[20] That is, Malpighi did not study any *single*, natural phenomenon; rather, he studied all phenomena that are only visible with a microscope.

> The applications of computers to a discipline should be considered properly a part of the natural evolution of the discipline. ... The mass spectrometer has permitted significant advances in chemistry, but there is no 'mass spectrometry science' devoted to the study of this instrument. (Loui, 1987, p. 177)

Similarly, the microscope has permitted significant advances in biology (and many other disciplines) but, arguably, *microscopy no longer exists as an independent science devoted to the study of that instrument.*

Now, if you search for 'Department of Microscopy' on the World Wide Web, you will, indeed, find that there are some universities and museums that have one. But, if you look closer, you will see that they are really departments of *microbiology*. Non-biologists who use microscopes (such as some geologists or even jewelers) are not found in departments of microscopy today. What has happened, apparently, is that the use of this artifact by scientists studying widely different phenomena was not sufficient to keep them in the same academic discipline. The academic discipline of microscopy splintered into those who use microscopes to study biology, those who use it to study geology, and so on, as well as those who build new kinds of microscopes (who might be found in an engineering or an optics department).

For over a hundred years, there was a *Quarterly Journal of Microscopical Science* (1853–1965), affiliated with "the Microscopical Society of London". Its inaugural Preface said:

> Recent improvements in the Microscope having rendered that instrument increasingly available for scientific research, and having created a large class of observers who devote themselves to whatever department of science may be investigated by its aid, it has been thought that the time is come when a Journal devoted entirely to objects connected with the use of the Microscope would contribute to the advancement of science, and secure the co-operation of all interested in its various applications.
>
> The object of this Journal will be the diffusion of information relating to all improvements in the construction of the Microscope, and to record the most recent and important researches made by its aid in different departments of science, whether in this country or on the continent. ...
>
> It is, perhaps, hardly necessary to apologise for the title of the Journal, as the term "Microscopical," however objectionable in its origin, has acquired a conventional meaning by its application to Societies having the cultivation of the use of the Microscope in view, and so fully expresses the objects of the Journal, that it immediately occurred as the best understood word to employ. It will undoubtedly be a Journal of Microscopy and Histology; but the first is a term but recently introduced into our language, and the last would give but a contracted view of the objects to which the Journal will be devoted. (Anonymous, 1853a)

If you replace 'microscope' with 'computer' (along with their cognates), and 'histology' with something like 'mathematical calculations' (or 'algorithms'!), then this reads like a manifesto for the ACM.

The first issue of the journal included, besides many articles on what we now call microbiology, a paper on "Hints on the Subject of Collecting Objects for Microscopical

Examination" and a review of a book titled *The Microscopist; or a Complete Manual on the Use of the Microscope*.

Here is a passage from that review:

> As *cutting with a sharp instrument* is better than *tearing with the nails*, so *vision with the microscope* is better than with *the naked eye.* Its use [that is, the *microscope's* use] is, therefore, as extensive as that of the organ which it assists, and it cannot be regarded as the property of one branch of science more than another. (Anonymous, 1853b, p. 52, my italics)

And here is a paraphrase:

> As *vision with the microscope* is better than *with the naked eye*, so *thinking with the computer* is better than with *the mind alone.* Its use [that is, the *computer's* use] is, therefore, as extensive as that of the organ which it assists, and it cannot be regarded as the property of one branch of science more than another.

This is reminiscent of the philosopher Daniel Dennett's arguments for the computer as a "prosthesis" for the mind (Dennett, 1982), that is, as a tool to help us think better.

But, based on the nature of many of their articles, the March 1962 issue of the journal announced a change in focus from microscopy to *cytology*,[21] thus apparently changing their interest from the *tool* to *what can be studied with it.* The change officially occurred in 1966, when the journal changed its name to the *Journal of Cell Science* (and restarted its volume numbers at 1).

Could the same thing happen to *computer* science that happened to *microscope* science? If so, what would fall under the heading of the things that can be studied with computers? A dean who oversaw the Department of Computer Science at my university once predicted that the same thing would happen to our department: The computer-theory researchers would move into the math department; the AI researchers would find homes in psychology, linguistics, or philosophy; those who built new kinds of computers would move (back) into electrical engineering; and so on. This hasn't happened yet (although (McBride, 2007) suggests that it is already happening; (Mander, 2007) disagrees). Nor do I forsee it happening in the near future, if at all. After all, as the computer scientist George Forsythe pointed out, in order to teach "nontechnical students" about computers and computational thinking, and to teach "specialists in other technical fields" about how to use computers as a tool (alongside "mathematics, English, statistics"), and to teach "computer science specialists" about how to "lead the future development of the subject",

> The first major step ... is to create a department of computer science ... Without a department, a university may well acquire a number of computer scientists, but they will be scattered and relatively ineffective in dealing with computer science as a whole. (Forsythe, 1967a, p. 5)

But the break-up of CS into component disciplines is something to ponder.

[21] On the (subtle) "Differences between Histology and Cytology", see http://www.differencebetween.com/differences-between-histology-and-vs-cytology

3.5.4 Objection: Computer Science Is Just a Branch of . . .

The microscopy story is, in fact, close to an objection to one of the missing premises that Newell, Perlis, & Simon consider, that the science of computers is not CS but some other subject: electrical engineering, or math, or, perhaps, psychology.

For example, computer historian Paul Ceruzzi doesn't explicitly say that CS is identical to electrical (more precisely, electronic) engineering, but he comes close (Ceruzzi, 1988, p. 257): First, "Electronics emerged as the 'technology of choice' [over those that were used in mechanical calculators or even early electric-relay-based computers] for implementing the concept of a computing machine . . . . This activity led to the study of 'computing' independently of the technology out of which 'computers' were built. In other words, it led to the creation of a new science: 'Computer Science'." Second, "As computer science matured, it repaid its debt to electronics by offering that engineering discipline a body of theory which served to unify it above the level of the physics of the devices themselves. In short, computer science provided electrical engineering a paradigm, which I call the 'digital approach,' which came to define the daily activities of electrical engineers in circuits and systems design" (Ceruzzi, 1988, p. 258).

One problem with trying to conclude from this that CS is (nothing but) electrical engineering is that there are now other technologies that are beginning to come into use, such as quantum computing and DNA computing. Assuming that those methods achieve some success, then it becomes clear that (and how) CS goes beyond any particular implementation technique or technology, and becomes a more abstract science (or study, or whatever) in its own right. And Ceruzzi himself declares, "The two did not become synonymous" (Ceruzzi, 1988, p. 273).

Further Reading:
On quantum computing, see Brian Hayes 1995, Grover 1999, Aaronson 2008, Monroe and Wineland 2008, Bacon 2010, Bacon and van Dam 2010, Aaronson 2011, Aaronson 2014, and Brian Hayes 2014b. On the relation of quantum computing to "hypercomputation", see the discussion in §11.4.1 and the Further Reading box on p. 447.

On DNA and other forms of biological computing, see Adleman 1998, Shapiro and Benenson 2006, Qian and Winfree 2011, and Lu and Purcell 2016.

Newell, Perlis, & Simon reply that, although CS does intersect electrical engineering, math, psychology, etc., there is no other, *single* discipline that subsumes *all* computer-related phenomena. (This is the missing premise.) This, however, assumes that CS *is* a single discipline, a cohesive whole. Is it? I began my professional university career in a philosophy department; although certain branches of philosophy were not my specialty (ethics and history of philosophy, for instance), I was expected to, and was able to, participate in philosophical discussions on these topics. But my colleagues in CS often do not, nor are expected to, understand the details of those branches of CS that are far removed from their own. As a computer scientist specializing in AI, I have far more in common with colleagues in the philosophy, psychology, and linguistics departments than I do with my computer-science colleagues down the hall who specialize in, say, computer networks or computer security. (And this is not just an autobiographical confession on my part; my colleagues in computer networks and computer security

would be the first to agree that they have more in common with some of their former colleagues in electrical engineering than they do with me.) So, perhaps CS is *not* a coherent whole. (For another take on this, see Question 10 at the end of this chapter.)

3.5.5 Objection: What about Algorithms?

The most interesting—and telling—objection to Newell, Perlis, & Simon's view is that CS is really the study, not (just) of *computers*, but (also) of *algorithms*: very roughly, the programs and rules that tell computers what to do. (We'll devote a great deal of time, beginning with Chapter 7, looking at what algorithms and programs are, so, at this point, I will just assume that you already have an idea of what they are and won't try to define them further.) For example, (Bajcsy et al., 1992, p. 1, my italics) explicitly mention "the (incorrect) assumption that ... [CS] is based *solely* on the study of a *device* ...".

What is interesting about this objection is how Newell, Perlis, & Simon respond: They agree with the objection! They now say:

> In the definition [of CS as the science of computers], *'computers' means ... the hardware, their programs or algorithms, and all that goes along with them.* **Computer science is the study of the phenomena surrounding computers**".
> (Newell et al., 1967, p. 1374, my italics and my boldface)

At the end, they even allow that the study of computers may also be an engineering discipline (Newell et al., 1967, p. 1374). So, they ultimately water down their definition to something like this: *Computer science is the science and engineering of computers, algorithms, and other related phenomena.*

Readers would be forgiven if they objected that the authors have changed their definition! But, instead of making that objection, let's turn to an interestingly different, yet similar, definition due to another celebrated computer scientist.

3.6 CS Studies *Algorithms*

Donald Knuth—Turing-award–winning author of a major, multi-volume work on algorithms (*The Art of Computer Programming* (Knuth, 1973)), as well as developer of the TEX computer typesetting system—gave an apparently different answer to the question of what CS is:

> [C]omputer science is ... **the study of *algorithms***.
> (Knuth, 1974b, p. 323; my boldface, Knuth's italics)

For an interview with Knuth, see Roberts 2018.

3.6.1 Only Algorithms?

He cited, approvingly, a statement by (Forsythe, 1968) that the central question of CS is: What can be automated? Presumably, a process can be automated—that is, done automatically, by a machine, without human intervention—if it can be expressed as an algorithm. (We'll return to this in §3.15.2.1.1.)

Further Reading:
For a book-length discussion of this, see (Arden, 1980).

(Knuth, 1974b, p. 324) even noted that the name 'comput*ing* science' might be better than 'comput*er* science', because the former sounds like the discipline is the science of *computing* (what you *do* with computers) as opposed to the science of *computers* (the *tools* themselves). Others have made similar observations: (Foley, 2002) says that "computing includes computer science" but goes beyond it. (Denning, 2013b, p. 35) says: "I have encountered less skepticism to the claim that 'computing is science' than to 'computer science is science'."

As Knuth pointed out,

> a person does not really understand something until he [sic] teaches it to someone else. Actually a person does not *really* understand something until he can teach it to a *computer*, that is, express it as an algorithm. (Knuth, 1974b, p. 327)

The celebrated cellist Janos Starker once said something similar: "When you have to explain what you are doing, you discover what you are really doing" (Fox, 2013).

Further Reading:
(Schagrin et al., 1985, p. xiii) and (Rapaport and Kibby, 2010, §2.4.2) also discuss this idea.

And expressing something as an algorithm requires "real" understanding, because every step must be spelled out in excruciating detail:

> It is a commonplace that a computer can do anything for which precise and unambiguous instructions can be given. (Mahoney, 2011, p. 80)

That is, a computer can do anything for which an *algorithm* can be given (for, after all, isn't an algorithm merely "precise and unambiguous instructions"?). Thought of this way, the comment is almost trivial. But consider that to give such instructions (to give an algorithm) is to be able to explicitly *teach* the computer (or the executor, more generally) how to do that thing. (For a commentary on this, see the *XKCD* cartoon in Figure 3.3.)

But there is a potential limitation to Knuth's theory that we *teach* computers how to do something—more specifically, to the theory that, insofar as CS is the study of what tasks are *computable*, it is the study of what tasks are *teachable*. The potential limitation is that teaching is "propositional", in the sense that it requires sentences (propositions) of a *language*. Hence, it is *explicit* or conscious. It is what psychologist and Nobel laureate Daniel Kahneman has called a "System 2" task:

Figure 3.3: http://xkcd.com/894/

> *System 2* allocates attention to the effortful mental activities that demand it, including complex computations. The operations of System 2 are often associated with the subjective experience of agency, choice, and concentration. (Kahneman, 2011, p. 21)

But there is another algorithmic way of getting a computer to do something: by *training* it, either via a connectionist, neural-network algorithm, or via a statistical, machine-learning algorithm. 'Learning', in this sense of 'machine learning', is different from being (propositionally) *taught*. Such training is *implicit* or *un*conscious. It is "System 1" thinking:

> *System 1* operates automatically and quickly, with little or no effort and no sense of voluntary control. (Kahneman, 2011, p. 20)

We, as external, third-person observers, don't consciously or explicitly know how to do a System-1 task. Knowing *how* is not necessarily the same as knowing *that*.

Further Reading:
The knowing-how/knowing-that distinction was first discussed in Ryle 1945. Stanley and Williamson 2001 argue that knowing how *is* a form of knowing that; Pavese 2015 takes up their view in the context of computer programs. For a survey of current work on the distinction, see Cath 2019.

The two "systems" or "types" of thinking are discussed in much greater detail in Evans and Stanovich 2013. There, "Type 1" (or "intuitive") processing is characterized as independent of working memory (which is a kind of short-term, conscious memory) and as "autonomous". And "Type 2" (or "reflective") processing is characterized as "requir[ing] working memory" and involving "cognitive decoupling" and "mental simulation". (Cognitive decoupling is, roughly, the ability to mentally represent a mental representation, so that the second-order representation can be thought about separately from the original representation. For more on the two "systems" of thinking, and on unconscious cognition more generally, see http://en.wikipedia.org/wiki/Dual_process_theory#Systems and the bibliography at: http://www.cse.buffalo.edu/∼rapaport/575/rules-connections.html#uncs-cognition. We'll return to these ideas in §12.4.4.1.2.2, when we discuss the difference between following rules and behaving in accordance with them, and again in §18.4, when we discuss whether computers can make decisions.

Here is an example that might help to explain the difference: Consider the game of tic-tac-toe. A computer (or a human player) might be programmed—that is, explicitly "taught"—to play winning tic-tac-toe by using a "conscious" or "System 2" algorithm that it explicitly follows. Most older children and adults have been taught a version of this algorithm (Zobrist, 2000):

For player X to win or draw, **do**:
begin

if there are 2 Xs in a row, **then** make 3 Xs in a row
elsif there are 2 Os in a row, then block with an X
elsif 2 rows intersect with an empty square
such that each row contains 1 X, no Os,
then place X at the intersection
elsif 2 rows intersect with an empty square
such that each row contains 1 O, no Xs,
then place X at the intersection
elsif there is a vacant corner square, **then** put X there
else place X on any vacant square.

end

Alternatively, a computer can be programmed to *learn how* to play winning tic-tac-toe in a "System 1" manner, without expressing (or being able to express) that strategy propositionally, that is, in a "System 2" manner. Such a learning mechanism can be found in (Michie, 1961). Briefly, the computer is "rewarded" for each random move that leads to a win or draw, and such moves are thus caused to be made more frequently in future games.

Further Reading:
There are at least two implementations of Michie's method online:

1. "MENACE: Machine Educable Noughts and Crosses Engine", http://www.mscroggs.co.uk/blog/19
2. "MENACE 2, an Artificial Intelligence Made of Wooden Drawers and Coloured Beads", http://we-make-money-not-art.com/menace-2-an-artificial-intelligence-made-of-wooden-drawers-and-coloured-beads/

An algorithm in the form of a System-1–style artificial neural network is akin to building in to the computer the ability, as if "innate", to do that thing. Such a computer could not necessarily tell us *how* it was doing it; it would not necessarily have any "conscious" access to its algorithm. An algorithm in the form of an explicit machine-learning program that would enable the computer to learn how to do that thing is somewhere in the middle; it would be conscious of its ability to learn, but not necessarily of how to do the thing; it might not necessarily be able to teach someone or something else how to do it, unless it could observe itself doing it and develop a theory of how to do it (which theory would be expressed in a System-2–style, explicit algorithm). (We'll return to these issues in §§3.9.5 and 3.14.4.)

Let's say for now that something is computable just in case "precise and unambiguous instructions can be given" for it. (We'll be more precise and unambiguous(!) in Chapter 7: "What Is an Algorithm?".) So, the question becomes: What tasks are amenable to "precise and unambiguous instructions"? Presumably, chess is computable in this sense, because there are explicit rules for how to play chess. (Playing *winning* chess is a different matter!) But vision would seem *not* to be thus computable. After all, one cannot give "precise and unambiguous instructions" that would enable someone to see. Yet there *are* computer-vision systems (see http://aitopics.org/topic/vision for an overview), so vision *does* seem to be computable in a different sense: A behavior is computable if it can be *described* in terms of such instructions. The entity that exhibits that behavior naturally might not *use*, or be able to use, those instructions in order to behave that way. But we might be able to give those instructions to another system that *could* use them to exhibit that behavior. So, for instance, the human brain might not literally compute in the sense of executing an algorithm in order to see, but a computer using that algorithm might be able to exhibit visual behavior. (Whether it "sees", phenomenologically, is a philosophical question!) Similarly, the solar system might not be executing Kepler's laws, but an artificial solar system might. (We'll look into this issue in §9.8.2.)

3.6.2 Or Computers, Too?

Knuth goes on to point out, however, that you need computers in order to properly study algorithms, because "human beings are not precise enough nor fast enough to carry out any but the simplest procedures" (Knuth, 1974b, p. 323). Indeed, he explicitly copies Newell, Perlis, & Simon's strategy, revising his initial definition to include computers, that is, the phenomena "surrounding" algorithms:

> When I say that computer science is the study of algorithms, I am singling out only one of the "phenomena surrounding computers," so computer science actually includes more. (Knuth, 1974b, p. 324)

Are computers really necessary? If they are, does that mean that CS is (as Newell, Perlis, & Simon claim) the study of computers? Let's consider some similar questions for other disciplines: Do you *need* a compass and straightedge to study geometry (or can you study it just by proving theorems about points, lines, and angles)? After all, the mathematician David Hilbert wrote a completely axiomatic treatment of geometry without any mention of compass or straightedge (Hilbert, 1899). Do you *need* a microscope to study biology? I doubt that Watson and Crick used one when they discovered the structure of DNA. Do you *need* a calculator (or a computer!) to study physics or mathematics (or do they just help you perform calculations more quickly and easily)? Even if you do need these tools, does that make geometry the study of compasses and straightedges, or physics and math the study of calculators, or biology the study of microscopes? I think most people would say that these disciplines are not studies of those tools. On the other hand, "deep learning" algorithms do seem to need computers in order to determine if they will really do what they are intended to do, and do so in real time (Lewis-Kraus, 2016). (We'll return to this in §3.12.)

About ten years later, (Knuth, 1985, pp. 170–171) backed off from the "related phenomena" definition, more emphatically defining CS as "primarily the study of algorithms", because he "think[s] of algorithms as encompassing the whole range of concepts dealing with well-defined processes, including the structure of data that is being acted upon as well as the structure of the sequence of operations being performed", preferring the name 'algorithmics' for the discipline. ((Gal-Ezer and Harel, 1998, p. 80) say that the "heart and basis" of the field is "algorithmics" even though this does not "cover the full scope of CS".) Knuth also suggests that what computer scientists have in common (and that differentiates them from people in other disciplines) is that they are all "algorithmic thinkers" (Knuth, 1985, p. 172). (We will see what it means to "think algorithmically" in §3.14.5, below, and in Chapter 7: "What Is an Algorithm?".

3.7 Physical Computers vs. Abstract Algorithms

So far, it may seem that we have two very different definitions of CS—as *the study of computers* or as *the study of algorithms*. But, just as Newell, Perlis, & Simon said that CS is the study of computers *and related phenomena such as algorithms*, Knuth says that it is the study of algorithms *and related phenomena such as computers*! Stated a bit more bluntly, Newell, Perlis, & Simon's definition comes down to this: Computer science is the science of *computers and algorithms*. Knuth's definition comes down to this: Computer science is the study of *algorithms and computers*.

Ignoring for now the subtle difference between "science" and "study", what we have here are extensionally equivalent, but intensionally distinct, definitions. They may approach the discipline from different viewpoints (one from the viewpoint of a physical tool, one from the viewpoint of an abstract procedure), but the "bottom line" is the same—only the emphasis is different.

Here's a more mundane example of a similar phenomenon: Large supermarkets these days not only carry groceries, but also pharmaceuticals, greeting cards, hardware, etc. Large drugstores these days not only carry pharmaceuticals, but also groceries, greeting cards, hardware, etc. And large "general stores" also carry pretty much the same mix of products. Each kind of store "approaches" merchandising "from different viewpoints": We tend to think of Walgreens as a drugstore, Wegmans as a supermarket, and Walmart as a general store. A philosophical term for this is to say that they are "intensionally distinct". But, because they sell the same mix of products, we can say that they are "extensionally equivalent": Their ("extensional") "bottom line" is the same; only their ("intensional") emphasis is different. (For more on "extensional equivalence", see (Rapaport, 2012a).)

On the other hand, (Arden, 1980, p. 9) claims that "the *study of algorithms* and the *phenomena related to computers* are not coextensive, since there are important organizational, policy, and nondeterministic aspects of computing that do not fit the algorithmic mold". But I don't think that either (Newell et al., 1967) or (Knuth, 1974b) had those things in mind. And if "phenomena related to computers" is taken as widely as Arden does, then it encompasses pretty much everything, thus making any definition based on such a wide notion virtually useless. The classical sciences (physics, chemistry, biology, etc.) also have "important organizational, policy, and nondeterministic aspects", but those aren't used in trying to define what those sciences are about.

So, we now have two (only slightly different) definitions:

1. Computer science is the study of computers (and related phenomena such as the algorithms that they execute).

2. Computer science is the study of algorithms (and related phenomena such as the computers that execute them).

Nearly 50 years ago, (Licklider and Taylor, 1968) fairly accurately predicted what computers would look like, and how they would be used, today. What they were writing about is clearly part of "computer science", yet equally clearly not (directly) part of the abstract, mathematical theory of computation. This strongly suggests that it would be wrong to treat CS as being primarily about algorithms *or* primarily about computers. *It is about both.* We'll see this more clearly in Chapter 6 when we trace the parallel histories of computers (as calculating machines) and computing (as it evolved from the search for a foundation for mathematics).

3.8 CS Studies *Information*

> The mechanical brain[22] does not secrete thought "as the liver does bile," as the earlier materialist claimed,[23] nor does it put it out in the form of energy, as the muscle puts out its activity. *Information is information, not matter or energy.* No materialism which does not admit this can survive at the present day. (Wiener, 1961, p. 132)

Others who have offered definitions of 'computer science' say "A plague on both your houses":[24] CS is *not* the study of computers *or* of algorithms, but of **information**.

For example, Forsythe said:

> I consider computer science, in general, to be the art and science of *representing and processing information* and, in particular, processing information with the logical engines called automatic digital computers.
> (Forsythe, 1967a, p. 3, my italics)

The computer scientist Peter J. Denning defined it as "the body of knowledge dealing with the design, analysis, implementation, efficiency, and application of *processes that transform information*" (Denning, 1985, p. 16, my italics); see also (Denning et al., 1989, p. 16).

Barwise (see §3.4.2, above) said that computers are best thought of as "information processors", rather than as numerical "calculators" or as "devices which traffic in formal strings ... of meaningless symbols" (Barwise, 1989a, pp. 386–387). Barwise's principal reason seems to be that "the ... view of computers as informational engines ... makes sense of the battle for computational resources" and enables us to "think about them so as to make the best decisions about their acquisition and use". And why is that? One reason is that this view enables us to understand the impact of computers along the same lines as we understand the impact of "books and printing [and] ... movable type .... [C]omputers are not just super calculators. They make available a new informational medium ... just as with printing." Although this may seem obvious to us now, Barwise was writing in 1989, way before the general use of the World Wide Web or the advent of Kindles and iPads, and his prediction certainly seems to be coming true.

But why does he say that *information processing* is the key, rather than, say, *symbol manipulation*? Arguably, information processing is nothing but symbol manipulation: After all, information has to be expressed in physical symbols. But symbols can be manipulated independently of their meaning (we'll go into this in more detail in Chapter 22, "Syntax and Semantics"),[25] whereas information processing is *interpreted* symbol manipulation. Moreover, not all symbol manipulation is necessarily information in some sense. So, perhaps, although computers may be nothing but symbol manipulators (this will become clearer when we look at Turing machines, in Chapter 8), it is as information processors that they have an impact.

[22] That is, a computer. —WJR

[23] Or as John Searle has suggested; we will see what he has to say in Ch, 19, §19.6.2.2. —WJR

[24] Shakespeare, *Romeo and Juliet*, Act III, scene 1.

[25] See also (Rapaport, 2017b).

However, Shannon's (1948) theory of information is purely "syntactic"; it is not concerned with the semantic meaning of the information. And Tenenbaum and Augenstein 1981, p. 6, claim that information in a computer has no meaning:

> [I]nformation itself has no meaning. Any meaning can be assigned to a particular bit pattern as long as it is done consistently. It is the interpretation of a bit pattern that gives it meaning.

(We'll return to their view in §14.3.3.)

Similarly, (Bajcsy et al., 1992, p. 1, my italics) say that CS is "a broad-based quantitative and qualitative study of *how information is represented, organized, algorithmically transformed, and used*." Bajcsy et al. also say that "Computer science is the discipline that deals with representation, implementation, manipulation, and communication of information" (Bajcsy et al., 1992, p. 2). I think this second characterization is too broad: Other disciplines (including journalism) also deal with these four aspects of information. But their first definition contains a crucial adverb—'algorithmically'. If that's what makes CS unique, then this just brings us back to algorithms as the object of study.

Indeed, (Hartmanis and Lin, 1992, p. 164) say that "The key intellectual themes in CS&E [computer science and engineering] are algorithmic thinking, the representation of information, and computer programs." But the "representation of information"—although an important branch of CS (in data structures, knowledge representation in AI, and database theory)—is also studied by logicians. And "computer programs"—although clearly another important branch of CS (in software engineering and program verification)—is, arguably, "merely" the implementation of algorithms. So, once again, it is algorithms that come to the fore, not information.

As a final example, (Hartmanis and Lin, 1992, p. 164) define CS this way:

> What is the object of study [of computer science and engineering]? For the physicist, the object of study may be an atom or a star. For the biologist, it may be a cell or a plant. But computer scientists and engineers focus on information, on the ways of representing and processing information, and on the machines and systems that perform these tasks.

Presumably, those who study "the ways of representing and processing" are the scientists, and those who study "the machines and systems" are the engineers. And, of course, it is not just information that is studied; there are the usual "related phenomena": Computer science studies how to *represent* and (algorithmically) *process* information, as well as the *machines* and systems that do this.

Question for the Reader:
Should humans be included among these "machines and systems"? After all, *we* represent and process information, too!

But why constrain the algorithmic processes to be only those that concern "information"? This may seem to be overly narrow: After all, the algorithmic processes that

undoubtedly underlie your use of Facebook on your laptop, tablet, or smartphone may not seem to be related to "information" in any technical sense.

One answer might be found in an earlier (1963) statement by Forsythe (an expression of one of the "Great Insights" of CS that we will look at in §3.15.2.1.2 and in more detail in Chapter 7):

> Machine-held strings of binary digits can simulate a great many *kinds* of things, of which numbers are just one kind. For example, they can simulate automobiles on a freeway, chess pieces, electrons in a box, musical notes, Russian words, patterns on a paper, human cells, colors, electrical circuits, and so on.
> —(Forsythe, quoted in (Knuth, 1972b, p. 722).)[26]

What's common to all of the items on Forsythe's list, encoded as (and thus simulated by) bit strings, is the information contained in them.

Simon takes an interesting position on the importance of computers as information processors (Simon, 1977, p. 1186): He discusses two "revolutions": The first was the Industrial Revolution, which "substitut[ed] ... mechanical energy for the energy of man [sic] and animal". The second was the Information Revolution,, itself consisting of three mini-revolutions, beginning with "written language", then "the printed book", and now the computer. He then points out that "The computer is a device endowed with powers of utmost generality for processing symbols." So, in contrast to what Barwise said, Simon claims that the computer is an information processor *because* information is encoded in symbols.

But here the crucial question is: What is information? The term 'information' as many people use it informally has many meanings: It could refer to Claude Shannon's mathematical theory of information (Shannon, 1948); or to Fred Dretske's or Kenneth Sayre's philosophical theories of information (Dretske, 1981), (Sayre, 1986); or to several others.

But, if 'information' isn't intended to refer to some specific *theory*, then it seems to be merely a vague synonym for 'data' (which is, itself, a vague term!). As the philosopher Michael Rescorla observes, "Lacking clarification [of the term 'information'], the description [of "computation as 'information processing' "] is little more than an empty slogan" (Rescorla, 2015a, §6.1).

Further Reading:
For a survey of various senses of 'information' as it applies to computing, see (Piccinini, 2015, Ch. 14). On the difficulty of defining 'information', see Allen (2017, p. 4239). And on how Shannon's definition differs from the novelist Jane Austen's, see Sloman 2019.

And the philosopher of computer science Gualtiero Piccinini has made the stronger claim that computation is distinct from information processing in *any* sense of 'information'. He argues, for example, that semantic information *requires* representation, but computation does *not*; so, computation is distinct from semantic information processing (Piccinini, 2015, Ch. 14, §3).

[26] For similar observations, see (Shannon, 1953, esp. p. 1235), (Hamming, 1980b, pp. 7–8).

It is important to decide what information is, but that would take us too far afield. As I noted in §1.3, the philosophy of information is really a separate topic from (but closely related to!) the philosophy of computer science.

Question for the Reader:
Are there any kinds of algorithmic processes that manipulate something *other than* information? If there *aren't*, does that make this use of the term 'information' rather meaningless (as simply applying to everything that computers manipulate)? On the other hand, if there *are*, does that mean that defining CS as the study of information is incorrect? (In Chapter 10, we'll look at some algorithms that apparently manipulate something other than information, namely, recipes that manipulate food.)

Further Reading:
Lots of work has been done on the nature of information and its relationship to CS, and on the philosophy of information. See, especially, (Machlup and Mansfield, 1983), (Pylyshyn, 1992), (Denning, 1995), (Floridi, 2002), (Floridi, 2003), (Floridi, 2004b), (Floridi, 2004a), (Dunn, 2008), (Allo, 2010), (Bajcsy, 2010), (Floridi, 2010), (Rosenbloom, 2010), (Scarantino and Piccinini, 2010), (Floridi, 2011), (Gleick, 2011), (Hilbert and López, 2011), (Piccinini and Scarantino, 2011), (Denning and Bell, 2012), (Dunn, 2013), (Primiero, 2016), and (Dennett, 2017, Ch. 6, pp. 105–136, "What Is Information?").

In particular, (Dunn, 2008) is a very readable survey of the nature of information and its role in computer science, covering many of the same topics and issues as this book.

3.9 CS *Is* a Science

> As we enter life, we all struggle to understand the world. Some of us continue this struggle with *dogged* determination. These are the *scientists*. Some of them realize that computation provides a privileged perspective to understand the world outside and the one within. These are the *computer scientists*. (Micali, 2015, p. 52).

As we saw in §3.5, Newell, Perlis, & Simon argue that CS is a *natural* science (of the phenomena surrounding computers). Others agree that it is a science, but with interesting differences.

3.9.1 Computer Science Is a *Formal* (Mathematical) Science

> Turing was born in 1912, and his undergraduate work at Cambridge during 1931–1934 was primarily mathematical. Turing machines were judged as *a mathematical interpretation of computational problem solving*; and computing was interpreted as *an entirely mathematical discipline*.
> —(Wegner, 2010, p. 2, my italics)

> The concept of *computation* is arguably the most dramatic advance in **mathematical thinking** of the past century. —(Frailey, 2010, p. 2, my boldface)

Before we investigate whether CS is a mathematical science, let's ask another question: *Is mathematics even a science at all?* As we saw in §2.6, sometimes a distinction is made between, on the one hand, *experimental* disciplines that investigate the physical world and, on the other, purely *rational* disciplines like mathematics. Let's assume, for the sake of argument, that mathematics is at least a special *kind* of science—a "rational" or "formal" science—and let's consider whether CS might be more like mathematics than like empirical sciences.

(Dijkstra, 1974, p. 608) argues that "programming [i]s a mathematical activity". He doesn't explicitly say that (all) of CS is a branch of mathematics, but it is quite clear that large portions of CS—not only programming—can be considered to be branches of math. As computer scientist Ray Turner puts it:

> That computer science is somehow a mathematical activity was a view held by many of the pioneers of the subject, especially those who were concerned with its foundations. At face value it might mean that the actual activity of programming is a mathematical one. ... We explore the claim that programming languages are (semantically) mathematical theories. —(Turner, 2010, p. 1706)

And the theories of computability and of computational complexity are also clearly mathematical—and no doubt other aspects of CS, too.

Here is Dijkstra's argument for programming to be considered mathematical (Dijkstra, 1974, p. 608):

1. A discipline *D* is a mathematical discipline iff *D*'s assertions are:
   (a) "unusually precise",
   (b) "general in the sense that they are applicable to a large (often infinite) class of instances", and
   (c) capable of being reasoned about "with an unusually high confidence level".
2. Programming satisfies "characteristics" (1a)–(1c).
3. ∴ Programming is a mathematical discipline.

Dijkstra does not argue for premise (1). He takes the "only if" half (mathematical disciplines satisfy (1a)–(1c)) as something that "most of us can agree upon". And he implicitly justifies the "if" half (disciplines that satisfy (1a)–(1c) are mathematical) on the grounds that the objects of mathematical investigation need not be restricted to such usual suspects as sets, numbers, functions, shapes, etc., because what matters is *how* objects are studied, not *what* they are. But the question of what math is (and whether it is a science) are beyond our scope. (For more on the nature of mathematics, see the references cited in §2.8.)

He argues for characteristic (a) of premise 2 (that programming is unusually precise) on the grounds that programming clearly requires extraordinary precision, that programs can accept a wide range of inputs (and thus are general), and that contemporary program-verification techniques are based on logical reasoning. I can't imagine anyone seriously disagreeing with this! And we will look into program-verification

techniques in Chapter 16: "Can Computer Programs Be Verified?", so let's assume that programming satisfies (1a) and (1c) for now.

That leaves characteristic (1b): Are programs really general in the same way that mathematical assertions are? A typical general mathematical assertion might be something like this: *For any triangle*, the sum of its angles is 180 degrees. In other words, the generality of mathematical assertions comes from their being universally quantified ("for any $x \ldots$"). Is that the kind of generality that programs exhibit? A program (as we will see more clearly in Chapter 7) computes a (mathematical) function. So, insofar as mathematical functions are "general", so are programs. Consider a simple mathematical function: $f(x) = 2x$. If we universally quantify this, we get: $\forall x[f(x) = 2x]$. This is general in the same way that our assertion about triangles was. An algorithm for computing f might look like this:[27]

```
Let x be of type integer;
begin
  input(x);
  f := 2 * x;
  output(f)
end.
```

The "preamble", which specifies the type of input, plays the role of the universal quantifier.[28] Thus, the *program* does seem to be general in the same way that a mathematical assertion is. So I think we can agree with Dijkstra about programming being mathematical.

Further Reading:
For more of Dijskstra's observations on mathematics and computer science, see (Dijkstra, 1986). For a brief biography of him, see (Markoff, 2002).

Knuth, on the other hand, is quite clear that he does *not* view CS as a branch of math, or vice versa (Knuth, 1974b, §2), primarily because math allows for *infinite* searches and *infinite* sets, whereas CS presumably does not. But there is no reason in principle why one couldn't write an algorithm to perform such an infinite search. The algorithm would never halt, but that is a *physical* limitation, not a theoretical one.

Mathematician Steven G. Krantz wrote that "Computer scientists, physicists, and engineers frequently do not realize that the technical problems with which they struggle on a daily basis are *mathematics*, pure and simple" (Krantz, 1984, p. 599). As a premise for an argument to the conclusion that CS is nothing but mathematics, this is obviously weak: After all, one could also conclude from it that physics and engineering are nothing but mathematics, a conclusion that I doubt Krantz would accept and that I am certain that no physcist or engineer would accept. (Krantz, 1984) offers a sketch of an argument to the effect that CS is (just) a branch of mathematics. He doesn't really say that; rather, his concern is that CS as an academic discipline is young and unlikely

[27] The notation '$x := y$' means "assign the value y to variable (or storage unit) x".

[28] Technically, it is a "restricted" universal quantifier, because it specifies the type of the variable. See, for example, https://www.encyclopediaofmath.org/index.php/Restricted_quantifier

to last as long as math: "Computer Science did not exist twenty-five years ago [that is, in 1959]; will it exist twenty-five years from now? [That is, in 2009]" (Krantz, 1984, p. 600). Now, as an academic department in a university, CS probably did not exist in 1959, although, arguably, it did two years later (Knuth, 1972b, p. 722); it certainly continued to exist, quite strongly, in 2009. So, if anything, CS is becoming much stronger as an academic discipline, not weaker. (But recall the history of microscopy!)

Let's see if we can strengthen Krantz's premise: Suppose that all of the problems that a discipline $D1$ (such as CS) is concerned with come from discipline $D2$ (such as math). Does it follow that $D1$ is nothing but $D2$? (Does it follow that CS is nothing but math?) Here's an analogy: Suppose that you want to express some literary idea; you could write a story or a poem. Does it follow that prose fiction and poetry are the same thing? Probably not; rather, prose and poetry are two different ways of solving the same problem (in our example, the problem of expressing a certain literary idea). Similarly, even if both CS and mathematics study the same problems, they do so in different ways: Mathematicians prove (declarative) theorems; computer scientists express their solutions algorithmically.

So, perhaps a better contrast between CS and mathematics is that mathematics makes *declarative* assertions, whereas CS is concerned with *procedural* statements. (Loui, 1987, p. 177) makes a similar point (quoting Abelson & Sussman's introductory CS text) in arguing that CS is not mathematics. And Knuth suggests this in (Knuth, 1974a, §3). (We'll come back to this, including the quote from Abelson & Sussman, in §3.14.4.) But is that distinction enough to show that CS is *not* math? After all, Euclidean geometry—which is clearly math—is procedural, not declarative. (We discuss this in further detail in §3.14.4.)

There is yet another way to think about the relationship between math and CS:

> I think it is generally agreed that mathematicians have somewhat different thought processes from physicists, who have somewhat different thought processes from chemists, who have somewhat different thought processes from biologists. Similarly, the respective "mentalities" of lawyers, poets, playwrights, historians, linguists, farmers, and so on, seem to be unique. Each of these groups can probably recognize that other types of people have a different approach to knowledge; and it seems likely that a person gravitates to a particular kind of occupation according to the mode of thought that he or she grew up with, whenever a choice is possible. C.P. Snow wrote a famous book about "two cultures," scientific vs. humanistic, but in fact there seem to be many more than two. —(Knuth, 1985, p. 171)

There is a saying that, to a hammer, everything looks like a nail.[29] This can be taken two ways: as a warning not to look at things from only one point of view, or as an observation to the effect that everyone has their own point of view. I take Knuth's remarks along the latter lines. And, of course, his eventual observation is going to be that computer scientists look at the world algorithmically. Given the wide range of different points of view that he mentions, one conclusion could be that, just as students are encouraged to study many of those subjects, so as to see the world from those

[29] http://en.wikipedia.org/wiki/Law_of_the_instrument

points of view, so we should add algorithmic thinking—computer science—to the mix, because of its unique point of view.

Comparing mathematical thinking to algorithmic thinking, (Knuth, 1985, p. 181) reveals several areas of overlap and two areas that differentiate the latter from the former. The areas of overlap include manipulating formulas, representing reality, problem solving by reduction to simpler problems (a form of recursion, which, as we'll see in later chapters, is at the heart of CS), abstract reasoning, dealing with information structures, and, of course, dealing with algorithms (presumably in a narrower sense). The two areas unique to algorithmic thinking are the "notion of 'complexity' or economy of operation . . . " (presumably, what is studied under the heading of "computational complexity" (Loui, 1996), (Aaronson, 2013b)) and—of most significance—"the dynamic notion of the *state* of a process: 'How did I get here? What is true now? What should happen next if I'm going to get to the end?' Changing states of affairs, or snapshots of a computation, seem to be intimately related to algorithms and algorithmic thinking". But exactly what constitutes "algorithmic thinking" will be discussed in more detail in §3.14.5, below.

Further Reading:
(Rosenbloom, 2010) offers an interesting twist on the relationship of math to CS: Arguing that "computing amounts to a *great scientific domain*, on par with the physical, life, and social sciences", he "subsum[es] mathematics within computing" (p. 2). In other words, instead of CS being a branch of math or being a mathematical science, Rosenbloom sees math as being a branch of CS!

3.9.2 CS Is the Science of *Intellectual* Processes

One of the founders of AI, John McCarthy, said:

> Computation is sure to become one of the most important *of the sciences*. This is because it is *the science of how machines can be made to carry out intellectual processes.* (McCarthy, 1963, p. 1, my italics)

First, note that he thinks that it *is* a science, presumably just like other sciences (else it wouldn't be destined "to become one of the most important of" them). Second, the nature of this science is akin to the view that CS is the study of what is computable: "Machines can be made to carry out intellectual processes" if those processes are computable. Why "*intellectual* processes"? Well, surely mathematical processes are intellectual, and to the extent that other intellectual processes are expressible mathematically—for example, by being codable into symbolic notation (and, ultimately, into binary notation)—those other intellectual processes are mathematical, hence potentially computable. Why is it a science? McCarthy doesn't say, but I would guess that it is a science in the same sense that mathematics is.

3.9.3 CS *Is* a Natural Science (of *Procedures*)

> So does nature compute, and does computation actually predate its invention, or rather discovery, by human beings? If it is the case, then this would actually lend credence to the claim that Computer Science is actually a science and not just and only a branch of engineering. —(Gelenbe, 2011, p. 1)

Then there are those who agree that CS *is* a natural science, but of neither computers, algorithms, *nor* information: Stuart C. Shapiro agrees with Newell, Perlis, & Simon that CS is a science, but he differs on what it is a science of, siding more with Knuth, but not quite:

> Computer Science is a **natural science** that studies **procedures**.
> (Shapiro, 2001, my boldface)

The computational linguist Martin Kay agrees: "[C]omputational linguists ... look to computer science for insight into their problems. If communication is ... about building structures by remote control in the heads of others, then it is all about *process*, and **computer science is the science of process, conceived in its most fundamental and abstract way**" (Kay, 2010, p. 2; italics in original; my boldface).

For Shapiro, CS *is* a *science*, which, like any science, has both theoreticians (who study the limitations on, and kinds of, possible procedures) as well as experimentalists. And, as (Newell and Simon, 1976) suggest in their discussion of empirical results (see §3.9.5, below), there are "fundamental principles" of CS as a science. Newell & Simon cite two: (1) the Physical Symbol System Hypothesis (their theory about the nature of symbols in the context of computers) and (2) Heuristic Search (which is a problem-solving method). Shapiro cites two others: (3) the Church-Turing Computability Thesis to the effect that any algorithm can be expressed as a Turing-machine program and (4) the Böhm-Jacopini Theorem that codifies "structured programming". (We will discuss these in Chapter 7: "What Is an Algorithm?", Chapter 8: "Turing's Analysis of Computation", and Chapter 10: "The Church-Turing Computability Thesis".)

And, although procedures are not natural *objects*, they are measurable natural *phenomena*, in the same way that events are not (natural) "objects" but are (natural) "phenomena". Several people have noted the existence of procedures in nature. Dennett has ...

> ... argued that natural selection is an *algorithmic* process, a collection of sorting algorithms that are themselves *composed* of generate-and-test algorithms that exploit randomness ... in the generation phase, and some sort of mindless quality-control testing phase, with the winners advancing in the tournament by having more offspring.
> —(Dennett, 2017, p. 43) (See also (Dennett, 1995), (Gelenbe, 2011, p. 4).)

And Denning observed that "Computer science ... is the science of information processes and their interactions with the world", adding that "There are many *natural* information processes" (Denning, 2005, p. 27, my emphasis). (Denning, 2007) cites examples of the "discovery" of "information processes in the deep structures of many fields": biology, quantum physics, economics, management science, and even the arts

and humanities, concluding that "computing is *now* a natural science", not (or no longer?) "a science of the artificial". For example, there can be algorithmic (that is, computational) theories or models of biological phenomena such as cells, plants, and evolution.

Further Reading:
On evolution as an algorithm, see Dennett 1995, especially Ch. 1, §§4–5 (and §17.7.2 later in this book).

For more on natural computation, see (Easton, 2006), (Gelenbe, 2011), (Mitchell, 2011), (Denning, 2013b, p. 37), (Pollan, 2013, pp. 104–105), (Covert, 2014), (Gordon, 2016), and (Livnat and Papadimitriou, 2016).

For Shapiro, procedures include, but are not limited to, algorithms. Whereas algorithms are typically considered to be precise, to halt, and to produce correct solutions, the more general notion allows for variations on these themes:

(1) Procedures (as opposed to algorithms) may be imprecise, such as in a recipe. Does *computer* science really study things like recipes? According to Shapiro (personal communication), the answer is 'yes': An education in CS should help you write a better cookbook, because it will help you understand the nature of procedures better!

Further Reading:
However, Peter Denning (2017, p. 38) says, "There is no evidence to support this claim." Sheraton 1981 discusses the difficulties of writing recipes; Moskin 2018 notes that it was the cookbook writer Fannie Farmer who was "the first ... to insist that scientific methods and precise measurements" should be used. We'll return to recipes many times again in this book.

(2) Procedures need not halt: A procedure might go into an infinite loop either by accident or, more importantly, on purpose, as in an operating system or a program that computes the infinite decimal expansion of π.

(3) Nor do they have to produce a correct solution: A chess procedure does not always play optimally. (We will return to these issues in §3.15.2.3, Chapter 7: "What Is an Algorithm?", and Chapter 11: "What Is Hypercomputation?".)

Moreover, Shapiro says that computer science is *not just* concerned with procedures that manipulate abstract information, but also with procedures that are linked to sensors and effectors that allow computers to "sense and operate on the world and objects in it" (p. 3). The philosopher and AI researcher Aaron Sloman makes a similar point when he says that one of the "primary features" of computers (and of brains) is "Coupling to environment via physical transducers" (Sloman, 2002, §5, #F6, pp. 17–18). This allows for "perceptual processes that control or modify actions" and "is how internal information manipulation often leads to external behaviour". (We'll return to this idea when we discuss interactive computation (§11.4.3) and the relation of computers to the world (§17.6.1).)

Procedures are, or could be, carried out in the real world by physical agents, which could be biological, mechanical, electronic, etc. Where do computers come in? According to Shapiro, a computer is simply "a general-purpose procedure-following machine". (But does a computer "follow" a procedure, or merely "execute" it? For some

discussion of this, see (Dennett, 2017, p. 70); we'll come back to this briefly in Chapters 9 and 17.)

So, Shapiro's view seems to be a combination of Knuth's and Newell, Perlis, & Simon's: CS is the natural science of procedures and surrounding phenomena such as computers.

Further Reading:
For another view of computer science as the study of processes, see (Frailey, 2010, esp. p. 4).

3.9.4 CS Is a Natural Science of *the Artificial*

In 1967, Simon joined with Newell and Perlis to argue that CS was the science of (the phenomena surrounding) *computers*. Two years later, in his classic book *The Sciences of the Artificial*, he said that it was a natural science of *the artificial* (Simon, 1996b, 3rd edition, esp. Ch. 1 ("Understanding the Natural and Artificial Worlds"), pp. 1–24).

Here is Simon's argument that CS is a science of the artificial:

1. "A natural science is a body of knowledge about some class of things ... in the world" (Simon, 1996b, p. 1)

   (Presumably, a natural science of X is a body of knowledge about Xs in the world. Note that he does not say that the Xs need to be "natural"! This premise is closely related to Newell, Perlis, & Simon's first premise, which we discussed in §3.5.1.)

2. "The central task of a natural science is ... to show that complexity, correctly viewed, is only a mask for simplicity; to find pattern hidden in apparent chaos" (Simon, 1996b, p. 1)[30]

3. "The world we live in today is much more a[n] ... artificial world than it is a natural world. Almost every element in our environment shows evidence of human artifice" (Simon, 1996b, p. 2).

   (Again, this allows artifacts to be among the values of X. His justification for this premise consists of examples: the use of artificial heating and air-conditioning to maintain temperature, "[a] forest may be a phenomenon of nature; a farm certainly is not. ... A plowed field is no more part of nature than an asphalted street—*and no less*", etc. (Simon, 1996b, pp. 2–3, my emphasis). All artifacts, he also notes, are subject to natural laws (gravity, etc.) (Simon, 1996b, p. 2).)

Now, Simon doesn't, in his first chapter, explicitly draw the conclusion that there can be sciences of artifacts or, more to the point, that CS is an "artificial science" of computers because computers are symbol systems (Simon, 1996b, pp. 17ff) (see also (Newell and Simon, 1976)) and symbols are "strings of artifacts" (Simon, 1996b, p. 2). For one thing, that's what his whole book is designed to argue. But he doesn't have to draw that conclusion explicitly: Even if it doesn't follow from the first premise that CS can be considered a natural science, it does follows from these premises that any

[30] Compare the opening quotation for this chapter by Nakra.

artifacts that impinge upon, or are produced by, nature or natural objects can be studied scientifically (in the manner of the second premise). It's almost as if he really wants to say that artificial sciences are natural sciences.

3.9.5 Computer Science Is an *Empirical Study*

A few years after the first edition of his book, Simon, along with Newell, gave yet another characterization. In a classic paper from 1976, Newell and Simon updated their earlier characterization. Instead of saying that CS is the *science* of computers and algorithms, they now said that it is the "*empirical*" "*study* of the phenomena surrounding computers", "not just the hardware, but *the programmed, living machine*" (Newell and Simon, 1976, pp. 113, 114; my italics).

The reason that they say that CS is not an "experimental" *science* is that it doesn't always strictly follow the scientific (or "experimental") method. (In §4.8, we'll talk more about what that method is. For an opposing view that CS *is* an experimental science, see (Plaice, 1995).) CS is, like experimental sciences, *empirical*—because programs running on computers are *experiments*, though not necessarily like experiments in other experimental sciences. For example, often just *one* experiment will suffice to answer a question in CS, whereas in other sciences, *numerous* experiments have to be run. Another difference between computer "science" and other experimental sciences is that, in CS, the chief objects of study (the computers and the progams) are not "black boxes" (Newell and Simon, 1976, p. 114); that is, most natural phenomena are things whose internal workings we cannot see directly but must infer from experiments we perform on them. But we know exactly how and why computer programs behave as they do (they are "glass boxes", so to speak), because *we* (not nature) designed and built the computers and the programs. We can understand them in a way that we cannot understand more "natural" things.

However, although this is the case for "classical" computer programs, it is not the case for artificial-neural-network programs: "A neural network, however, was a black box" (Lewis-Kraus, 2016, §4) (see the comments about Google Translate in §3.12, below.)

Further Reading:
Rosenblueth and Wiener 1945, pp. 318–319, talk about "closed-box" and "open-box" problems, surely an early version of the notion of "black" and "glass" "boxes". For more on the history of these terms, see https://en.wikipedia.org/wiki/Black_box.

On black boxes, programs as experiments, and their relationship to knowing-how and knowing-that in the context of neural-network algorithms, see Knight 2017; Metz 2017; Mukherjee 2017, 2018:

> Here is the strange rub of such a deep learning system: It learns, but it cannot tell us why it has learned; it assigns probabilities, but it cannot easily express the reasoning behind the assignment. Like a child who learns to ride a bicycle by trial and error and, asked to articulate the rules that enable bicycle riding, simply shrugs her shoulders and sails away, the algorithm looks vacantly at us when we ask, "Why?" It is, like death, another black box. —Mukherjee (2018)

The computer scientist Joseph Weizenbaum (1976, pp. 40–41) considered this to be a fatal flaw:

> Indeed, we are often quite distressed when a repairman returns a machine to us with the words, "I don't know what was wrong with it. I just jiggled it, and now it's working fine." He [sic] has confessed that he failed to come to understand the law of the broken machine and we infer that he cannot now know, and neither can we or anyone, the law of the "repaired" machine. If we depend on that machine, we have become servants of a law we cannot know, hence of a capricious law. And that is the source of our distress.

Recent work in cognitive neuroscience suggests that "recording from neurons at the highest stage of the visual system ... [shows] that there's no black box", and that this might apply to computational neural networks (Wade, 2017).

Neural-network CS has been likened to something other than "real" science, namely, *alchemy*! See a debate on this at https://www.reddit.com/r/MachineLearning/comments/7i1uer/n_yann_lecun_response_to_ali_rahimis_nips_lecture/. For discussion of this, see Fortnow 2018a, which includes the following joke:

> Q: Why did the neural net cross the road?
> A: Who cares as long as it got to the other side.

For discussions of attempts to get such systems to be able to "account" for themselves, see Lipton 2016; Kuang 2017; Metz 2018.

Sometimes, a distinction is made between a *program* and a *process*: A *program* might be a static piece of text or the static way that a computer is hardwired—a textual or physical implementation of an algorithm. A *process* is a dynamic entity—the program in the "process" of actually being executed by the computer.

Further Reading:
We'll look at some of these distinctions in more detail in Chapter 12. On the program-process distinction, see Eden and Turner 2007, §2.2, Denning 2010, p. 4, and Frailey 2010, p. 2. Manovich 2013, p. B11, uses the term 'performance' instead of 'process', "because what we are experiencing is constructed by software in real time. ... we are engaging with the dynamic outputs of computation."

By "programmed, living machines", Newell & Simon meant computers that are actually running programs—*not just* the static machines sitting there waiting for someone to use them, *nor* the static programs just sitting there on a piece of paper waiting for someone to load them into the computer, *nor* the algorithms just sitting there in someone's mind waiting for someone to express them in a programming language—but *"processes" that are actually running on a computer*.

To study "programmed living machines", we certainly do need to study the algorithms that they are executing. After all, we need to know what they are doing; that is, it seems to be necessary to know what algorithm a computer is executing. On the other hand, in order to study an algorithm, it does *not* seem to be *necessary* to have a computer around that can execute it or to study the computer that is running it. It can be helpful and valuable to study the computer and to study the algorithm actually being run on the computer, but the mathematical study of algorithms and their computational complexity doesn't *need* the computer. That is, the algorithm can be studied as a mathematical object, using only mathematical techniques, without necessarily executing it. It may be very much more convenient, and even useful, to have a computer handy, as Knuth notes, but it does not seem to be necessary. If that's so, then it would seem that *algorithms* are really the essential object of study of CS: Both views require algorithms, but only one requires computers.

But is it really the case that you cannot study computers without studying algorithms? Compare the study of computers with neuroscience: the study of brains and the nervous system. Although neuroscience studies both the anatomy of the brain (its static, physical structure) and its physiology (its dynamic activity), it generally treats the brain as a "black box": Its parts are typically named or described, not in terms of what they *do* (their *function*), but in terms of *where they are located* (their *structure*).

Further Reading:
On the function-structure distinction, see Bechtel and Abrahamsen 2005, §3.

For example, the "frontal lobe" is so-called because it is in the *front* of the brain; its *functions* include memory, planning, and motivation. The "temporal lobe" is so-called because it is near the *temples* on your head; its *functions* include processing sensory input. And the "occipital lobe" is so-called because it is near the occipital bone (itself so-called because it is "against" (*ob-*) the head (*caput*)); its *functions* include visual processing.

It is as if a person from the 19th century found what we know to be a laptop computer lying in the desert and tried to figure out what it was, how it worked, and what it did, with no documentation.

Further Reading:
See Weizenbaum 1976, Ch. 5 for the source of this kind of thought experiment. "... Stonehenge, the world's largest undocumented computer" (Brooks, 1975, p. 163) and the Antikythera Mechanism (§6.5.1) are real-life examples.

They might identify certain physical features: a keyboard, a screen, internal wiring (and, if they were from the 19th century, they might describe these as buttons, glass,

and strings), and so on. More likely, they would describe the device as we do the brain, in terms of the *locations* of the parts: an array of button-like objects on the lower half, a glass rectangle on the upper half, and so on.

But without knowledge of what the entire system and each of its parts was supposed to *do*—what their *functions* were—they would be stymied. Yet this seems to be what neuroscientists study. Of course, modern neuroscience, especially modern *cognitive* neuroscience, well understands that it cannot fully understand the brain without understanding its processing (its algorithms, if indeed it executes algorithms) (Dennett, 2017, p. 341). Only recently have new maps of the brain begun to identify its regions *functionally*, that is, in terms of what the regions do, rather than where they are located (Zimmer, 2016). But this is a topic for another branch of philosophy: the philosophy of cognitive science.

Further Reading:
On the philosophy of cognitive science, relevant readings include (Fodor, 1968), (Gazzaniga, 2010), (Piccinini, 2010a), and (Rapaport, 2012b).

So it seems to be necessary to study algorithms in order to fully understand computers.

3.10 CS Is *Engineering*

We have a number of open questions: Insofar as CS studies either algorithms or computers (or both), we need to look further into what, exactly, algorithms are (and how they are related to the more general notion of "procedure"), what kinds of things they manipulate (information? symbols? real-world entities?), what computers are, and how computers and algorithms are related to each other. All of this in due time. (These questions are, after all, the focus of the rest of this book!)

Another question that we still need to explore more closely is whether CS is a science or not. Don't forget, we are calling the subject 'computer *science*' only for convenience; it is not a tautology to say that computer science *is* a science nor is it a self-contradiction to say that computer science is *not* a science. We won't be able to reach a final answer to this question at least until Chapter 4, where we look more closely at what science is.

We have just looked at some reasons for classifying CS as a science—either a natural science, an "empirical" inquiry (a "science of the artificial"), or a formal science (akin to math). An alternative is that CS is *not a science* at all, but a kind of *engineering*. For now, we will assume that engineering is, strictly speaking, something different from science. Again, a final answer to this will have to wait until Chapter 5, where we look more closely at what engineering is.

Frederick P. Brooks, Jr.—another Turing Award winner, perhaps best known as a software engineer—says that CS isn't science because, according to him, it is not concerned with the "discovery of facts and laws" (Brooks, 1996). Rather, he argues, CS is "an engineering discipline": Computer scientists are "toolmakers", "concerned with *making things*": with physical tools such as computers and with abstract tools such as

algorithms, programs, and software systems for others to use. He uses J.R.R. Tolkien's phrase the "gift of subcreation" to describe this concern. CS, he says, is concerned with the usefulness and efficiency of the tools it makes; it is *not*, he says, concerned with newness for its own sake (as scientists are). And the purpose of the tools is to enable us to manage complexity. So, "the discipline we call 'computer science' " is really the "synthetic"—that is, the *engineering*—discipline that is concerned with computers, whereas science is "analytic". (I'll explain this "analytic-synthetic" distinction in a moment.)

Here is Brooks's argument:

1. "[A] science is concerned with the *discovery* of facts and laws." (Brooks, 1996, p. 61, col. 2)

2. "[T]he scientist *builds in order to study*; the engineer *studies in order to build*. (Brooks, 1996, p. 62, col. 1)[31]

3. The purpose of engineering is to *build* things.

4. Computer scientists "are concerned with *making things*, be they computers, algorithms, or software systems". (Brooks, 1996, p. 62, col. 1)

5. ∴ "the discipline we call 'computer science' is in fact not a science but a *synthetic*, an engineering, discipline." (Brooks, 1996, p. 62, col. 1)

The accuracy of the first premise's notion of what science is will be our concern in Chapter 4. By itself, however, Brooks's first premise doesn't *necessarily* rule out CS as a science. First, computer scientists who study the mathematical theory of computation certainly seem to be studying scientific laws. Second, computer scientists like Newell, Simon, and Shapiro have pointed out that Heuristic Search, the Physical Symbol System Hypothesis, the Computability Thesis, or the Böhm-Jacopini theorem certainly seem to be scientific theories, facts, or laws. And "Computer *programming* is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning" (Hoare, 1969, p. 576, my italics). (We'll look into this claim in more detail in Ch. 16: "Can Computer Programs Be Verified?".) So, it certainly seems that at least *part* of CS *is* a science. (We'll return to this in §3.13.) We'll assume the truth of the first premise for the sake of the argument (revisiting it in the next chapter).

The point of the second premise is this: If a scientist's goal is to discover facts and laws—that is, to study rather than to build—then anything built by the scientist is only built for that ultimate purpose. But building is the ultimate goal of engineering, and any studying (or discovery of facts and laws) that an engineer does along the way to building something is merely done for that ultimate purpose. For science, building is a side-effect of studying; for engineering, studying is a side-effect of building. Both scientists and engineers, according to Brooks, build and study, but each focuses more on one than the other. (Does this remind you of the algorithms-vs.-computers dispute

[31] (Petroski, 2008a) argues that all scientists are sometimes engineers and all engineers are sometimes scientists.

earlier?) Kay (see §3.9.3, above) considers computational linguistics to be scientific, whereas natural-language processing is engineering: "scientists try to understand their subject mainly for its own sake, though they are gratified when some of what they do proves useful. Engineers seek only the understanding needed to reach practical ends" (Kay, 2010, p. 1).

The second premise supports the next premise, which Brooks does not explicitly state. It defines engineering as a discipline whose goal is to build things, that is, a "synthetic"—as opposed to an "analytic"—discipline. To *analyze* is to pull apart; to *synthesize* is to put together. "We speak of engineering as concerned with 'synthesis,' while science is concerned with 'analysis' " (Simon, 1996b, p. 4). "Where physical science is commonly regarded as an analytic discipline that aims to find laws that generate or explain observed phenomena, CS is predominantly (though not exclusively) synthetic, in that formalisms and algorithms are created in order to support specific desired behaviors" (Hendler et al., 2008, p. 63). Similarly, (Arden, 1980, pp. 6–7) argues that engineering is concerned with "implementation, rather than understanding", which "is the best distinction" between engineering and science. And implementation is surely on the "building" side of the spectrum (as we'll see in more detail in Chapter 14). Because of multiple implementations of a single theory, questions of efficiency come to the fore in engineering, and "much of computer science is concerned with ... efficiency". But surely computational-complexity theory—the area of CS that is concerned with mathematical analyses of computational efficiency—is on the mathematical or scientific side of the border between science and engineering. Whether or not Brooks's notion of *engineering* is accurate will be our focus in Chapter 5. So, let's assume the truth of the second and third premises for the sake of the argument.

Clearly, if the fourth premise is true, then the conclusion will follow validly (or, at least, it will follow that computer scientists belong on the engineering side of the science–engineering, or studying–building, spectrum). So, is it the case that computer scientists are (only? principally?) concerned with building or "making things"? And, if so, what kind of things?

Interestingly, Brooks seems to suggest that computer scientists *don't* build computers, even if that's what he says in the conclusion of his argument! Here's why: He says that "Even when we build a computer the computer scientist designs only the abstract properties—its architecture and implementation. Electrical, mechanical, and refrigeration engineers design the realization" (Brooks, 1996, p. 62, col. 1). I think this passage is a bit confused,[32] but it makes an interesting point: Brooks seems to be saying that computer scientists only design *abstractions*, whereas other (real?) engineers *implement them in reality.* This is reminiscent of the distinction between the relatively abstract *specifications* for an algorithm (which typically lack detail) and its relatively concrete (and highly detailed) implementation in a computer *program* (we'll look into this in Chapter 10: "The Church-Turing Computability Thesis: What Is a Procedure?"). Brooks (following (Zemanek, 1971)) calls CS "the engineering of abstract objects": If engineering is a discipline that builds, then what CS-considered-as-engineering builds is *implemented abstractions* (see Chapter 14: "What Is Implementation?" for further

[32] You'll understand why I say that when we look into the notion of implementation, in Ch. 14. Briefly, I think the "abstract properties" *are* the design *for* the realization; the electrical (etc.) engineers *build* the realization (they don't *design* it).

discussion).

In 1977, when he first wrote these words (see (Brooks, 1996, p. 61, col. 1), very few people other than scientists, engineers, business people, and a few eduational institutions had access to computing machines (typically, large mainframes or only slightly smaller "minicomputers")—certainly, there were no personal computers (sometimes these used to be called "microcomputers"), or laptops, tablets, or smartphones. So, for Brooks, what computer scientists build, unlike what other engineers build, are not things for direct human benefit but, rather, things that in turn can be used to build such directly beneficial things. Put more simply, his answer to the question "What is a computer?" seems to be: A computer is a tool (and a computer scientist, who makes such tools, is a "toolsmith") (Brooks, 1996, p. 62, col. 1).

But much of what he says *against* CS being considered a *science* smacks of a different battle, one between science and engineering, with scientists belittling engineers. Brooks takes the opposite position: "as we honor the more mathematical, abstract, and 'scientific' parts of our subject more, and the practical parts less, we misdirect young and brilliant minds away from a body of challenging and important problems that are our peculiar domain, depriving the problems of the powerful attacks they deserve" (Brooks, 1996, p. 62, col. 2).

(We'll come back to these issues in Chapter 5, §5.10, question 2.)

3.11 Science *xor* Engineering?

So, is CS a science of some kind (natural or otherwise), or is it not a science at all, but some kind of engineering? The term 'xor' in the title of this section refers to the "exclusive-or" of propositional logic: So, the title of this section means "science or engineering, *but not both*?". Here, we would be wise to listen to two skeptics about the exclusivity of this choice:

> Let's remember that there is only one nature—the division into science and engineering, and subdivision into physics, chemistry, civil and electrical, is a human imposition, not a natural one. Indeed, the division is *a human failure*; it reflects *our limited capacity to comprehend the whole.* That failure impedes our progress; it builds walls just where the most interesting nuggets of knowledge may lie. (Wulf, 1995, p. 56; my italics)

> Debates about whether [CS is] science or engineering can ... be counterproductive, since we clearly are *both, neither, and more* ... . (Freeman, 1995, p. 27, my italics)

3.12 CS as "Both"

> [L]ike electricity, these phenomena [surrounding computers] belong both to engineering and to science. (Knuth, 1974b, p. 324)

> Computer science is both a scientific discipline and an engineering discipline. ... The boundary [between "the division of computer science into theory" (that is, science) "and practice" (that is, engineering)] is a fuzzy one. (Abrahams, 1987, p. 472)

Could CS be *both* science *and* engineering—perhaps the *science* of computation and the *engineering* of computers—that is, the study of the "programmed living machine"?

It certainly makes no sense to have a computer without a program: "A computer without a program is just a box with parts in it" (qFiasco, 2018, p. 38). It doesn't matter whether the program is *hardwired* (in the way that a Turing machine is; see §8.13); that is, it doesn't matter whether the computer is a *special*-purpose machine that can only do one task. Nor does it matter whether the program is a piece of *software* (like a program inscribed on a universal Turing machine's tape; see §8.14); that is, it doesn't matter whether the computer is a *general*-purpose machine that can be loaded with different "apps" allowing the *same* machine to do many *different* things.

Without a program, a computer wouldn't be able to ***do anything****.*

But it also makes very little sense to have a program without a computer to run it on. Yes, you can study the program mathematically; for example, you can try to verify it (see Chapter 16: "Can Computer Programs Be Verified?"), and you can study its computational complexity (Loui, 1996), (Aaronson, 2013b)):

> The ascendancy of logical abstraction over concrete realization has ever since been a guiding principle in computer science, which has kept itself organizationally almost entirely separate from electrical engineering. The reason it has been able to do this is that computation is primarily a logical concept, and only secondarily an engineering one. To compute is to engage in formal reasoning, according to certain formal symbolic rules, and *it makes no logical difference how the formulas are physically represented, or how the logical transformations of them are physically realized.* (Robinson, 1994, p. 12, my italics)

But what good would it be (for that matter, what fun would it be) to have, say, a program for passing the Turing test[33] that never had an opportunity to pass it? Hamming said:

> Without the [computing] machine almost all of what we [computer scientists] do would become idle speculation, hardly different from that of the notorious Scholastics of the Middle Ages. (Hamming, 1968, p. 5)

So, *computers require programs* in order for the *computer* to do anything, and *programs require computers* in order for the *program* to do anything.

This is reminiscent of the 18th-century philosopher Immanuel Kant's slogan that

> Thoughts without content are empty, intuitions without concepts are blind. … The understanding can intuit nothing, the senses can think nothing. Only through their union can knowledge arise. (Kant, 1781, p. 93 (A51/B75))

[33] In Ch. 19, we'll discuss this "test" for whether a computer can be said to be "intelligent".

Philosophical Digression and Further Reading:
In more modern terms, Kant (who lived from 1724 to 1804, about 250 years ago) can be understood as saying that the part of the brain that *thinks* doesn't *sense* the external world (that is, thoughts have to be thoughts *about* something; they have to have "content"), and the part of the brain (or nervous system) that *senses* ("intuits") needs organizing principles ("concepts") in order to *think* about what is sensed. "The understanding" *by itself* doesn't sense the external world; the senses *by themselves* don't think. Only through the "union" of rational thought and empirical sensation "can knowledge arise". This was Kant's way of resolving the opposing views of the nature of knowledge due to the rationalist philosophers (Descartes, Liebiniz, and Spinoza) and the empiricist philosophers (Locke, Berkeley, and Hume). (Recall our discussion in §2.6, of the different kinds of "rationality".) For an informal presentation of some of Kant's ideas, see (Cannon, 2013). For a more philosophically sophisticated introduction, see (Rohlf, 2010) and other articles on Kant in the online *Stanford Encyclopedia of Philosophy* at http://plato.stanford.edu/search/searcher.py?query=kant. For more on what Kant meant by 'intuition', see http://www.askphilosophers.org/question/204.

Similarly, we can say: "Computers without programs are empty; programs without computers are blind. Only through the union of a computer with a program can computational processing arise."[34]

A good example of this is the *need* for computers to test certain "deep learning" algorithms that Google used in their Translate software: Without enough computing power, there was no way to prove that their connectionist programs would work as advertised (Lewis-Kraus, 2016, §2). So, CS must be both a science (that studies algorithms) and an engineering discipline (that builds computers).

But we need not be concerned with the two fighting words 'science' and 'engineering', because, fortunately, there are two very convenient terms that encompass both: 'scientific' and 'STEM'. Surely, not only natural science, but also engineering, not to mention "artificial science", "empirical studies", many of the social sciences, and mathematics are all *scientific* (as opposed, say, to the arts and humanities). And, lately, both the National Science Foundation and the popular press have taken to referring to "STEM" disciplines—science, technology, engineering, and mathematics—precisely in order to have a single term to emphasize their similarities and interdependence, and to avoid having to try to spell out differences among them.[35]

(Cerf, 2012) says, not that CS *is* a science, but that "there really is science to be found *in* computer science" (my emphasis). And, presumably, there is engineering to be found in it, and mathematics, too (and maybe art!). This is a slightly different metaphor from the "spectrum".

So let's agree for the moment that CS might be *both* science *and* engineering. What about Freeman's other two options: *neither* and *more*? Let's begin with "more".

[34] A literary version of 'computers without programs are empty" is the legend of the Golem, a purely material statue that comes to life when certain Hebrew words are inscribed on it. As Ted Chiang's story "Seventy-Two Letters" suggests, the linguistic text can be thought of as the computer program for a robot (Chiang, 2002).

[35] Nothing should be read into the ordering of the terms in the acronym: The original acronym was the less melifluous 'SMET'! And educators, perhaps with a nod to Knuth's views, have been adding the arts, to create 'STEAM' (http://stemtosteam.org/).

3.13 CS as "More"

> Hartmanis calls CS "a new species among the sciences"[36] . . . . [I]t would be more accurate to call computer science a new species of engineering . . .
> (Loui, 1995, p. 31)

Perhaps CS is engineering *together with being something else* (such as a science), or perhaps CS is science plus something else, or that CS can be divided into subdisciplines (such as a science subdiscipline and an engineering subdiscipline). The computer engineer Michael C. Loui takes the first position; the computer scientist Juris Hartmanis takes the second.

3.13.1 CS Is a *New Kind of Engineering*

There are two parts to Loui's argument. Here is the first part:

3.13.1.1 CS Is a *Kind* of Engineering

> The goal of engineering is the production of useful things . . . .
> [C]omputer science is concerned with producing useful things . . . .
> Computer science is therefore a . . . kind of engineering.
> (Loui, 1995, p. 31)

Unfortunately, *this is invalid!* Just because two things share a common property (in this case, the property of producing useful things), it does not follow that one is subsumed under the other. For example, just because both cats and dogs are household pets, it doesn't follow that cats are a kind of dog (or that dogs are a kind of cat). Loui could equally well have concluded that engineering was a kind of "computer science"!

Fortunately, there are better arguments for considering CS to be a branch of engineering, as we just saw in our discussion of Brooks (and will look at more closely in Chapter 5: "What Is Engineering?"). In fact, Loui himself gave another argument in an earlier article! There, (Loui, 1987, p. 175) argued (against (Krantz, 1984)) that CS *is* a legitimate academic discipline (not *another* discipline, such as math, nor something that will disappear or dissolve into other disciplines, like microscopy). And he argues (against people like Newell, Perlis, & Simon) that it is not the study of "the use of computers" (Loui, 1987, p. 175). But that word 'use' is important. What Loui has in mind certainly includes the study of hardware (Loui, 1987, p. 176); what he is rejecting is that CS, as an academic discipline, is the study of *how to use* computers (in the way that driving schools teach how to drive cars).

His definition is this:

> Computer science is the theory, design, and analysis of algorithms for processing [that is, for storing, transforming, retrieving, and transmitting] information, and the implementations of these algorithms in hardware and in software.
> (Loui, 1987, p. 176)

[36] (Hartmanis, 1995a), quoted in §3.3.2.2, above; but see also (Hartmanis, 1993, p. 1). We will discuss Hartmanis in §3.13.2, below.

He then goes on to argue that CS is an engineering discipline (Loui, 1987, p. 176), because engineering ...

1. ... is concerned with what *can* exist (as opposed to what *does* exist),
2. "has a scientific basis",
3. is concerned with "design",
4. analyzes "trade-offs", and
5. has "heuristics and techniques" .

"Computer science has all the significant attributes of engineering"; therefore, CS is a branch of engineering.

Let's consider each of these "significant attributes": First, his justification that CS is *not* "concerned with ... what *does* exist" is related to the claim that CS is not a natural science, but a science of human-made artifacts. We have already considered two possible objections to this: First, insofar as procedures are natural entities, CS—as the study of procedures—*can* be considered a natural science. Second, insofar as some artifacts—such as bird's nests, beehives, etc.—are natural entities, studies of artifacts can be considered to be natural science.

Next, he says that the "scientific basis" of CS is mathematics. Compare this with the scientific basis of "traditional engineering disciplines such as mechanical engineering and electrical engineering", which is physics. (We'll come back to this in §3.13.1.2.)

As for design, Forsythe said that

> a central theme of computer science is analogous to a central theme of engineering science—namely, the *design* of complex systems to optimize the value of resources. (Forsythe, 1967a, p. 3, col. 2).

According to Loui, engineers apply the principles of the scientific base of their engineering discipline to "design" a product: "[A] computer specialist applies the principles of computation to design a digital system or a program" (Loui, 1987, p. 176). But not all computer "specialists" design systems or programs; some do purely theoretical work. And if the scientific basis of CS is mathematics, then why does Loui say that computer "specialists" apply "the principles of *computation*"? Shouldn't he have said that they apply the principles of *mathematics*? Perhaps he sees "computation" as being a branch of mathematics (but that's inconsistent with his objections to Krantz; recall our discussion of CS as math, in §3.9.1.) Or perhaps he doesn't think that the abstract mathematical theory of computation is part of CS. However, that seems highly unlikely, especially in view of his definition of CS as including the theory and analysis of algorithms. It's almost as if he sees computer *engineering* as standing to computer *science* in the same way that mechanical or electrical engineering stand to physics. But then it is not computer *science* that is a branch of engineering.

Let's turn briefly to trade-offs: "To implement algorithms efficiently, the designer of a computer system must continually evaluate trade-offs between resources" such as time vs. space, etc. (Loui, 1987, p. 177). This is true, but doesn't support his argument as well as it might. For one thing, it is not only system designers who evaluate such

trade-offs; so do theoreticians—witness the abstract mathematical theory of complexity. And, as noted above, not all computer scientists design such systems. So, at most, it is only those who do who are doing a kind of engineering.

Finally, consider heuristics. There are at least two different notions of "heuristics": as rough-and-ready "rules of thumb" and as formally precise theories. Loui seems to have the first kind in mind. (We'll look at the second kind in §3.15.2.3, below.) Insofar as engineers rely on such heuristics (see Chapter 5's discussion of Koen's definition of 'engineering'), and insofar as some computer scientists also rely on them, then those computer scientists are doing something that engineers also do. But so do many other people: Writers surely rely on such rule-of-thumb heuristics ("write simply and clearly"); does that make them engineers?[37] This is probably his weakest premise.

3.13.1.2 CS Is a *New* Kind of Engineering

The second part of Loui's argument is to show how CS is a "*new*" kind of engineering. Here is his argument for this (Loui, 1995, p. 31):

1. "[E]ngineering disciplines have a scientific basis".
2. "The scientific fundamentals of computer science ... are rooted ... in mathematics."
3. "Computer science is therefore a *new* kind of engineering." (italics added)

This argument can be made valid by adding two missing premises:

A. Mathematics is a branch of science.

B. No other branch of engineering has mathematics as its basis.

We can assume from the first part of his argument that CS is a *kind* of engineering. So, from that and premise (1), we can infer that CS (as an engineering discipline) must have a scientific basis. We need premise (A) so that we can infer that the basis of CS (which, by premise (2), is mathematics) is indeed a scientific one. Then, from premise (B), we can infer that CS must differ from all other branches of engineering. It is, thus, mathematical engineering.

(Abrahams, 1987, p. 472) also explicitly makes this claim. And (Halpern et al., 2001) can be read as making a case for CS as being based more on *logic* than on mathematics, so—if it is a kind of engineering—perhaps it is *logical* engineering? This assumes, of course, that you are willing to consider mathematics (or logic) to be a natural science, or else that science is not limited to studying natural entities. But in the latter case, why worry about whether CS is concerned with what *can*, rather that what *does*, exist? (We'll return to CS as mathematical engineering in §3.13.2, below.)

Towards the end of his essay, Loui says this: "It is impossible to define a reasonable boundary between the disciplines of computer science and computer engineering. They are the same discipline" (Loui, 1987, p. 178). But does't that contradict the title of his essay ("Computer Science Is an Engineering Discipline")?

[37] However, see (Carey, 2010) for an argument to the effect that learning how to write computer programs can help one become a better writer!

3.13.2 CS Is a New Kind of *Science*

> [C]omputer science differs from the known sciences so deeply that it has to be viewed as a new species among the sciences. (Hartmanis, 1993, p. 1).

Hartmanis comes down on the side of CS being a science: It is a "new species *among the sciences*".

What does it mean to be a "new species"? Consider biological species. Roughly speaking, different species are members of the same genus; different genera[38] are grouped into "families", families into "orders", orders into "classes", and classes into "kingdoms". Now consider three different species of the animal kingdom: chimpanzees, lions, and tigers. Lions and tigers are both species within the genus Panthera, that genus is in the order of carnivores, and carnivores are in the class of mammals. Chimps, on the other hand, are in the order of primates (not carivores), but they are also in the class of mammals. So, lions and tigers are more closely related to each other than either is to chimps, but all three are mammals.

But what does it mean to be "a new species" of science? Is the relation of CS to other sciences more like the relation of chimps to tigers (relatively distant, only sharing in being mammals) or lions to tigers (relatively close, sharing in being in Panthera)? A clue comes in Hartmanis's next sentence:

> This view is justified by observing that theory and experiments in computer science play a different role and do not follow the classic pattern in physical sciences. (Hartmanis, 1993, p. 1)

This strongly suggests that CS is not a *physical* science (such as physics or biology), and Hartmanis confirms this suggestion on p. 5: "computer science, *though not a physical science*, is indeed a science" (my italics; see also (Hartmanis, 1993, p. 6), (Hartmanis, 1995a, p. 11)). The non-physical sciences are typically taken to include both social sciences (such as psychology) and formal sciences (such as mathematics). So, it would seem that the relation of CS to other sciences is more like that of chimps to tigers: distantly related species of the same, high-level class. And, moreover, it would seem to put CS either in the same camp as (either) the social sciences or mathematics, or else in a brand-new camp of its own, that is, *sui generis*.

Hartmanis said that he would not define CS (see the epigraph to §3.3.3, above). But immediately after saying that, he seems to offer a definition:

> At the same time, it is clear that *the objects of study in computer science are information and the machines and systems which process and transmit information.* From this alone, we can see that computer science is concerned with the abstract subject of information, which gains reality only when it has a physical representation, and the man-made devices which process the representations of information. The goal of computer science is to endow these information processing devices with as much intelligent behavior as possible.
> (Hartmanis, 1993, p. 5, my italics) (see also (Hartmanis, 1995a, p. 10))

[38] That's the plural of 'genus'.

Although it may be "clear" to Hartmanis that information, an "abstract subject", is (one of) the "objects of study in computer science", he does not share his reasons for that clarity. Since, as we have seen, others seem to disagree that CS is the study of information (others have said that it is the study of computers or the study of algorithms, for instance), it seems a bit unfair for Hartmanis not to defend his view. But he cashes out this promissory note in (Hartmanis, 1995a, p. 10, my italics), where he says that "what sets it [that is, CS] apart from the other sciences" is that it studies "processes [such as information processing] that are *not directly governed by physical laws*". And why are they not so governed? Because "information and its transmission" are "abstract entities" (Hartmanis, 1995a, p. 8). This makes CS sound very much like mathematics. That is not unreasonable, given that it was this aspect of CS that led Hartmanis to his ground-breaking work on computational complexity, an almost purely mathematical area of CS.

But it's not just information that is the object of study; it's also information-processing *machines*, that is, computers. Computers, however, don't deal directly with information, because information is abstract, that is, non-physical. For one thing, this suggests that, insofar as CS is a new species of *non-physical* science, it is not a species of *social* science: Despite its name, the "social" sciences deal with pretty physical things: societies, people, speech, etc.

(This, by the way, is controversial. After all, one of the main problems of philosophy is the problem of the relation of the mind to the brain. The former seems to be non-physical, and is studied by the social science of psychology. The latter is clearly physical, and is studied by the physical sciences of biology and neuroscience. And philosophers such as (Searle, 1995) have investigated the metaphysical nature of social institutions such as money, which seem to be neither purely abstract (many people cash a real weekly paycheck and get real money) nor purely natural or physical (money wouldn't exist if people didn't exist).)

So, if CS is a science, but is neither physical nor social, then perhaps it is a "formal" science like mathematics. (We investigated this in §3.9.1.)

Terminological Digression and Further Reading:
This is as good a place as any to discuss the meaning of the word 'formal', as it appears in phrases like 'formal logic' or 'formal science'. In this use, it is *not* synonymous with words like 'prim' or 'methodical', and it has nothing directly to do with concepts like "a formal dinner party". Rather, it relates to "form", "shape", or "structure". (See https://www.merriam-webster.com/dictionary/formal).

On "formal" sciences in general, see http://en.wikipedia.org/wiki/Formal_science.

For another thing, to say that computers don't deal directly with information, but only with *representations* of information suggests that CS has a split personality: Part of it deals directly with something *abstract* (information), and part of it deals directly with something *real* but that is (merely?) a representation of that abstraction (hence dealing *indirectly* with information). Such real (physical?) representations are called "implementations"; we will look at that notion in more detail in Chapter 14, and we will look at the relation of computers to the real world in Chapter 17.

Finally, although Hartmanis's description of the goal of CS—"to endow ... [computers] with ... intelligent behavior"—sounds like he is talking about AI (and he might very well be; see §3.14.6, below), another way to think about that goal is this: Historically, we have "endowed" calculating machines with the ability to do both simple and complex mathematics. What other abilities can we give to such machines? Or, phrased a bit differently, what can be automated—what can be computed? (Recall §3.6.1, above, and see §3.15, below.)

Here is another reason why Hartmanis thinks that CS is not a physical science and probably also why it is not a social science:

> [C]omputer science is concentrating more on the *how* than the *what*, which is more the focal point of physical sciences. In general the *how* is associated with engineering, but computer science is **not** a subfield of engineering. (Hartmanis, 1993, p. 8; Hartmanis's italics, my boldface)

But there are two ways to understand "how": Algorithms are the prime formal entities that codify *how* to accomplish some goal. But, as Hartmanis quickly notes, engineering is the prime discipline that is concerned with how to do things, how to build things. The first kind of "how" is mathematical and abstract (indeed, it is *computational*!—see §§3.14.4 and 3.14.5); the second is more physical. One way to see this as being consistent with Hartmanis's description of the objects of study of CS is to say that, insofar as CS studies abstract information, it is concerned with how to process information (that is, it is concerned with algorithms), and, insofar as CS studies computers, it is concerned with how to process *representations* (or *implementations*) of information (that is, it is concerned with the physical devices).

But that latter task would seem to be part of engineering (perhaps, historically, electrical engineering; perhaps, in the future, quantum-mechanical or bioinformatic engineering; certainly computer engineering!). So why does he say that "computer science is *not* a subfield of engineering"? In fact, he seems to regret this strong statement, for he next says that "the engineering in our field has different characterizations than the more classical practice of engineering" (Hartmanis, 1993, p. 8): So, CS certainly *overlaps* engineering, but, just as he claims that CS is a new species of science, he also claims that "it is a new form of engineering" (Hartmanis, 1993, p. 9). In fact, he calls it "[s]omewhat facetiously ... the engineering of mathematics" (recall our discussion of Loui, in §3.13.1.2); however, he also says that "we should not try to draw a sharp line between computer science and engineering" (Hartmanis, 1993, p. 9).

To sum up so far, Hartmanis views CS as a new species both of science and of engineering. This is due, in part, to his view that it has two different objects of study: an abstraction (namely, information) as well as its implementations (that is, the physical representations of information, typically in strings of symbols). But isn't it also reasonable to think that, perhaps, there are really two different (albeit new) disciplines, namely, a new kind of science *and* a new kind of engineering? If so, do they interact in some way more deeply and integratively than, say, chemistry and chemical engineering, so that it makes sense to say that "they" are really a single discipline?

Hartmanis suggests two examples that show a two-way interaction between these two disciplines (or two halves of one discipline?): Alan Turing's interest in the *mathematical* nature of computation led to his development of *real* computers; and John

von Neumann's interest in *building* computers led to his *theoretical* development of the structure of computer architecture (Hartmanis, 1993, p. 10). The computational logician J. Alan Robinson made similar observations:

> Turing and von Neumann not only played leading roles in the design and construction of the first working computers, but were also largely responsible for laying out the general logical foundations for understanding the computation process, for developing computing formalisms, and for initiating the methodology of programming: in short, for founding computer science as we now know it. ...
>
> Of course no one should underestimate the enormous importance of the role of engineering in the history of the computer. Turing and von Neumann did not. They themselves had a deep and quite expert interest in the very engineering details from which they were abstracting, but they knew that the logical role of computer science is best played in a separate theater. (Robinson, 1994, pp. 5, 12)

Hartmanis explicitly says that CS *is* a science and is *not* engineering, but his comments imply that it is both. I don't think he can have it both ways. Both Loui and Hartmanis agree that CS is a new kind of something or other; each claims that the scientific and mathematical aspects of it are central; and each claims that the engineering and machinery aspects of it are also central. But one *calls* it 'science', while the other *calls* it 'engineering'. This is reminiscent of the dialogue between Newell, Perlis, & Simon on the one hand, and Knuth on the other. Again, it seems to be a matter of point of view.

A very similar argument (that does not give credit to Hartmanis!) that CS is a new kind of *science* can be found in (Denning and Rosenbloom, 2009). We'll look at some of what they have to say in §3.14.1.

3.14 CS as "Neither"

In this section, we will look at claims of CS as having a unique paradigm (being truly *sui generis* and not just a new kind of science or engineering), as **art**, as the study of **complexity**, as **philosophy**, as a **way of thinking**, as **AI**, and as **magic**(!).

3.14.1 CS Has Its Own Paradigm

We just saw that Hartmanis argued that CS was *sui generis* among the sciences (§3.13.2) and that Loui argued that CS was *sui generis* within engineering (§3.13.1.2). Denning & Peter A. Freeman offer a slightly stronger argument to the effect that CS is neither science, nor engineering, nor math; rather, CS has a "unique paradigm" (Denning and Freeman, 2009, p. 28).

But their position is somewhat muddied by their claim that "computing is a fourth great domain *of science* alongside the physical, life, and social sciences" (Denning and Freeman, 2009, p. 29, my italics). That implies that CS *is* a science, though of a different kind, as Hartmanis suggested.

It also leaves mathematics out of science! In a related article published three months earlier in the same journal, Denning & Paul S. Rosenbloom assert without

argument that "mathematics ... has traditionally not been considered a science" (Denning and Rosenbloom, 2009, p. 28) (see also (Rosenbloom, 2010)). Denying that math is a science allows them to avoid considering CS as a *mathematical* science (an option that we explored in §3.9.1).

To justify their conclusion that CS is truly *sui generis*, Denning & Freeman need to show that it is not a physical, life, or social science. Denning & *Rosenbloom* say that "none [of these] studies computation per se" (Denning and Rosenbloom, 2009, p. 28). This is only half of what needs to be shown; it also needs to be shown that CS doesn't study physical, biological, or social entities. Obviously, it does study such things, though that is not its focus. As they admit, CS is "used extensively in all the domains" (Denning and Rosenbloom, 2009, p. 28); that is, computation is used by scientists in these domains *as a tool.*

So, what makes CS different? Denning & *Freeman* give a partial answer:

> The central focus of the computing paradigm can be summarized as information processes—natural or constructed processes that transform information. ... [T]he computing paradigm ... is distinctively different because of its central focus on information processes. (Denning and Freeman, 2009, pp. 29–30)

This is only a partial answer, because it only discusses the *object of study* (which, as we saw in §3.8, is either vague or multiply ambiguous).

The rest of their answer is provided in a table showing the methodology of CS (Denning and Freeman, 2009, p. 29, Table 2), which comes down to their version of "computational thinking" (Denning and Freeman, 2009, p. 30, col. 3): "The computing paradigm", they say, begins by "determin[ing] if the system ... can be represented by information processes", then "design[s] a computational model", "implement[s the] designed processes", "test[s] the implementation", and finally "put[s] the results to action in the world". (We'll explore "computational thinking" further in §3.14.5. Denning & Freeman's version of it is close to what I will present as "synthetic" computational thinking in §3.15.2.1.2.)

3.14.2 CS Is *Art*

Recall from §3.8 that Forsythe said that CS was "the *art* and science of representing and processing information ... with ... computers" (Forsythe, 1967a, p. 3, my italics). Why might he have said that CS is an "art" (in addition to being a science)? Recall something else that he said: "strings of binary digits can simulate ... numbers ... automobiles ... , chess pieces, electrons ... , musical notes, ... words, patterns on paper, human cells, colors, electrical circuits, and so on" (cited in (Knuth, 1972b, p. 722)). Perhaps because some of these things are not "scientific", then, if CS is going to study them, then CS must be an "art". After all, 'art' is often opposed to 'science'.

Knuth defends his use of the word 'art' in the title of his multi-volume classic *The Art of Computer Programming* (Knuth, 1973) not by saying that all of CS is an art, but that 'art' can be applied to, at least, computer programming. The application is not in opposition to 'science', but alongside it. He gives a useful survey of the history of the term 'art': According to (Knuth, 1974a, p. 668, col. 1), 'art' in the non-painting sense

once "meant something devised by man's intellect, as opposed to activities derived from nature or instinct", as in the "liberal arts", and it later came to be ...

> ... used ... for the *application* of knowledge [where 'science' was "used to stand for knowledge"]. ... The situation was almost exactly like the way in which we now distinguish between "science" and "engineering."
> (Knuth, 1974a, p. 668, col. 2, my italics)

Today, when one thinks of the "liberal arts", one tends to think of the humanities rather than the sciences, but, classically, there were seven liberal arts: the linguistic liberal arts (grammar, rhetoric, and logic) and the mathematical liberal arts (arithmetic, music, geometry, and astronomy). The phrase originally referred to "those subjects ... considered essential for a free person (*liberalis*, 'worthy of a free person')" (https://en.wikipedia.org/wiki/Liberal_arts_education). Thought of this way, it becomes more reasonable to consider CS as a modern version of these.

Further Reading:
On the liberal arts in general, see the *Wikipedia* article "Liberal Arts Education", https://en.wikipedia.org/wiki/Liberal_arts_education. On CS as a liberal art, see (Lindell, 2001).

(Perlis, 1962, p. 210) agrees:

> I personally feel that the ability to analyze and construct processes is a very important ability, one which the student has to acquire sooner or later. I believe that he [sic] does acquire it in a rather diluted way during four years of an engineering or science program. I consider it also important to a liberal arts program.

Indeed,

> Pedagogically, computer programming has the same relation to studying CS as playing an instrument does to studying music or painting does to studying art. (Tucker et al., 2003, p. V)

Knuth has a more interesting suggestion about art:

> Science is knowledge which we understand so well that we can teach it to a computer; and *if we don't fully understand something, it is an art to deal with it.* Since the notion of an algorithm or a computer program provides us with an extremely useful test for the depth of our knowledge about any given subject, the process of going from an art to a science means that we learn how to automate something. (Knuth, 1974a, p. 668, col. 2, my italics)

In (Knuth, 2001, p. 168), he adds this: "Every time science advances, part of an art becomes a science, so art loses a little bit. Yet, mysteriously, art always seems to register a net gain, because as we understand more we invent more new things that we can't explain to computers." (We saw a similar comment, by Wheeler, in connection with our discussion of philosophy in §2.5.1.)

This suggests that being an art is a possibly temporary stage in the development of our understanding of something, and that our understanding of computer programming is (or at least was at the time tht Knuth was writing) not yet fully scientific. If some subject is never fully scientifically understood, then what is left over remains an art:

> Artificial intelligence has been making significant progress, yet there is a huge gap between what computers can do in the foreseeable future and what ordinary people can do. ... [N]early everthing we do is still an art. (Knuth, 1974a, pp. 668–669).

But he then goes on to give a slightly different meaning to 'art', one more in line with the sense it has in 'the fine arts', namely, as "an art *form*, in an aesthetic sense" (Knuth, 1974a, p. 670). In this sense, programming can be "beautiful" and "it can be like composing poetry or music" (Knuth, 1974a, p. 670). This is not inconsistent with programming being considered a science or a branch of mathematics. Indeed, many scientists and mathematicians speak of theories, theorems, demonstrations, or proofs as being "beautiful". One can, presumably, scientifically (or mathematically) construct a logically verifible program that is ugly (for example, difficult to read or understand) as well as one that is beautiful (for example, a pleasure to read or easy to understand); Knuth himself has advocated this under the rubric "literate programming" (Knuth, 1984). More recently, (Hill, 2017b) has suggested various criteria that make programs "elegant".

So, CS can certainly be considered to have interesting relationships to "art" in all of that term's senses. But it is surely not *just* an art.

Further Reading:
For more discussion of CS as art, see Soare 2016, pp. xvii–xviii, which distinguishes between art as a study of beauty and art as a craft, but argues that CS should be viewed as an art in both senses; and Decker et al. 2017, which treats "computing education as an artistic practice" and "the act of programming ... [as] a form of creative expression". See also Dennett 2017, pp. 81–82 for similar observations. On beauty in science more generally, see Quine 1987, "Beauty", pp. 17–18.

3.14.3 CS Is the Study of *Complexity*

> [T]he art of programming is the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.
> —Edsger W. Dijkstra (1972, p. 6)

> Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one, a subroutine .... In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound. ... The complexity of software is an essential property, not an accidental one. —Frederick P. Brooks (1995, pp. 182–183)[39]

It has been suggested that CS is the study of *complexity*—not just the mathematical subject of "computational complexity", which is really more a study of *efficiency*—but complexity in general and in all of nature. (Ceruzzi, 1988, pp. 268–270) ascribes this to the electrical engineer and MIT president Jerome Wiesner (Wiesner, 1958). But all

[39] On essential vs. accidental properites, see §§2.8 and 9.5.4.

Wiesner says is that "Information processing systems are but one facet of ... communication sciences ... that is, the study of ... the problems of organized complexity' " (quoted in (Ceruzzi, 1988, p. 269)). But even if computer science is part of a larger discipline ("communication sciences"?) that studies complexity, it doesn't follow that CS itself *is* the study of complexity.

According to Ceruzzi, Edsgar Dijkstra also held this view: "programming, when stripped of all its circumstantial irrelevancies, boils down to no more and no less than very effective thinking so as to avoid unmastered complexity" (Dijkstra, 1975a, §4, p. 3). (We'll look in more detail at the nature of this kind of thinking in §3.14.5, below.) What's missing from Dijkstra's argument, in any case, is a premise to the effect that computer science is the study of programming, but Dijkstra doesn't say that—not in (Dijkstra, 1975a) nor in (Dijkstra, 1976), the document that Ceruzzi says contains that premise. ((Khalil and Levy, 1978), however, do make that claim.)

Programming helps "avoid complexity" because "Computer science offers a standard way to handle complexity: hierarchical structure" (Lamport, 2012, p. 16). That is,

> [P]rograms are built from programs. ... Programs are compilations in another sense as well. Even the smallest sub-program is also a compilation of sub-components. Programmers construct sub-programs by assembling into a coherent whole such discrete program elements as data, data structures, and algorithms. The "engineering" in software engineering involves knowing how to assemble these components to produce the desired behavior.
> (Samuelson et al., 1994, pp. 2326–2327)

This is the idea that a complex program is "just" a construction from simpler things, each of which—"recursively"—can be analyzed down to the primitive operations and data structures of one's programming system (for a Turing machine, these would include the operations of printing and moving, and data structures constructed from '0's and '1's). It is the underlying way in which complexity can be dealt with. It is also where engineering (considered as a form of construction) comes into the picture.

Digression on Recursion:
Briefly, a recursive function is one that is defined in terms of "base" cases that give explicit values for some inputs to the function and then "recursive" cases that give values for other inputs in terms of *previous* values (that is, in terms of—or constructed from—previous *outputs*). We'll go into further detail in Ch. 7.

But, again, at most this makes the claim that *part* of computer science is the study of complexity. CS certainly offers many techniques for handling complexity: structured programming, abstraction, modularity, hierarchy, top-down design, stepwise refinement, object-oriented programming, recursion, etc. So, yes, CS is one way—perhaps even the best way—to *manage* (or *avoid*) complexity, but that does not mean that it is *the study of* complexity.

Indeed, (Denning et al., 1989, p. 11) point out that viewing " 'computer science [as] the study of abstraction and the mastering of complexity' ... also applies to physics,

mathematics, or philosophy"; no doubt many other disciplines also study complexity. So *defining* CS the study of complexity doesn't seem to be right.

Further Reading:
For more on CS and complexity, see (Lloyd, 1990), (Pylyshyn, 1992), and (Hartmanis, 1993, pp. 5–6).

3.14.4 CS Is the *Philosophy*(!) of Procedures

Could CS be the study of procedures (as Shapiro urged; see §3.9.3, above), yet be a branch of *philosophy* instead of science?

Here is an interesting definition:

> *Computer science is philosophy.* Logic is the foundation of philosophy. It's also the foundation of Computer Science. (Rupp, 2003, my italics)

The first sentence is the title of Rupp's essay. The next two sentences (which are the first two sentences of the essay) are the only support that he offers for his definition. But this argument is invalid: Just because two disciplines share a common foundation, it does not follow that one of them "is" the other, or that they are identical.

A more interesting argument can be found in an introductory CS text that claims that CS is neither a science nor the study of computers (Abelson et al., 1996, "Preface to the First Edition"). Rather, it is what the authors call 'procedural epistemology', that is:

> the study of the structure of knowledge from an *imperative* point of view, as opposed to the more *declarative* point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to." (Italics added.)

And epistemology is, after all, the branch of *philosophy* that studies knowledge and belief (see §2.8).

Further Reading:
(Abelson et al., 1996) was greatly influenced by Seymour Papert, a CS educator and creator of the Logo programming language (https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-8.html#%_chap_Temp_5).
For more on Papert, see (Papert, 1980) and §3.14.5, below.

"How to" is certainly important, and interestingly distinct from "what is". But is there really a difference between "how to" and "what is"? As (Bringsjord, 2006) argues, talk of procedures can be replaced by declarative talk of "first-order logic, and proofs and interpretations". Many imperative statements can be equally well be expressed as declarative ones: Consider, for example, Lisp programs, which appear to be merely declarative definitions of recursive functions. Or consider that each "p :- q"

rule of a Prolog program can be interpreted either procedurally ("to achieve *p*, execute *q*") or declaratively ("*p* if *q*").

Or consider Euclid's *Elements*, which was originally written in "how to" form (Toussaint, 1993): To construct an equilateral triangle using only compass and straightedge, follow this algorithm.[40] (Compare: To compute the value of this function *using only the operations of a Turing-machine*, follow this algorithm.)[41] But today it is expressed in "what is" form: The triangle that is constructed by following that algorithm is equilateral: "When Hilbert gave a modern axiomatization of geometry at the beginning of the present century, he asserted the bald existence of the line. Euclid, however, also asserted that it can be constructed" (Goodman, 1987, §4). (We'll return to this topic in §10.3.) Note that the declarative version of a geometry theorem can be considered to be a formal proof of the correctness of the procedural version. This is closely related to the notion of program verification, which we'll look at in Chapter 16.

Much more can be said on this issue. For example, there is a related issue in philosophy concerning the difference between knowing *that* something is the case (knowing that a declarative proposition is true) and knowing *how* to do something (knowing a procedure for doing it). This, in turn, may be related to Knuth's view of programming as teaching a computer (perhaps a form of knowing-that), contrasted with the view of a machine-learning algorithm that allows a computer to learn on its own by being trained (perhaps a form of knowing-how). The former can easily gain declarative "knowledge" of what it is doing so that it can be programmed to explain what it is doing; the latter not so easily. (We looked at this briefly in §3.6.1.)

Even if procedural language can be intertranslated with declarative language, the two are surely distinct. And, just as surely, CS *is* concerned with procedures! So, we need to be clearer about what we mean by 'procedure' (as well as phrases like 'computational thinking' or 'algorithmic thinking'). This is a philosophical issue worthy of discussion (and we'll return to it in Chapter 7: "What Is an Algorithm?").

Further Reading:
For a short bibliography on the procedural-declarative distinction, see "Some References on the Procedural-Declarative Controversy",
http://www.cse.buffalo.edu/∼rapaport/676/F01/proc.decl.html

3.14.5 CS Is *Computational Thinking*

A currently popular view is to say that CS is a "way of thinking", that "computational", or "algorithmic", or "procedural" thinking—about anything(!)—is what makes CS unique:

> CS is the new "new math," and people are beginning to realize that CS, like math, is unique in the sense that many other disciplines will have to adopt that *way of thinking*. It offers a sort of conceptual framework for other disciplines, and that's

[40] http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.01.0086:book=1:type=Prop:number=1

[41] For further discussion of the "to accomplish goal *G*, do procedure *P*" formula, see Ch. 17, below, or (Rapaport, 2017a).

> fairly new. ... Any student interested in science and technology needs to learn *to think algorithmically*. That's the next big thing.
> —Bernard Chazelle, interviewed in (Anthes, 2006, my italics)

Jeannette Wing's notion of "computational thinking" (Wing, 2006, echoing (Papert, 1980))[42] is thinking in such a way that a problem's solution "can effectively be carried out by an information-processing agent" (Wing, 2010) (see also (Guzdial, 2011)). Here, it is important not to limit such "agents" to computers, but to include humans (as (Wing, 2008a, p. 3719) admits).

The view of CS as computational thinking may offer compromises on several controversies: It avoids the procedural-declarative controversy, by including both concepts, as well as others. Her definition of CS (Wing, 2006, p. 34, col. 2) as "the study of computation—what can be computed and how to compute it" is nice, too, because the first conjunct clearly includes the theory of computation and complexity theory ('can' can include "can in principle" as well as "can efficiently"), and the second conjunct can be interpreted to include both software programming as well as hardware engineering. 'Study' is nice, too: It avoids the science-engineering controversy.

Another insight into "computational thinking" comes from a news item that "New South Wales [in Australia] ... has made it illegal to possess not just guns, *but digital files that can be used to create guns* using a 3D printer or milling machine" (New Scientist, 2016, my italics).[43] The point is that one can think of an object in two ways: (1) as a "completed" (or implemented) physical *object* or (2) as an *algorithm* for constructing it; the latter way of thinking is computational thinking. Note, too, that it is recursive: The completed physical object is the "base case"; the algorithm is the "recursive case".

Five years before Perlis, along with Newell & Simon, defined CS as the science of *computers*, he emphasized what is now called computational *thinking* (or *procedural* thinking:

> [T]he purpose of ... [a] first course in programming ... is not to teach people how to program a specific computer, nor is it to teach some new languages. *The purpose of a course in programming is to teach people how to construct and analyze processes.* ...
>
> A course in programming ..., if it is taught properly, is concerned with abstraction: the abstraction of constructing, analyzing, and describing *processes.* ...
>
> This, to me, is the whole importance of a course in programming. It is a simulation. The point is not to teach the students how to use [a particular programming language, such as] ALGOL, or how to program [a particular computer, such as] the 704. These are of little direct value. The point is *to make the students construct complex processes out of simpler ones* (and this is always present in programming) in the hope that the basic concepts and abilities will rub off. A

[42] (Papert, 1980) only mentions 'computational thinking' on p. 182 and 'procedural thinking' on p. 155, but his entire book can be thought of as an extended characterization of this kind of thinking and learning. For more on Papert and his version of computational thinking, see (Papert, 1996) and (Barba, 2016); see also §3.14.4, above.

[43] See the actual law at
http://parliament.nsw.gov.au/prod/parlment/nswbills.nsf/0/db59405bb0f2f006ca257eeb00194514/$FILE/2R%20Firearms%20and%20cognate.pdf

> properly designed programming course will develop these abilities better than any other course. (Perlis, 1962, pp. 209–210, my italics)

Further Reading:
For a commentary on Perlis's view of what is now called 'computational thinking', see (Guzdial, 2008). Similar points have been made by (Wheeler, 2013, p. 296), (Lazowska, 2014, p. A26), and (Scott and Bundy, 2015, p. 37).

Some of the features of computational thinking that various people have cited include: abstraction,[44] hierarchy, modularity, problem analysis, structured programming, the syntax and semantics of symbol systems, and debugging techniques.[45] Note that all of these are among the methods cited in §3.14.3 for handling complexity!

Here is another characterization of CS, one that also characterizes computational thinking:

> Computer science is in significant measure all about analyzing problems, breaking them down into manageable parts, finding solutions, and integrating the results. The skills needed for this kind of thinking apply to more than computer programming. They offer a kind of disciplined mind-set that is applicable to a broad range of design and implementation problems. These skills are helpful in engineering, scientific research, business, and even politics![46] Even if a student does not go on to a career in computer science or a related subject, these skills are likely to prove useful in any endeavor in which analytical thinking is valuable. (Cerf, 2016, p. 7)

Denning also recognizes the importance of "algorithmic thinking" (Denning, 2009, p. 33). However, he dislikes it as a *definition* of CS, primarily on the grounds that it is too narrow:

> Computation is present in nature even when scientists are not observing it or thinking about it. Computation is more fundamental than computational thinking. For this reason alone, computational thinking seems like an inadequate characterization of computer science. (Denning, 2009, p. 30)

Note that, by 'computation', Denning means Turing-machine computation. (For his arguments about why it is "present in nature", see the discussion in §3.9.3, above. A second reason why Denning thinks that defining CS as computational thinking is too narrow is that there are other equally important forms of thinking: "design thinking, logical thinking, scientific thinking, etc." (Denning et al., 2017).

[44] See Kramer 2007, Wing 2008a, pp. 3717–3719, and our discussion of abstraction and implementation in Ch. 14.

[45] See, for example, the list in Grover and Pea 2013, pp. 39–40.

[46] And even the humanities (Ruff, 2016)—WJR footnote.

Further Reading:
For more on computational thinking, see:

1. The homepage for the Center for Computational Thinking, http://www.cs.cmu.edu/∼CompThink/;
2. (Lu and Fletcher, 2009), which gives examples of how computational thinking can be introduced in primary- and secondary-school curricula even before any formal introduction to CS;
3. (Carey, 2010) (cited in §3.13.1.1, above), which argues for the value of algorithmic thinking in fields other than computer science (including finance and journalism).
4. (Tedre and Denning, 2016), which gives a good survey of the history of "computational thinking",
5. (Denning, 2017) and (Glass and Paulson, 2017), which cast a skeptical eye on the notion; and
6. (Pappano, 2017), which discusses how computational thinking is being taught at all levels.

3.14.6 CS Is *AI*

> [Computer science] is the science of how machines can be made to carry out *intellectual processes*. (McCarthy, 1963, p. 1, my italics)

> The goal of computer science is to endow these information processing devices with as much *intelligent behavior* as possible.
> (Hartmanis, 1993, p. 5, my italics) (see also (Hartmanis, 1995a, p. 10))

> Understanding the activities of an animal or human mind in algorithmic terms seems to be about the greatest challenge offered to computer science by nature. (Wiedermann, 1999, p. 1)

> Computational Intelligence *is* the manifest destiny of computer science, the goal, the destination, the final frontier. (Feigenbaum, 2003, p. 39)

These aren't exactly definitions of CS, but they could be turned into ones: Computer science—note: CS, *not* AI!—is the study of (choose one): (a) how to get computers to do what humans can do; (b) how to make computers (at least) as "intelligent" as humans; (c) how to understand (human or animal) cognition computationally.

As we will see in more detail in Chapter 6, the history of computers supports this: It is a history that began with how to get machines to do *some* human thinking (in particular, certain mathematical calculations), then more and more. And (as we will see in Chapter 8: "Turing's Analysis of Computation") the Turing machine, as a model of computation, was motivated by how *humans* compute: (Turing, 1936, §9) analyzes how humans compute, and then designs what we would now call a computer program that does the same thing. But the branch of CS that analyzes how humans perform a task and then designs computer programs to do the same thing is AI. So, the Turing machine was the first AI program!

But, as I will suggest in §3.15.2.1.1, *defining* CS as AI is probably best understood as a special case of its fundamental task: determining what tasks are computable.

3.14.7 Is CS *Magic*?

Figure 3.4: ©2014, John L. Hart FLP

> To engender empathy and create a world using only words is the closest thing we have to magic. —Lin-Manuel Miranda (2016)[47]

The great science-fiction author Arthur C. Clarke famously said that "Any sufficiently advanced technology is indistinguishable from magic" (http://en.wikipedia.org/wiki/Clarke's_three_laws). Could it be that the advanced technology of CS is not only indistinguishable from magic, but really *is* magic? Not magic as in tricks, but magic as in Merlin or Harry Potter? As one CS student put it,

> Computer science is very empowering. It's kind of like knowing magic: you learn the right stuff and how to say it, and out comes an answer that solves a real problem. That's so cool.
> —Euakarn (Som) Liengtiraphan, quoted in (Hauser, 2017, p. 16)

Brooks makes an even stronger claim than Clarke:

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He [sic] builds castles in the air, creating by the exertion of the imagination .... Yet the program construct, unlike the poet's words [or the magician's spells?], is real in the sense that it moves and works, producing visible outputs separate from the construct itself. ... ***The magic of myth and legend has come true in our time.*** *One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.*
> (Brooks, 1975, pp. 7–8, my emphases).[48]

[47] https://www.nytimes.com/2016/04/10/books/review/lin-manuel-miranda-by-the-book.html

[48] For a nice illustration of computational implementations of "things that never were", see Fig. 3.5. And compare what von Kármán says about engineering, quoted later in this book in Ch. 5, §5.3.

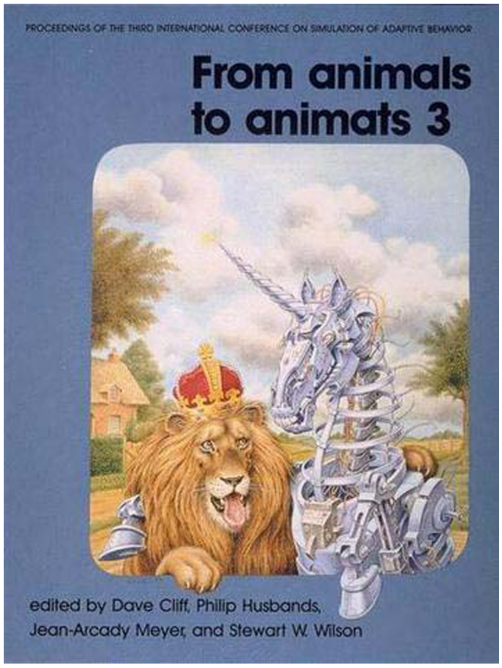


Figure 3.5: Computers can "show ... things that never were nor could be".

What is "magic"? Here's how one anthropologist defines it:

> In anthropology, magic generally means beliefs in the use of symbols to control forces in nature .... (Stevens, 1996, p. 721, col. 1)
>
> A definition of magic can be constructed to say something like the following: Magic involves the human effort to manipulate the forces of nature directly, through symbolic communication and without spiritual assistance.
> (Stevens, 1996, p. 723, col. 2).[49]

Clearly, programming involves exactly that kind of use of symbols. Or, as Abelson & Sussman put in their introductory CS text (which we discussed in §3.14.4):

> A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. *The programs we use to conjure processes are like a sorcerer's spells.* They are carefully composed from symbolic expressions in arcane and esoteric programming languages that prescribe the tasks we want our processes to perform. (Abelson et al., 1996, my italics)[50]

[49] For more on definitions of 'magic', see (Stairs, 2014).

[50] https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-9.html#%_idx_8

How is magic supposed to work? Anthropologist James G. Frazer (Frazer, 1915) "had suggested that primitive people imagine magical impulses traveling over distance through 'a kind of invisible ether.' " (Stevens, 1996, p. 722, col. 1). That sounds like a description of electrical currents running from a keyboard to a CPU, or information traveling across the Internet, or text messaging.

According to another anthropologist, Bronisław Malinowski,

> The magical act involves three components: the *formula*, the *rite*, and the condition of the *performer*. The rite consists of three essential features: the dramatic expression of emotion through *gesture* and physical attitude, the use of objects and substances that are *imbued with power by spoken words*, and, most important, the *words* themselves. (Stevens, 1996, p. 722, col. 2, my italics; citing Malinowski)

A "wizard", *gesturing* with a "wand", *performs* a "spell" consisting of a *formula* expressed in the *words* of an arcane language; the spell has real-world effects, *imbuing objects with power*.

We see all of this in computing: *Programs* play the role of spells; the *programmer* plays the role of the wizard; *a mouse, trackpad, or touchscreen* plays the role of the wand; *programming languages* (or, in the case of Siri or Alexa, English itself) plays the role of the arcane language; and *computations* are "powered" by "words" with real-world effects.

Here is another aspect of the role of symbols in magic:

> [A symbol] can take on the qualities of the thing it represents, and it can take the place of its referent; indeed, as is evident in religion and magic, *the symbol can become the thing it represents*, and in so doing, the symbol takes on the power of its referent. (Stevens, 1996, p. 724, my italics)

We see this happening in computers when we treat desktop icons (which are symbols) or the screen output of a WYSIWYG word processor (such as a page of a Microsoft Word document) as if they were the very things they represent. More significantly, we see this in the case of those computer simulations in which the simulation of something really *is* that (kind of) thing: In online banking, the computational simulation of transferring funds between accounts *is* the transferring of funds; digitized signatures on online Word or PDF documents carry legal weight; in AI, computationally simulated cognition (arguably) *is* cognition (Rapaport, 2012b, §8). And a National Research Council report (cited by (Samuelson et al., 1994, p. 2324, notes 44 & 46; 2325, note 47)) talks about user interfaces as "illusions":

> Unlike physical objects, the virtual objects created in software are not constrained to obey the laws of physics. ... In the desktop metaphor, for example, the electronic version of file folders can expand, contract, or reorganize their contents on demand, quite unlike their physical counterparts.
> (Samuelson et al., 1994, p. 2334)

Isn't that magic?

Newell says some things about the nature of physical symbol systems (that is, computers) that have "magical" overtones. The symbols of such a system "stand for some entity", that is:

> An entity X designates an entity Y relative to a process P, if, when P takes X as input, its behavior depends on Y. (Newell, 1980, p. 156)

Here, I take it that what Newell means is that P's behavior *really* depends on Y *instead of on X*, even though X (not Y) is P's input. But that seems to be the essence of magic; it is "action at a distance: The process behaves as if inputs, remote from those it in fact has, effect it" (Newell, 1980, p. 156). Process P behaves as it does because of a symbolic "spell" cast at a distance from P itself.

Perhaps computers are not just *metaphorically* magic (as Arthur C. Clarke might have said); they *are* magic (as Brooks said)!

However, there *is* a difference between computing and "the magic of myth and legend": The latter lacks (or at least fails to specify) any causal connection between incantation and result, whereas computation is quite clear about the connection: Recall the emphasis on algorithms (and see the discussion in §3.15.2.1.2, below). Thus, although CS may have the outward appearance of magic, and even accomplish (some of) the things that magic accomplishes, the way that it does it is different. CS has a method; magic does not. Actually, CS has more in common with magic *tricks* than with "real" magic,[51] because magic tricks require intermediary steps that accomplish the illusions of magic:

> Alan Perlis referred to AI researchers as "illusionists" because they try to create the illusion of intelligence. He argued they should be considered stage magicians rather than scientists. (Parnas, 2017, p. 5)

Another way to put this is that magic *tricks*—and computation—work "locally", whereas "real" magic is "non-local":

> The idea of locality emerged early in the history of science. For the Greek atomists, it was what distinguished naturalistic explanations from magical ones. Whereas the gods were believed to be capable of acting nonlocally, by simply willing remote events to occur, genuine causality for the atomists was always local, a matter of hard little atoms bumping into one another. (Holt, 2016, p. 50)[52]

Put another way, magic does what it does magically; CS does those things computationally:[53]

---

[51]

> "I'm writing a book on magic," I explain, and I'm asked, "Real magic?" By real magic people mean miracles, thaumaturgical acts, and supernatural powers. "No," I answer: "Conjuring tricks, not real magic." *Real magic, in other words, refers to the magic that is not real, while the magic that is real, that can actually be done, is not real magic.*
> —Lee Siegel, quoted in (Dennett, 2017, p. 318, my italics)

[52] In this regard, quantum mechanics would be more "magical" than computing, because it violates the principle of locality, allowing what Einstein called "spooky action at a distance".

[53] It's like the situation in Isaac Asimov's *Foundation* trilogy, where the character known as "the Mule"

Figure 3.6: ©2017 RWO Studios; http://rhymeswithorange.com/comics/october-26-2017/

> Everything going on in the software [of a computer] has to be physically supported by something going on in the hardware. Otherwise the computer couldn't do what it does from the software perspective—**it doesn't work by magic.**[54] But usually we don't have to know how the hardware works—only the engineer and the repairman do. We can act as though the computer just carries out the software instructions, period. **For all we care, as long as it works, it might as well *be* magic.** (Jackendoff, 2012, p. 99, my boldface, italics in original)

Further Reading:
The difference between magic and computation is related to the difference between what Dennett (1995, Ch. 3, §4) calls 'skyhooks' and 'cranes': "Skyhooks" are "imaginary" (or magical) devices that do their work (such as lift things up) miraculously, with no explanation; "cranes" are real (non-magical) devices that do the same thing "in an honest, non-question-begging fashion".

For other comments on computer science and magic, see (Crowcroft, 2005, p. 19n2), (Green, 2014), and Fig. 3.6.

---

does things intuitively that Hari Selden's followers needed to do step by step. But they did it so quickly that it was equivalent in input-output behavior to what the Mule did. This is related to the knowing-how/knowing-that distinction that we looked at earlier:

> The baseball player, who's thrown a ball over and over again a million times, might not know any equations but knows exactly how high the ball will rise, the velocity it will reach, and where it will come down to the ground. The physicist can write equations to determine the same thing. But, ultimately, both come to the identical point.
> —Geoffrey Hinton, quoted in (Mukherjee, 2017, p. 51)

[54] As the logician Joseph R. Shoenfield wrote, "a [computational] method must be *mechanical*. … [M]ethods which involve chance procedures are excluded … . **[M]ethods which involve magic are excluded** … . [M]ethods which require insight are excluded" (Shoenfield, 1967, p. 107, italics in original, my boldface).

3.15 So, What Is Computer Science?

It is time to take stock by summarizing the insights from our survey. But we have a long way to go to flesh out the details! You, the reader, should feel free—or even obligated!—to challenge this summary and to come up with *a reasoned one of your own.*

3.15.1 Computer Science and Elephants

Consider the fable of the blind men and the elephant: Six blind, wise men try to describe an elephant that they can only touch, not see. The first touches its side and says that the elephant is like a wall. The second touches its tusk and says that the elephant is like a spear. The third touches its trunk and says that the elephant is like a snake. The fourth touches its knee and says that the elephant is like a tree. The fifth touches its ear and says that the elephant is like a fan. The sixth touches its tail and says that the elephant is like a rope. As John Godfrey Saxe's 1873 poem sums it up,

> And so these men of Indostan
> Disputed loud and long,
> Each in his own opinion
> Exceeding stiff and strong,
> Though each was partly in the right,
> And all were in the wrong!
> (http://www.noogenesis.com/pineapple/blind_men_elephant.html)[55]

Our exploration of the various answers to the question "What is CS?" suggests that it has no simple, one-sentence answer. Any attempt at one is no better than the fabled blind men's descriptions of an elephant: Many, if not most or all, such attempts wind up describing the subject by focusing on only one aspect of it, as we saw with Newell, Perlis, & Simon and with Knuth.

Now that we have looked at all sides of our "elephant" (to continue the earlier metaphor), I would put it differently: CS is the scientific study of a *family* of topics surrounding both abstract (or theoretical) and concrete (or practical) computing: It is a "portmanteau" discipline.[56] Let me explain:

When the discipline was first getting started, it emerged from various other disciplines: "electrical engineering, physics, mathematics, or even business" (Hamming, 1968, p. 4). In fact, the first academic computer programming course I took (in Fortran)—the only one offered at my university in the late 1960s—was given by its School of Business.

Charles Darwin said that "all true classification … [is] genealogical" (Darwin, 1872, Ch. 14, §"Classification", p. 437). CS's genealogy involves two historical traditions: (1) the study of algorithms and the foundations of mathematics (from ancient Babylonian mathematics (Knuth, 1972a), through Euclid's geometry, to inquiries into

[55] See also https://en.wikipedia.org/wiki/Blind_men_and_an_elephant

[56] A "portmanteau" is a suitcase that opens into two equal sections. A "portmanteau *word*"—the term was coined by the great logician and storyteller Lewis Carroll (Carroll, 1871)—is one with "two meanings packed up into one word", like 'slithy' (meaning "lithe and slimy") or 'smog' (meaning "smoke and fog").

the nature of logic, leading ultimately to the Turing machine) and (2) the attempts to design and construct a calculating machine (from the Antikythera Mechanism of ancient Greece; through Pascal's and Leibniz's calculators and Babbage's machines; to the ENIAC, iPhone, and beyond). (We'll go into more detail in Chapter 6: "What Is a Computer? A Historical Perspective"; for a brief version from Hartmanis's point of view, see (Hartmanis, 1993, pp. 9–11).)

(Denning, 2003, p. 15) makes an offhand comment that has an interesting implication. He says, "Computer science was born in the mid-1940s with the construction of the first electronic computers." This is no doubt true. But it suggests that the answer to the question of what CS is *has to be* that it is the study of *computers*. The study of *algorithms* is much older, of course, dating back at least to Turing's 1936 formalization of the notion, if not back to Euclid's geometry or ancient Babylonian mathematics. Yet the study of algorithms is clearly part of modern CS. So, modern CS is the result of a marriage between (or merger of) the engineering problem of building better and better automatic calculating devices (itself an ancient endeavor) and the mathematical problem of understanding the nature of algorithmic computation. And that implies that modern CS has *both* engineering *and* science in its DNA. Hence its portmanteau nature.

The topics studied in contemporary CS roughly align along a spectrum ranging from the mathematical theory of computing, at one end, to the engineering of physical computers, at the other, as we saw in §3.4.1. (Newell, Perlis, & Simon were looking at this spectrum from one end; Knuth was looking at it from the other end.) The topics share a family resemblance (and perhaps nothing more than that, except for their underlying DNA), not only to each other, but also to other disciplines (including mathematics, electrical engineering, information theory, communication, etc.), and they overlap with issues discussed in the cognitive sciences, philosophy (including ethics), sociology, education, the arts, and business:

> I reject the title question ["Are We Scientists or Engineers?"]. ... Computer Science ... spans a multidimensional spectrum from deep and elegant mathematics to crafty programming, from abstraction to solder joints, from deep truth to elusive human factors, from scholars motivated purely by the desire for knowledge or practitioners making my everyday life better. It embraces the ethos of the scholar as well as that of the professional. To answer the question would be to exclude some portion of this spectrum, and I would be poorer for that. (Wulf, 1995, p. 57)

3.15.2 Five Central Questions of CS

In this section, rather than try to say *what* CS is the study of, or whether it is *scientific* or not, I want to suggest that it tries to answer five central questions. The single most central question is:

1A. What *can* be computed?

But to answer that, we also need to ask:

1B. *How* can it be computed?

The other questions follow logically from the central one. So, the five questions that CS is concerned with are:

1. **What *can* be computed, and *how*?**
2. **What can be computed *efficiently*, and how?**
3. **What can be computed *practically*, and how?**
4. **What can be computed *physically*, and how?**
5. **What can be computed *ethically*, and how?**

Let's consider each of these in a bit more detail:

3.15.2.1 Computability

3.15.2.1.1 *What* Can Be Computed?

> What is computation? This has always been the most fundamental question of our field. —(Denning and Wegner, 2010)

Question (1A) is the central question, because all other questions presuppose it. The fundamental task of any computer scientist—whether at the purely mathematical or theoretical end of the spectrum, or at the purely practical or engineering end—is to determine whether there is a computational solution to a given problem, and, if so, how to implement it. But those implementation questions are covered by the rest of the questions on the above list, and only make sense after the first question has been answered. (Alternatively, they facilitate answering that first question; in any case, they serve the goal of answering it.)

Question (1A) includes the questions:

What is *computation*?
What *kinds of things* are computed?
What is *computable*?

It is the question that logicians and computing pioneers Alonzo Church, Turing, Kurt Gödel, and others were originally concerned with—**Which mathematical functions are computable?**—and whose answer has been given as the Church-Turing Computability Thesis: A *function* is computable if and only if it is computable by a

Turing machine (or any formalism logically equivalent to a Turing machine, such as Church's lambda calculus or Gödel's general recursive functions). It is important to note that not all functions are computable. (A standard example of a *non*-computable function is the Halting Problem.) If all functions were computable, then computability would not be as interesting a notion.

Various branches of CS are concerned with identifying which problems can be expressed by computable functions. So, a corollary of the Computability Thesis is that a *task* is computable if and only if it can be expressed as a computable function. In (Soare, 2012, p. 3289)'s characterization, the output of a Turing machine "is the total number of 1's on the tape." So, the key to determining what is computable (that is, what kinds of tasks are computable) is finding a coding scheme that allows a sequence of '1's—that is, (a representation of) an integer—to be *interpreted as* a symbol, a pixel, a sound, etc.

Here are some examples:

- Is chess computable? (Shannon, 1950) investigated whether we can *computationally* analyze chess. (That is, can we play chess rationally?)
- Is cognition computable? The central question of AI is whether the functions that describe cognitive processes are computable. (This is one reason why I prefer to call AI "computational cognition" (Rapaport, 1995), (Rapaport, 2003).) Given the advances that have been made in AI to date, it seems clear that at least some aspects of cognition *are* computable, so a slightly more precise question is: *How much* of cognition is computable? (Rapaport, 2012b, §2, pp. 34–35).
- Is the weather computable? See Brian Hayes 2007a.
- Is fingerprint identification computable? See (Srihari, 2010).
- Is final-exam-scheduling computable? Faculty members in my department once debated whether it was possible to write a computer program that would schedule final exams with no time conflicts and in rooms that were of the proper size for the class. Some thought that this was a trivial problem; others thought that there was no such algorithm (on the (perhaps dubious!) grounds that no one in the university administration had ever been able to produce such a schedule). In fact, this problem is *NP*-complete (http://www.cs.toronto.edu/∼bor/373s13/L14.pdf). (See also an early discussion of this problem in (Forsythe, 1968, §3.3, p. 1027).)

This aspect of question (1A)—which *tasks* are computable?—is close to Forsythe's famous concern:

> The question "What can be automated?" is one of the most inspiring philosophical and practical questions of contemporary civilization. (Forsythe, 1968, p. 1025)

Although similar in intent, Forsythe's question can be understood in a slightly different way: Presumably, a process can be automated—that is, done automatically, by a machine, without human intervention—if it can be expressed as an algorithm. That is, computable implies automatable. But automatable does not imply computable: Witness the invention of the electro-mechanical, direct-dialing system in telephony, which

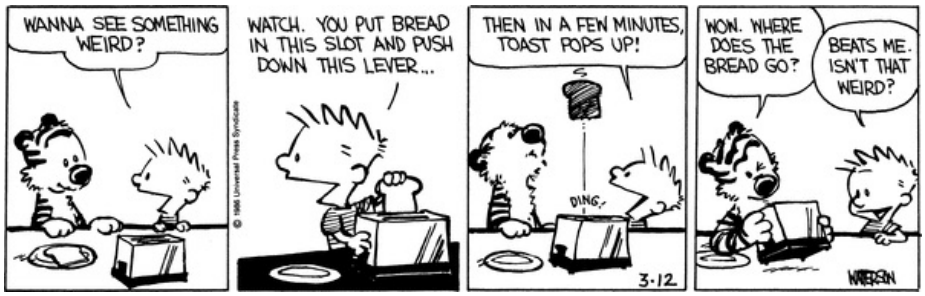


Figure 3.7: ©1986, Universal Press Syndicate

automated the task of the human operator. Yes, direct dialing is also *computable*, but it wasn't a computer that did this automation.[57]

3.15.2.1.2 *How* Is It Computable? Question (1B)—the "how" aspect of our central question—is equally important: CS cannot be satisfied with a mere existence statement to the effect that a problem *is* computable; it also requires a constructive answer in the form of an algorithm that explicitly shows *how* it is computable.

In the *Calvin and Hobbes* cartoon in Figure 3.7, Calvin discovers that if you input one thing (bread) into a toaster, it outputs something else (toast). Hobbes wonders what happened to the input. It didn't disappear, of course, nor did it "magically" turn into the output. The toaster *did something* to the bread (heated it); that intervening process is the analogue of an algorithm for the bread-to-toast function. Finding "intervening processes" requires algorithmic thinking, and results in algorithms that specify the transformational relations between input and output. (In psychology, behaviorism focused only on inputs and outputs: Pavlov's famous experiment input a bell to a dog, and the dog output saliva; but Pavlov didn't ask how the input and output were connected. *Cognitive* psychology focused on the intervening algorithms (Miller et al., 1960).)

In §2.8, we observed that, for every x, there is a philosophy of x. Similarly, we can ask, given some x, whether there is a computational theory of x. *Finding* a computational solution to a problem requires "computational thinking", that is, *algorithmic* (or procedural) thinking (as we discussed in §3.14.5, above).

Computational thinking includes what I call the five Great Insights of CS;[58] we'll revisit these in more detail in §7.5):

1. The *representational* insight:
   Only 2 nouns are needed to represent information
   ('0', '1')

2. The *processsing* insight:
   Only 3 verbs are needed to process information.
   (*move*(left or right), *print*('0' or '1'), *erase*)

[57] "Strowger Switch", https://en.wikipedia.org/wiki/Strowger_switch

[58] The first four were introduced at http://www.cse.buffalo.edu/∼rapaport/computation.html

3. The *structural* insight:
   Only 3 grammar rules are needed to combine actions.
   (sequence, selection, repetition)

4. The "*closure*" insight:
   Nothing else is needed.
   (This is the import of the Church-Turing Computability Thesis.)[59]

5. The *implementation* insight:
   The first three insights can be physically implemented.

And computational thinking involves both synthesis and analysis:

Synthesis: Design an algorithm to solve a *problem p*:

1. Express p as a mathematical function F_p
   (or a collection of interacting functions;
   that is, give an input-output specification of p).
2. Try to find or design an algorithm A_{F_p} for computing F_p
   (that is, for transforming the input to the output;
   then try to find or design an efficient and practical version of A_{F_p}).
3. Implement A_{F_p} on a physical computer.

(What I am calling synthetic computational thinking is closely related to an analysis of information processing put forth by David Marr, a pioneer of computational vision (Marr, 1982) (see the discussion in Chapter 17, "How Do Programs Relate to the World?", or (Rapaport, 2017a)).

Analysis: Understand the real world computationally:

Given a *real-world process p*
(physical, biological, psychological, social, economic, etc.),
try to find a *computational process* A_p that models p.

"Modeling" p is only one way to characterize what A_p does; we could also say that A_p "describes", or "simulates", or "explains" p, etc. (In Chapter 15, we'll look at whether computer programs are scientific theories that might "explain" a real-world process p.) Note that, once found, A_p can be re-implemented. In other words, the computational model of p can be implemented in a different "medium" from the one in which p was found. And that often means that p *itself* can be implemented in a different medium. (If p is cognition, then—to the extent that AI succeeds—computers can (be said to) think! (Rapaport, 2000b).)

[59] The exact number of nouns, verbs, or grammar rules depends on the formalism. E.g., some presentations add 'halt', 'read' or 'exit' as verbs, or use recursion as the single rule of grammar, etc. The point is that there is a very minimal set and that nothing else is *needed*. Of coure, more nouns, verbs, or grammar rules allow for greater ease of expression.

3.15.2.2 Efficient Computability

Question (2) is the question studied by the branch of computer science known as computational complexity theory. Given an algorithm, we can ask how much time it will take to be executed (roughly, the number of operations that will be needed) and how much space (memory) it will need. Computational-complexity theory is concerned with efficiency, because it is concerned with the economics of the spatio-temporal resources needed for computing. A more general question is this: Given the set of computable functions, which of them can be computed in, so to speak, less time than the age of the universe, or less space than the size of the universe? The principal distinction is whether a function is in the class called P (in which case, it is "efficiently" *computable*) or in the class NP (in which case it is *not* efficiently computable but it *is* efficiently "verifiable"):

> Even children can multiply two primes, but the reverse operation—splitting a large number into two primes—taxes even the most powerful computers. The numbers used in asymmetric encryption are typically hundreds of digits long. Finding the prime factors of such a large number is like trying to unmix the colors in a can of paint, ... "Mixing paint is trivial. Separating paint isn't." (Folger, 2016, p. 52)

Many, if not most, algorithms of practical importance are in P. By contrast, one important algorithm that is in NP is the Boolean Satisfiability Problem: Given a molecular proposition of propositional logic with n atomic propositions, under what assignment of truth-values to those atomic propositions is the molecular proposition true (or "satisfied")? Algorithms that are equivalent to Satisfiability are said to be "NP-complete":

> What [Turing-award winner Stephen] Cook did was show that every problem in NP has a reduction to satisfiability. Solve satisfiability and you can solve all of NP. If you have an efficient algorithm for solving satisfiability, then all the problems whose solutions we can efficiently check have efficient algorithms, and P = NP. ... "NP-complete" means those problems in NP powerful enough that they can be used to solve any other problem in NP. (Fortnow, 2013, pp. 54, 58)

Whether $P = NP$ is one of the major open questions in mathematics and CS. Most computer scientists both hope and believe that $P \neq NP$. Here's why:

> What happens if P = NP? We get a beautiful world where everything is easy to compute. We can quickly learn just about everything, and the great mysteries of the world fall quickly, from cures [for] deadly diseases to the nature of the universe. The beautiful world also has a dark underbelly, including the loss of privacy and jobs, as there is very little computers cannot figure out or accomplish. (Fortnow, 2013, p. 9)

Further Reading:
P is so-called because it is the class of functions computable in "*P*olynomial time", and *NP* is so-called because it is the class of functions computable in "*N*on-deterministic *P*olynomial time"; for more technical details, see https://en.wikipedia.org/wiki/Non-deterministic_Turing_machine and Bernhardt 2016, pp. 63–67.

On computational complexity and $P = NP$, see (Austin, 1983), (Cook, 1983), (Mycielski, 1983), (Fortnow, 2009), and (Walsh, 2014) for semi-technical discussions, and see (Fortnow, 2013) for a non-technical discussion.

3.15.2.3 Practical Computability

> His was a slap-dash method, but the world has been built slap-dash, and the beauty of mountain and river and sunset may be but the varnish with which the unskilled artificer hides his joins. —E.M. Forster (1910, Ch. 26, p 165)

Question (3) is considered both by complexity theorists as well as by more practically-oriented software engineers. Given a computable function in *P* (or, for that matter, in *NP*) what are some *practically* efficient methods of actually computing it? For example, under certain circumstances, some sorting algorithms are more efficient in a practical sense (for example, faster) than others. Even a computable function that is in *NP* might be practically computable in special cases. And some functions might only be practically computable "indirectly" via a "heuristic": A *heuristic for problem p* can be defined as an *algorithm* for some problem p', where the solution to p' is "good enough" as a solution to p (Rapaport, 1998, p. 406). Being "good enough" is, of course, a subjective notion; (Oommen and Rueda, 2005, p. 1) call the "good enough" solution "a *sub-optimal* solution that, hopefully, is arbitrarily close to the *optimal*." The idea is related to Simon's notion of "bounded rationality": We might not be able to solve p because of limitations in space, time, or knowledge, but we might be able to solve p' algorithmically within the required spatio-temporal-epistemic limits. And if the *algorithmic* solution to p' gets us closer to a solution to p, then it is a *heuristic* solution to p. But it is still an algorithm. A classic case of this is the Traveling Salesperson Problem, an *NP*-complete problem that software like Google Maps solves special cases of every day (even if their solutions are only "satisficing" ones (Simon, 1959, 1996a)).

Further Reading:

1. On heuristics:
   (a) There are two important surveys of meanings of the term 'heuristic': (Romanycia and Pelletier, 1985) and (Chow, 2015).
   (b) Other discussions include: (Simon and Newell, 1958) (which, in addition to its discussion of the division of labor—see §6.5.3—and its infamous prediction that a computer would become the world chess champion by 1968, this classic paper also distinguishes algorithmic problem solving of "well-structured" problems from heuristic problem solving of "ill-structured" problems), (Newell and Simon, 1976), (Korf, 1992), (Shapiro, 1992a), (Findler, 1993), and the classic (Polya, 1957).
   (c) (Thagard, 2007) is not about heuristics, but presents a theory of "approximate" truth that bears a close resemblance to the idea that a heuristic is an algorithm that computes an approximately correct answer.
2. Satisficing is "finding optimal solutions for a simplified world, or finding satisfactory solutions for a more realistic world" (Simon, 1978, p. 350).
3. On the difference between computer *science* approaches to an important theoretical problem and computer *engineering* approaches to a practical version of the same problem, see (Vardi, 2014).

3.15.2.4 Physical Computability

Question (4) brings in both empirical (hence scientific) and engineering considerations. To the extent that the only (or the best) way to decide whether a computable function really does what it claims to do is to execute it on a real, physical computer, computers become an integral part of CS. Even a practically efficient algorithm for computing some function might run up against physical limitations. Here is one example: Even if, eventually, computational linguists devise practically efficient algorithms for natural-language "competence" (understanding and generation; (Shapiro, 1989), (Shapiro and Rapaport, 1991)), it remains the case that humans have a finite life span, so the infinite capabilities of natural-language competence are not really required (a Turing machine isn't needed; a push-down automaton might suffice). This is also the question that issues in the design and construction of real computers ("computer engineering") are concerned with. And it is where investigations into alternative physical implementations of computing (quantum, optical, DNA, etc.) come in.

3.15.2.5 Ethical Computability

Question (5) brings in ethical considerations. Arden, elaborating Forsythe's question, said that "the basic question [is] ... what can *and should* be automated" (Arden, 1980, p. 29, my italics) (Tedre 2015, pp. 167–168 makes the same elaboration). As Matti Tedre (2008, p. 48, my italics) observes,

> Neither the theoretician's question "What can be efficiently automated?" nor the practitioner's question "How can processes be automated reliably and efficiently?"

include, explicitly or implicitly, any questions about *why* processes should be automated at all, if it is *desirable* to automate things or to introduce new technologies, or *who decides* what will be automated.

Actually, the question "What should be computed?" is slightly ambiguous. It could simply refer to questions of practical efficiency: Given a sorting problem, which sorting algorithm *should* be used; that is, which one is the "best" or "most practical" or "most efficient" in the actual circumstances? But this sense of 'should' does not really differentiate this question from question (3).

It is the *ethical* interpretation that makes this question interesting: Suppose that there is a practical and efficient algorithm for making certain decisions (for example, as in the case of autonomous vehicles). There is still the question of whether we *should* use those algorithms to actually make decisions for us. Or let us suppose that the goal of AI—a computational theory of cognition—is practically and efficiently computable by physically plausible computers. One can and should still raise the question whether such "artificial intelligences" *should* be created, and whether we (their creators) have any ethical or moral obligations towards them, and vice versa! (See (Delvaux, 2016), (Nevejans, 2016).) And there is the question of implicit biases that might be (intentionally or unintentionally) built into some machine-learning algorithms. (We will discuss these topics at greater length in Chapter 18: "Computer Ethics I: Are There Decisions Computers Should Never Make?" and Chapter 20: "Computer Ethics II: Should We Build Artificial Intelligences?".)

3.15.3 Wing's Five Questions

I said that CS *is* concerned with five central questions. It might have been better to say that it *should be* concerned with them. You will see why, when I compare our five questions to Wing's "Five Deep Questions in Computing" (Wing, 2008b).

1. Wing's *first* question is

   $P = NP$?

   This is part of our second question: "What is efficiently computable?".

2. Curiously, her *second* question:

   What is computable?

   is our *central* one! (I should note, however, that a later essay (Wing, 2008a, p. 3724) says that her five questions are a "set", thus "no ordering implied".)

3. Her third question is:

   What is intelligence?

   This can be rephrased as "How much of (human) cognition is computable?", which is a special case of our central question.

4. Her fourth question:

   > What is information?

   can be seen as asking an ontological question about the nature of what it is that is computed: Is it numbers (0s and 1s)? Is it symbols ('0's and '1's)? Is it information in some sense (and, if so, in which sense)? (Recall from our discussion in Ch. 2, §2.8, that ontology is the philosophical study of what kinds of things exist.) In the present context, "What is information?" is closely related to the question we asked earlier in this chapter (§3.4.2) about what objects CS studies. Thus, it is an aspect of our central question.

5. Wing's last question:

   > (How) can we build complex systems simply?

   is ambiguous between two readings of 'build': (a) On a software reading, this question can be viewed in an abstract (scientific, mathematical) way as asking about the structural nature of software: Structured programming and the issues concerning the proper use of the "goto" statement (Dijkstra, 1968) would fall under this category. As such, it concerns the grammar rules, and so it is an aspect of our central question. (b) On a hardware reading, it is an engineering question: How should we build *physical* computers? On that interpretation, it is part of our fourth question.

(Wing, 2008a, p. 3724) adds a sixth question: "the most basic question of all: *what is a computer?*". Whether or not this is the most basic question (isn't "What is computable?" more basic?), it would seem to be an aspect of the "how" part of either our central question or our fourth question: How can something be computed physically?

Thus, Wing's five questions can be boiled down to two:

- What is computation such that only some things can be computed? (And what can be computed (efficiently), and how?)

This is equivalent to our questions (1)–(3).

- (How) can we build devices to perform these computations?

This is equivalent to our question (4). And, in this case, we see once again the two parts of the discipline: the scientific (or mathematical, or abstract) and the engineering (or concrete).

But it is interesting and important to note that none of Wing's questions correspond to our ethical question (5). As computer scientist and philosopher Robin K. Hill observes:

> Whereas the philosophy of computer science has heretofore been directed largely toward the study of formal systems by means of other formal systems … concerned professionals have also devoted attention to the ethics of computing, taking on issues like privacy, the digital divide, and bias in selection algorithms. Let's keep it up. There are plenty. (Hill, 2017a)

3.15.4 Conclusion

I said that our survey *suggests* that there is no *simple, one-sentence* answer to the question: What is computer science? If we were to summarize the discussion in this chapter in *one* sentence, it would look something like this:

> Computer science is the scientific (or STEM) study of:
>
> what problems can be solved,
> what tasks can be accomplished,
> and what features of the world can be understood ...
>
> ... *computationally*, that is, using a language with only:
>
> 2 nouns ('0', '1'),
> 3 verbs ('move', 'print', 'halt'),
> 3 grammar rules (sequence, selection, repetition),
> and nothing else,
>
> and then to provide algorithms to show how this can be done:
>
> efficiently,
> practically,
> physically,
> and ethically.

But this definition is hardly a *simple* sentence!

However, one of the opening quotations for this chapter—from an interview with a computational musician—comes closer, so we will end where this chapter began:

> **The Holy Grail of computer science is *to capture the messy complexity of the natural world* and *express it algorithmically.***
> — Teresa Marrin Nakra, quoted in (Davidson, 2006, p. 66, my italics).

3.16 A Look Ahead

We are now ready to look into all of these issues in more detail, so that we'll be able to have a better grasp of exactly what CS is. A more complete answer is going to depend on answers to many other questions. In the next chapter, we will look at the first one: **What is science?**

3.17 Questions for the Reader

1. Computer scientist and philosopher Amnon H. Eden (Eden, 2007) seeks to bring clarity to the science-vs.-math-vs.-engineering controversy by taking up a distinction due to Peter Wegner (Wegner, 1976) among three different "Kuhnian paradigms" (see Ch. 4, §4.9.2): a view of CS as (1) a "rationalist" or "mathematical" discipline, (2) a "technocratic" or "technological" discipline, and (3) a "scientific" discipline. ((Tedre and Sutinen, 2008) also discusses these three paradigms.) Eden then argues in favor of the scientific paradigm.

   But must there be a single paradigm? Are there any disciplines with multiple paradigms? Does the existence of multiple paradigms mean that there is no unitary discipline of CS? Or can all the paradigms co-exist?

2. Journalist Steve Lohr quotes a high-school math and CS teacher as saying, "I do feel that computer science really helps students understand mathematics ... And I would use computers more in math, if I had access to a computer lab." (Lohr, 2008)

   Is CS best seen as the use of a physical tool, or as the study of (as well as the use of) a method of thinking ("computational thinking")?

3. The philosopher Gottfried Wilhelm Leibniz (1646–1716) thought that a *lingua characteristica universalis* (or universal formal language) and a *calculus ratiocinator* (or formal logic) would offer "mankind ... a new instrument which will enhance the capabilities of the mind to a far greater extent than optical instruments strengthen the eyes" (Leibniz, 1677, p. 23). From this statement, computer scientist Moshe Vardi derives a "definition of computing, as an 'instrument for the human mind' " (Vardi, 2011a). This is similar to Daniel C. Dennett's suggestion that the computer is a "prosthesis" for the mind (see, for example, (Dennett, 1982)).

   Is that a reasonable definition of CS?

4. In §3.12, I said that it makes no—or very little—sense to have a program without a computer to run it on. That a computer is useful, but not necessary, is demonstrated by the "Computer Science Unplugged" project (http://csunplugged.org/). And some of the earliest AI programs (for playing chess) were executed by hand.[60]

   So, *did* these programs "have a computer to run on"? Were the humans, who hand-executed them, the "computers" that these programs "ran on"? When you debug a computer program, do you do the debugging by hand?[61]

[60] (Shannon, 1950), (Turing, 1953), https://chessprogramming.wikispaces.com/Turochamp

[61] Thanks to Stuart C. Shapiro for this suggestion.

5. Forsythe observed that,

> in the long run the solution of problems in field X on a computer should belong to field X, and CS should concentrate on finding and explaining the principles ["the methodology"] of problem solving [with computers]. (Forsythe, 1967b, p. 454)

   Should contributions made by AI researchers to philosophy or psychology be considered to be the results of AI? Or are they philosophical or psychological results that were only *produced* or *facilitated* by computational techniques?

6. Maybe when Knuth says that CS is the "study" of algorithms, by 'study' he means *both* science *and* engineering. In what sense does the study of electricity belong both to engineering and to science? Certainly, the science of physics studies electricity as a physical phenomenon. And, certainly, electrical engineering studies electricity from an engineering perspective. But physics and electrical engineering are typically considered to be separate (albeit related) disciplines.

   Should the same be said for computer *science* (which would study algorithms) and computer *engineering* (which would study computers and, perhaps, software engineering)?

7. (Arden, 1980, p. 9) suggests, but does not endorse, a "committee-produced, all-purpose" definition:[62] "computer science is the study of the design, analysis, and execution of algorithms, in order to better understand and extend the applicability of computer systems". Note that this avoids the science-vs.-engineering quandary, by its use of 'study', and tries to cover all the ground. Arden suggests, however, that his entire book should be taken as the "elaboration" of this definition.

   Isn't this like saying that CS is what computer scientists do?

8. "*Computer Science* is the science of using computers to solve problems" (George Washington University Department of Computer Science, 2003) (see also (Roberts, 2006, p. 5)). Because this definition doesn't limit the kind of problems being solved, it has the advantage of illuminating the generality and interdisciplinarity of CS. And, because it implicitly includes the software (algorithmic) side of computing—after all, you can't use a computer to solve a problem unless it has been appropriately programmed—it nicely merges the computer-vs.-algorithm aspects of the possible definitions. Something more neutral could just as easily have been said: Computer science is the science of solving problems computationally, or algorithmically—after all, you can't solve a problem that way without executing its algorithmic solution on a computer.

   But can there really be a *science* of problem solving? And, even if there could be, is it CS? Or is that *all* that CS is?

[62] That is, a "klunky" one designed to be acceptable to a variety of competing interests. The standard joke about such definitions is that a camel is a horse designed by a committee. See http://en.wikipedia.org/wiki/Design_by_committee

9. As we mentioned in Ch. 2, §2.6.2.2, (McGinn, 2015b) argues that philosophy is a science just like physics (which is an empirical science) or mathematics (which is a "formal" science), likening it more to the latter than the former (p. 85). To make his argument, he offers this characterization of science:

   > [W]hat distinguishes a discourse as scientific are such traits as these: rigor, clarity, literalness, organization, generality (laws or general principles), technicality, explicitness, public criteria of evaluation, refutability, hypothesis testing, expansion of common sense (with the possibility of undermining common sense), inaccessibility to the layman, theory construction, symbolic articulation, axiomatic formulation, learned journals, rigorous and lengthy education, professional societies, and a sense of apartness from naïve opinion. (McGinn, 2015b, p. 86)

   Does CS fit that characterization?

10. In §3.5.4, we considered the possibility that CS is not a "coherent" discipline. Consider the following interpretation of the blind-men-and-the-elephant story:

    > The man at the tail is sure he has found a snake; the man at the tusks believes he's holding spears. Through teamwork, they eventually discover the truth. "But what if they were wrong?" [magician Derek] DelGaudio asks onstage. "What if that thing was some sort of magical creature that had a snake for a nose and tree-trunk legs, and they convinced it it was an elephant? Maybe that's why you don't see those things anymore." (Weiner, 2017)

    Might CS have been such a "magical creature"? Is it still?

11. In this chapter, we asked what CS is: Is it a science? A branch of engineering? Or something else? But we could also have responded to the question with another one: *Does it matter?* Is it the case that, in order for a discipline to be respectable, it has to be (or claim to be!) a science? Or is it the case that a discipline's usefulness is more important? (For instance, whether or not medicine is a science, perhaps what really matters is that it is a socially useful activity that draws upon scientific—and other!—sources.)[63]

    So: Does it matter what CS is? And what would it mean for a discipline to be "useful"?

[63] Thanks to Johan Lammens (personal communication, 2017) for the observations in this question.

12. Jeannette M. Wing says this about computational thinking:

> I argued that the use of computational concepts, methods and tools would transform the very conduct of every discipline, profession and sector. Someone with the ability to use computation effectively would have an edge over someone without. So, I saw a great opportunity for the computer science community to teach future generations how computer scientists think. Hence "computational thinking."

How *do* computer scientists think? At the very least, we might say that they think *procedurally*. Is that the same as saying that they think *algorithmically*? We might also say that they think *recursively*. Because procedural theories of computability (such as Turing's) are logically equivalent to recursive theories (such as Gödel's), is procedural (or algorithmic) thinking the same as recursive thinking? Is thinking *abstractly* part of computational thinking, as in the case of procedural abstraction (or is thinking abstractly merely something that is more generally part of thinking "logically" or "scientifically")? Are there other ways in which computer scientists think that is unique to computer science?

13. A related (but distinct) question is: What is a computer scientist? Bill Gasarch (https://blog.computationalcomplexity.org/2018/09/what-is-physicist-mathematician.html) considers a number of reasons why the answer to this question is not straightforward: Does it depend on whether the person is in a CS department? Whether the person's degree is in CS? What the person's research is? For example, the computer scientist Scott Aaronson received a prize in physics, yet he insists that he his not a physicist (Aaronson, 2018). Read Gasarch's post and try to offer some answers. (We'll return to this issue in §15.4.4.)

3.18 Further Sources of Information

Items in **boldface** are especially interesting.

1. Books and Essays:

   (a) Abrahams, Paul (1987), "What Is Computer Science?", *Communications of the ACM* 30(6) (June): 472–473.

   - "My personal definition of the field and its name would be '*computology*: the study of computational processes and the means by which they may be realized.' But alas, the name 'computer science,' like OS/360 Job Control Language, will probably persist until the sun grows cold" (p. 473).

   (b) Abrahams, Paul W.; & Lee, Graham (2013), "Computer Science Is Not a Science" (two separate letters to the editor), *Communications of the ACM* 56(1) (January): 8.

   (c) Bornat, Richard (2006), "Is 'Computer Science' Science?", *4th European Conference on Computing and Philosophy (ECAP'06, 22–24 June, Trondheim, Norway)*.

   (d) Cerf, Vinton G. (2012), "Where Is the Science in Computer Science?", *Communications of the ACM* 55(10) (October): 5.

   - Argues that, even if CS might once have focused on computing machines, it should now be more focused on "predict[ing] likely outcomes based on models[, which] is fundamental to the most central notions of the scientific method".

   (e) Crowcroft, Jon (2005), "On the Nature of Computing", *Communications of the ACM* 48(2) (February): 19–20.

   - Argues that CS is not the study of either the natural or the artificial but of the *virtual*.
   - Another essay on CS and "the virtual" is (Pylyshyn, 1992).

   (f) Denning, Peter J. (1980), "What Is Experimental Computer Science?", *Communications of the ACM* 23(10) (October): 543–544.

   (g) Denning, Peter J. (1999), "Computer Science: The Discipline", in Anthony Ralston & David Hemmindinger (eds.) (2000), *Encyclopedia of Computer Science*: PAGES.

   (h) Denning, Peter J. (2005), "Is Computer Science Science?", *Communications of the ACM* 48(4) (April): 27–31.

   (i) **Denning, Peter J.; Comer, Douglas E.; Gries, David; Mulder, Michael C.; Tucker, Allen; Turner, A. Joe; & Young, Paul R. (1989), "Computing as a Discipline", *Communications of the ACM* 32(1) (January): 9–23.**,
   https://pdfs.semanticscholar.org/0d3f/facbb4f5b004a14a2ab05e798f68159b2bc4.pdf

   - "The final report of the Task Force on the Core of Computer Science presents a new intellectual framework for the discipline of computing and a new basis for computing curricula. ... Short Definition[:] The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, 'What can be (efficiently) automated?' " (pp. 9, 12). A longer definition is in the Appendix.

   (j) Easton, Thomas A. (2006), "Beyond the Algorithmization of the Sciences", *Communications of the ACM* 49(5) (May): 31–33.

(k) Ensmenger, Nathan (2011), "Computing as Science and Practice", *Science* 333 (9 September): 1383.

(l) Feitelson, Dror G. (guest ed.) (2007), Special Issue on Experimental Computer Science, *Communications of the ACM* 50(11) (November): 24–59.

(m) Forsythe, George F. (1967), "A University's Educational Program in Computer Science", *Communications of the ACM* 10(1) (January): 3–8.

(n) Hayes, Brian (2015), "Cultures of Code", *American Scientist* 103(1) (January-February): 10–13, http://www.americanscientist.org/issues/pub/cultures-of-code

- Distinguishes "three communities in the world of computation": computer science, computational science, and software development.

(o) Hsu, Francis (2013), "Is Computing Science?", *Communications of the ACM* 56(8) (August): 9.

- Argues that "there are no clear boundaries" between branches of knowledge.

(p) Kukla, A. (1989), "Is AI an Empirical Science?", *Analysis* 49: 56–60.

- The answer is: No; it is an *a priori* science or discipline like mathematics.
- For the opposite point of view, see:
  Burkholder, Leslie (1999), "Are AI and Mechanics Empirical Disciplines?", *Journal of Experimental and Theoretical Artifical Intelligence* 11: 497–500.

(q) Naur, Peter (1995), "Computing as Science", Appendix 2 of *An Anatomy of Human Mental Life* (Gentofte, Denmark: naur.com Publishing): 208–217, http://www.naur.com/naurAnat-net.pdf, http://www.naur.com/Nauranat-ref.html.

(r) Tedre, Matti (2015), *The Science of Computing: Shaping a Discipline* (Boca Raton, FL: CRC Press).

- An investigation of the philosophical issues around the nature and history of computer science, examining whether it is a science, and, if so, what kind of science it might be.

(s) Tedre, Matti; & Moisseinen, Nella (2014), "Experiments in Computing: A Survey", *Scientific World Journal*, Vol. 2014, Article ID 549398, http://dx.doi.org/10.1155/2014/549398

- A survey of the nature of experiments in science, and whether CS is experimental in nature.

(t) Tedre, Matti; & Sutinen, Erkki (2008), "Three Traditions of Computing: What Educators Should Know", *Computer Science Education* 18(3) (September): 153–170.

2. Websites:

   (a) Chazelle, Bernard (2006), "The Algorithm: Idiom of Modern Science" http://www.cs.princeton.edu/∼chazelle/pubs/algorithm.html

   (b) The College Board (2012), "AP Computer Science: Principles; Course Annotations", http://www.collegeboard.com/prod_downloads/computerscience/1_Annotations-08-21-12_ld.pdf

   (c) CSTA [Computer Science Teachers Association] Curriculum Improvement Task Force (2005), "The New Educational Imperative: Improving High School Computer Science Education" (New York: Association for Computing Machinery), http://csta.acm.org/Communications/sub/DocsPresentationFiles/White_Paper07_06.pdf

      - Relevant to this chapter, see §1.1 (pp. 17–19), "What Is Computer Science Anyway?".

   (d) Denning, Peter J., "Great Principles of Computing", http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html

   (e) Denning, Peter J. (ed.), Symposium on "The Science in Computer Science", *Ubiquity*, http://ubiquity.acm.org/symposia.cfm

   (f) Department of Computing Sciences, Elon University, Elon, NC (2007), "What Is Computer Science?", http://www.elon.edu/e-web/academics/elon_college/computing_sciences/curriculum/cs.xhtml

      - This is from the departmental website of a small college's "computing sciences" (note the name!) department. It discusses pretty much all the issues we've been looking at: Is it a science? What does it study? Is it an engineering discipline?

   (g) Tedre, Matti (2007), "Lecture Notes in the Philosophy of Computer Science", especially the following lectures:

      - "Introduction to the Course" (http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes1.pdf),
      - "Part I: Struggling for Status" (http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes2.pdf),
      - "Part II: Emerging Interdisciplinarity" (http://cs.joensuu.fi/∼mmeri/teaching/2006/philcs/files/lecture_notes3.pdf)

Chapter 4

What Is Science?

Version of 28 June 2019 DRAFT © 2004–2019 by William J. Rapaport

> Science is the great antidote to the poison of enthusiasm and superstition. (Smith, 1776, V.1.203)

> *The most remarkable discovery made by scientists is science itself.* The discovery must be compared in importance with the invention of cave-painting and of writing. Like these earlier human creations, science is an attempt to control our surroundings by entering into them and understanding them from inside. And like them, science has surely made a critical step in human development which cannot be reversed. We cannot conceive a future society without science. (Bronowski, 1958, my italics)

> [A] science is an evolving, but never finished, interpretive system. And fundamental to science ... is its questioning of what it thinks it knows. ... Scientific knowledge ... is a system for coming to an understanding. (Barr, 1985)

> Science is *all about* the fact that we don't know everything. *Science is the learning process.* (Dunning, 2007)

> [S]cience is not a collection of truths. It is a continuing exploration of mysteries. (Dyson, 2011b, p. 10)

4.1 Readings:

In doing these readings, remember that our ultimate question is whether CS is a science.

0. Okasha, Samir (2002), *Philosophy of Science: A Very Short Introduction* (Oxford: Oxford University Press).

- This is the very best introduction to philosophy of science, although it's an entire (but short) book. You may read it instead of any of the following.

1. Required:

   - Kemeny, John G. (1959), *A Philosopher Looks at Science* (Princeton: D. van Nostrand).
     - Introduction, pp. ix–xii
     - Ch. 5, "The [Scientific] Method", pp. 85–105.
     - You can skim Ch. 10, "What Is Science?", pp. 174–183, because his answer is just this: A science is any study that follows the scientific method.
   - Kolak, Daniel; Hirstein, William; Mandik, Peter; & Waskan, Jonathan (2006), *Cognitive Science: An Introduction to Mind and Brain* (New York: Routledge), §4.4.2. "The Philosophy of Science"
   - Papineau, David (1996), "Philosophy of Science", in Nicholas Bunnin & E.P. Tsui-James (eds.), *The Blackwell Companion to Philosophy* (Oxford: Blackwell): 290–324.
     - "The Problem of Induction", §1.1, pp. 290–291
     - "Popper's Falsificationism", §1.2, pp. 291–293
     - "The Failings of Falsificationism", §1.3, pp. 293–294
     - "Instrumentalism versus Realism", §1.5, pp. 298–300
     - "Theory, Observation and Incommensurability", §1.6, pp. 300–302
     - "The Underdetermination of Theory by Evidence", §1.7, pp. 302–305 "The Pessimistic Meta-Induction", §1.8, pp. 305–307
     - "Naturalized Epistemology of Science", §1.9, pp. 307–308
     - "Causation", §2.1, pp. 308–310
     - Skim the rest.

2. Very Strongly Recommended:

   - Quine, Willard van Orman (1951), "Two Dogmas of Empiricism", *Philosophical Review* 60: 20–43, http://www.ditext.com/quine/quine.html
   - Popper, Karl R. (1953), "Science: Conjectures and Refutations", in Karl R. Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge* (New York: Harper & Row, 1962), http://xxsy.library.nenu.edu.cn/pluginfile.php/1066/mod_resource/content/1/%5BKarl_Popper%5D_Conjectures_and_Refutations_The_Gro(Bookos.org)%20(1).pdf
   - Kuhn, Thomas S. (1962), *The Structure of Scientific Revolutions* (Chicago: University of Chicago Press), Ch. IX, "The Nature and Necessity of Scientific Revolutions", http://www.marxists.org/reference/subject/philosophy/works/us/kuhn.htm

3. Strongly Recommended:

   - Hempel, Carl G. (1966), *Philosophy of Natural Science* (Englewood Cliffs, NJ: Prentice-Hall), "Scope and Aim of this Book".
     - On empirical vs. non-empirical sciences.
   - Kyburg, Henry E., Jr. (1968), *Philosophy of Science: A Formal Approach* (New York: Macmillan), Ch. 1, "The Concept of a Formal System".
   - Ziman, John M. (1968), "What Is Science?", in John M. Ziman, *Science Is Public Knowledge* (Cambridge, UK: Cambridge University Press), reprinted in Alex C. Michalos (ed.) *Philosophical Problems of Science and Technology* (Boston: Allyn & Bacon, 1974): 1–27.
   - Salmon, Wesley C. (1984), *Scientific Explanation and the Causal Structure of the World* (Princeton: Princeton University Press), Ch. 1, "Scientific Explanation: Three General Conceptions".

4.2 Introduction

> All these processes are very complex, and they tend to follow the rule that the more you find out about them, the more you discover that you didn't know .... That is both the joy and the frustration of science .... —Gregory L. Murphy (2019, §1)

We have seen that one answer to our principal question—What is CS?—is that it is a science (or that parts of it are science). Some say that it is a science of computers, some that it is a science of algorithms or procedures, some that it is a science of information processing. And, of course, some say that it is not a science at all, but that it is a branch of engineering. In Chapter 5, we will explore what engineering is, so that we can decide whether CS is a branch of engineering. In the present chapter, we will explore what it means to be a science, so that we can decide whether CS is one (or whether parts of it are).[1]

In keeping with the definition of philosophy as the *personal search* for truth by rational means (§2.7), I won't necessarily answer the question, "Is CS a science?". But I will provide considerations to help *you* find *and defend* an answer that *you* like. It is more important for you to determine an answer for yourself than it is for me to present you with my view; this is part of what it means to do philosophy *in* the first person *for* the first person. And it is very important for you to be able to *defend* your answer; this is part of what it means to be rational (it is the view that philosophy is intimately related to critical thinking). We will follow this strategy throughout the rest of the book.

4.3 Science and Non-Science

The word 'science' originally simply meant "knowledge" or "knowing"; it derives from the Latin verb *scire*, which meant "to know".[2] But, of course, it has come to mean much more than that.

Let's begin by contrasting the term 'science' with some other terms. First, of course, science is often opposed to engineering. Because this will be our focus in Chapter 5, I won't say more about it here.

Second, science is sometimes opposed to "art", not only in the sense of the fine arts (such as painting, music, and so on) but also in the sense of an informal body of experiential knowledge, or tricks of the trade: information that is the result of personal experience, perhaps unanalyzable (or, at least, unanalyzed), and creative. This is "art" in the sense of "the art of cooking". By contrast, science is formal, objective, and systematic.

This contrast can be seen in the titles of two classic texts in CS: Donald Knuth's *The **Art** of Computer Programming* (Knuth, 1973) and David Gries's *The **Science** of Programming* (Gries, 1981). The former is a multi-volume handbook of different techniques, catalogued by type, but analyzed (albeit incompletely by today's standards).

[1] Recall from Ch. 3 that we will refer to computer science as 'CS' so as not to beg any questions about whether it is a science simply because its name suggests that it is.

[2] According to the *Oxford English Dictionary*, http://www.oed.com/view/Entry/172672. The word 'scientist' was coined by the philosopher William Whewell, on a parallel with 'artist'; see http://www.oed.com/view/Entry/172698.

The latter is a compendium of formal methods for program development and verification, an application of logic to programming. (For a detailed defense of the title of Knuth's work, see (Knuth, 1974a); recall our discussion of Knuth's views on art vs. science in §3.14.2.)

Finally, science is opposed (both semantically and politically) to "pseudo-science": any discipline that masquerades as science, but is not science. The problem of determining the dividing line between "real" science and "pseudo"-science is called the 'demarcation problem'. For example, almost everyone will agree that *astronomy* is a "real" science and that *astrology* is not. But what is the difference between "real" and "pseudo"-sciences? We will return to this in §4.9.1, because to explain the contrast between science and pseudo-science is part of the philosophical exploration of what science is.

One might think that the philosophy of science would be the place to go to find out what science is, but philosophers of science these days seem to be more interested in questions such as the following (the first two of which are the closest to our question):

- What is a scientific theory?
  (Here, the emphasis is on the meaning of the term 'theory'.)
- What is scientific explanation?
  (Here, the emphasis is on the meaning of the term 'explanation'.)
- What is the role of probability in science?
- What is the nature of induction? (Why) will the future resemble the past?
- What is a theoretical term?
  (That is, what do the terms of (scientific) theories mean? Do they necessarily refer to something in the real world? For example, there used to be a scientific concept in the theory of heat called 'phlogiston', but we no longer think that this term refers to anything.)
- How do scientific theories change? When they do, are their terms "commensurable"—that is, do they mean the same thing in different theories?
  (For example, what is the relationship between 'phlogiston' and 'heat'? Does 'atom', as used in ancient Greek physics, or even 19th-century physics, mean the same as 'atom' as used in 21st-century physics?)
- Are scientific theories "realistic" (do they attempt to describe the world?) or merely "instrumental" (are they just very good predicting-devices that don't necessarily bear any obvious resemblance to reality, as sometimes seems to be the case with our best current theory of physics, namely, quantum mechanics)?

And so on.

These are all interesting and important questions, and it is likely that a good answer to our question, "What is science?", will depend on answers to many of these. If so, then a full answer will be well beyond our present scope, and the interested reader is urged to explore a good book on the philosophy of science (such as those listed in the Readings and Further Sources of Information sections of this chapter). Here, we will only be able to consider a few of these questions.

4.4 Early Modern Science

Sir Francis Bacon (1561–1626), who lived about 400 years ago (a contemporary of Shakespeare), devised one of the first "scientific methods". He introduced science as a *systematic study*. (So, when you read about computer scientists who call CS a "study" rather than a "science", maybe they are not trying to deny that CS is a science but are merely using a euphemism.) Bacon...

> told us to ask questions instead of proclaiming answers, to collect evidence instead of rushing to judgment, to listen to the voice of nature rather than to the voice of ancient wisdom. (Dyson, 2011a, p. 26)

He emphasized the importance of "replicability":

> Replicability begins with the idea that science is not private; researchers who make claims must allow others to test those claims. (Wainer, 2012, p. 358)

Perhaps science is merely *any systematic* activity, as opposed to a *chaotic* one. There is a computer program called 'AlphaBaby', designed to protect your computer from young children who want to play on your computer but who might accidentally delete all of your files while randomly hitting keys. AlphaBaby's screen is blank; when a letter or numeral key is hit, a colorful rendition of that letter or numeral apears on the screen; when any other key is hit, a geometric figure or a photograph appears. Most children hit the keys randomly ("chaotically") rather than systematically investigating which keys do what ("scientifically").

(Williamson, 2011) suggests something similar when he characterizes the "scientific spirit" as "emphasizing values like curiosity, honesty, accuracy, precision and rigor". And the magician and skeptical investigator known as The Amazing Randi said: "Science, after all, is simply a logical, rational and careful examination of the facts that *nature* presents to us" (quoted in (Higginbotham, 2014, p. 53, my italics)). Although Shapiro would be happy with the word 'nature' here (§3.9.3), others might not be, but I think that it can be eliminated without loss of meaning and still apply to computer "science". (For further discussion of this aspect of science, in the context of whether both philosophy and CS are sciences, see §3.17, Question 9.)

To study something, X, systematically is:

- to find positive and negative instances of X—to find things are *are* Xs and things that are *not* Xs;
- to make changes in Xs or their environment (that is, to do experiments);
- to observe Xs and to observe the effects of experiments performed with them;
- to find correlations between Xs, their behavior, and various aspects of their environment.

One important question in the history of science has concerned the nature of these correlations. Are they (merely) *descriptions*, or are they *explanations*? In other words, is the goal of science to describe the world, or is it to explain the world?

4.5 The Goals of Science

At least three different things have been identified as the goals of science: *description*, *explanation*, and *prediction*. They are not independent of each other: At the very least, you need to be able to describe things in order to explain them or to predict their behavior. But they are distinct: A theory that predicts doesn't necessarily also explain (for some examples, see (Piccinini, 2015, p. 94)).

4.5.1 Description as the Goal of Science

Ernst Mach (1838–1916) was a physicist and philosopher of science who lived about 130 years ago, at the time when the atomic theory was being developed. He was influenced by Einstein's theory of relativity and is probably most famous for having investigated the speed of sound (which is now measured in "Mach" numbers, "Mach 1" being the speed of sound).

For Mach, the goal of science was to discover regular patterns among our sensations in order to enable the prediction of future sensations, and then to *describe* those patterns in an efficient manner. Scientific theories, he argued, are (merely) *shorthand—or summary—descriptions* of how the world *appears* to us.

According to the philosophy of science known as "physicalism", our sensory perception yields reliable (but corrigible)[3] knowledge of ordinary, medium-sized physical objects and events. For Mach, because atoms were not observable, there was no reason to think that they exist. Perhaps it seems odd to you that a physicist would be interested in our *sensations* rather than in the *world* outside of our sensations. This makes it sound as if science should be done "in the first person, for the first person", just like philosophy! That's almost correct; many philosophically oriented scientists at the turn of the last century believed that science should begin with observations, and what are observations but our sensations? The great 18th-century philosopher Immanuel Kant distinguished between what he called 'noumena' (or "things in themselves") and what he called 'phenomena' (or things as we perceive and conceive them as filtered through our conceptual apparatus). He claimed that we could only have knowledge about phenomena, not noumena, because we could not get outside of our first-person, subjective ways of conceiving and perceiving the world. This is why some philosophers of science have argued that sciences such as quantum mechanics are purely instrumental and only concerned with prediction, rather than being realistic, or concerned with the way the world "really" is.

We'll come back to this in §17.3.2 when we discuss the relation of computer programs to the world. For further discussion, see Becker 2018.

[3] That is, "correctable".

4.5.2 Explanation as the Goal of Science

By contrast, the atomic theory was an attempt to *explain* why the physical world appears the way it does. Such a goal for science is to devise theories that explain observed behavior. Such theories are not merely descriptive summaries of our observations, but go beyond our observations to include terms that refer to things (like atoms) that we might not be able to observe (yet). So, the task of science is not, in spite of Mach, merely to *describe* the complexity of the world in simple terms, but to *explain* the world:

> This is the task of natural science: to show that the wonderful is not incomprehensible, *to show how it can be comprehended*...
> (Simon, 1996b, p. 1, my italics)

One major theory of the nature of scientific explanation is the philosopher Carl Hempel's Deductive-Nomological Theory (Hempel, 1942, 1962). It is "deductive", because the statement (Qc) that some object c has property Q is explained by showing that it can be validly deduced from two premises: that c has property P (Pc) and that all Ps are Qs ($\forall x[\mathrm{P}x \rightarrow \mathrm{Q}x]$). And it is "nomological", because the fact that all Ps are Qs is lawlike or necessary, not accidental: Anything that is a P *must* be a Q. (This blending of induction and deduction is a modern development; historically, Bacon (and other "empiricists", chiefly in Great Britain) emphasized experimental "induction and probabilism", while Descartes (and other "rationalists", chiefly on the European continent) emphasized "deduction and logical certainty" (Uglow, 2010, p. 31).)

One of the paradoxes of explanation (it is sometimes called the "paradox of analysis") is that, by showing how something mysterious or wonderful or complicated is really just a complex structure of simpler things that are non-mysterious or mundane, we lose sight of the original thing that we were trying to understand or analyze. (We will see this again in Chapter 7 when we look at Dennett's notion of Turing's "inversion". It is also closely related to the notion of recursion (see §7.5.3 in Chapter 7), where complex things are defined in terms of simpler ones.) Simon demurs:

> ... the task of natural science... [is] to show how it [the wonderful] can be comprehended—*but not to destroy wonder. For when we have explained the wonderful, unmasked the hidden pattern, a new wonder arises at how complexity was woven out of simplicity.*
> (Simon, 1996b, pp. 1–2, my italics)

So, for instance, the fact—if it is a fact (we will explore this issue in Chapter 19)—that non-cognitive computers can exhibit (or even merely simulate) cognitive behaviors is itself something worthy of wonder and further (scientific) explanation.

Question for the Reader:
Are some computer programs theories? In particular, consider an AI program that allows a robot to "see" or to use natural language. Does such a program constitute a psychological (hence scientific) *theory* of vision or language? If so, would it be a *descriptive* theory or an *explanatory* one? (We'll look at some answers to these questions in Ch. 15: "Are Computer Programs Theories?".)

4.5.3 Prediction as the Goal of Science

> ... prediction is always the bottom line. It is what gives science its empirical content, its link with nature. ... This is not to say that prediction is the *purpose* of science. It was once ... when science was young and little; for success in prediction was ... the survival value of our innate standards of subjective similarity. But prediciton is only one purpose among others now. A more conspicuous purpose is technology, and an overwhelming one is satisfaction of pure intellectual curisity—which may once have had its survival value too. —Willard van Orman (Quine, 1987, p. 162)

Einstein "thought the job of physics was to give a complete and intelligible account of ... [the] world" (Holt, 2016, p. 50)—that is, to *explain* the world. Both scientific descriptions and explanations of phenomena enable us to make *predictions* about their future behavior. This stems, in part, from the fact that scientific descriptions must be *general* or *universal* in nature: They must hold for all times, including future times. As the philosopher Moritz Schlick put it,

> For the physicist ... the absolutely decisive and essential thing, is that the equations derived from any data **now** also hold good of *new* data.
> —Schlick, "Causaltiy in Contemporary Physics" (1931), as quoted in (Coffa, 1991, p. 333, my boldface, Schlick's italics)

Thus, "[t]he 'essential characteristic' of a law of nature 'is the *fulfillment of predictions*' " (Coffa, 1991, p. 333, embedded quotation from Schlick).

According to Hempel (1942, §4), prediction and explanation are not mutually exclusive. In fact, Hempel argues that they are opposite sides of the same coin. As we saw in the previous section, to *explain* an event is to find (perhaps abductively) one or more "initial conditions" (usually, earlier events) and one or more general laws such that the even to be explained can be deduced from them. For Hempel, to *predict* an event is to use already-known initial conditions and general laws to deduce a future event:

> The customary distinction between explanation and prediction rests mainly on a pragmatical difference between the two: While in the case of an explanation, the final event is known to have happened, and its determining conditions have to be sought, the situation is reversed in the case of a prediction: here, the initial conditions are given, and their "effect"—which, in the typical case, has not yet taken place—is to be determined. —Hempel (1942, p. 38)

But some scientists and philosophers hold that prediction is the *only* goal that is important, and that description and explanation are either not important or impossible to achieve. One of the main reasons for this comes from quantum mechanics. Some aspects of quantum mechanics are so counter-intuitive that they seem to fail both as descriptions of reality as we think we know it and as explanations of that reality: For example, according to quantum mechanics, objects seem to be spread out rather than located in a particular place—until we observe them; there seems to be "spooky" action at a distance (quantum entanglement); and so on. Yet quantum mechanics is the most

successful scientific theory (so far) in terms of the predictions it makes. Niels Bohr (one of the founders of quantum mechanics) said "that quantum mechanics was meant to be an *instrument for predicting* our observations", neither a description of the world, nor an explanation of it (Holt, 2016, p. 50, my italics).

The explanation-vs.-prediction debate underlies another issue: Is there a world to be described or explained? That is, does science tell us what the world is "really" like, or is it just an "instrument" for helping us get around in it?

Further Reading:
For a discussion of prediction as the goal of science, in the context of trusting what science has to tell us about climate and about solar eclispses, see Gillis 2017. For a cultural critic's views on what we can learn about the nature of science from the paradox of quantum entanglement in physics, see Adam Gopnik 2015b. For more on quantum mechanics, see Weinberg 2017; Albert 2018.

4.6 Instrumentalism vs. Realism

Here's a simplified way of thinking about what a scientific theory is: We can begin by considering two things: the *world* and our *beliefs about* the world (or our descriptions of the world). Those beliefs or descriptions are theories—theories about the world, about what the world is like. Such theories are *scientific* if they can be tested by empirical or rational evidence, in order to see if they are "good" beliefs or descriptions, that is, beliefs or descriptions that are true (that correspond to what the world is really like). The testing can take one of two forms: confirmation or refutation. A theory is confirmed if it can be shown that it is consistent with the way the world really is. And a theory is refuted if it can be shown that it is not the way the world really is.

A picture might help:

Line *W*—a continuous line—is intended to represent the world, a continuum. Line *O*—a line with gaps—is intended to represent observations that we can make about the world: Some parts of the world we have observed (or we can observe)—they are represented in *O* by the line segments. Others we have not observed (or we cannot observe)—those are the gaps. The solid lines in *O* represent things that we believe about the world; the gaps represent things that we don't know (yet) about the world. Line *T* is intended to represent a scientific theory about the world (about line *W*); here, the gaps are filled in. Those fillings-in are predictions about what the world is like at those locations where we cannot observe it; they are guesses (hypotheses) about the world.

Suppose we have an explanatory scientific theory of something, say, atomic theory. Such theories, as we have seen, often include "unobservables"—terms referring to things that we have not (yet) observed but whose existence would help explain things that we *have* observed. One way of looking at this is to think of an experiment as taking some input (perhaps some change deliberately made to some entity being studied) and observing what happens after the experiment is over—the output of the experiment. Between the input and the output, something happens, but we don't necessarily know what it is. It is as if what we are studying is a "black box", and all we can observe are its inputs and outputs. A scientific theory (or, for that matter, a computer algorithm!) can be viewed as an explanation of what is going on inside the black box. Can it be viewed merely as a *description* of what's going on inside? Probably not, because you can only describe what you can observe, and, by hypothesis, we can't observe what's going on inside the black box. Such a theory will usually involve various *unobservables* structured in various ways.

Do the unobservables that form part of such an explanatory theory really exist? If you answer 'yes', then you are a "realist"; otherwise, you are an "instrumentalist". A realist believes in the real existence of explanatory unobservables. An instrumentalist believes that they are merely useful tools (or "instruments") for making predictions.

The debate between realism and instrumentalism is as old as science itself. Galileo (1564–1642) . . .

> . . . and the Church came to an implicit understanding: if he would claim his work only as "*istoria,*" and not as "*dimonstrazione,*" the Inquisitors would leave him alone. The Italian words convey the same ideas as the English equivalents: a new story about the cosmos to contemplate for pleasure is fine, a demonstration of the way things work is not. You could calculate, consider, and even hypothesize with Copernicus. You just couldn't believe in him. —Adam Gopnik (2013, p. 107)

In Mach's time, it was not clear how to treat the atomic theory. Atoms were clearly of instrumental value, but there was no observable evidence of their existence. But they were so useful scientifically that it eventually became unreasonable to deny their existence, and, eventually, they were observed. In our time, black holes have moved from being "merely" theoretical entities to being considered among the denizens of the universe, despite never having been observed directly. (Arguably, there is only circumstantial evidence for them (Bernstein and Krauss, 2016)—recall our discussion in §2.6.1.3, and see §4.9.1.1, below.)

Quantum mechanics poses a similar problem. If the world really is as quantum mechanics says that it is, then the world is really weird. But quantum mechanics is our best current theory about how the world is, So, possibly quantum mechanics is merely a useful calculating tool for scientific prediction and shouldn't be taken literally as a description of the real world.[4]

[4]Besides being opposed to realism, instrumentalism can also be opposed to one kind of understanding (which is closely related to explanation). There are (at least) two ways to understand something: You can understand something in terms of something else that you are more familiar with ("semantic understanding"), and you can understand something in terms of itself, by being very familiar with it directly ("syntactic understanding"). (For more on this distinction, see Ch. 22 and (Rapaport, 1995).) The physicist Jeremy Bernstein has said that there is "a misguided but humanly understandable desire to explain quantum mechanics by

Can an instrumentalist theory evolve into a realist one?:

> Though Galileo ... wants to convince ... [readers] of the importance of looking for yourself, he also want to convince them of the importance of *not* looking for yourself. The Copernican system is counterintuitive, he admits—the Earth certainly doesn't seem to move. It takes intellectual courage to grasp the argument that it does. —Adam Gopnik (2013, p. 107)

So, just as the Copernican theory, initially proposed merely as an instrumentalist claim, became a realist-explanatory theory, so, eventually, the quantum-mechanical view of the world may come to be accepted as a realist description.

Further Reading:
Fine 1986 is a "state of the art" survey article on realism vs. instrumentalism in science, at least as of 1986.

Indeed, the great 20th-century philosopher Willard van Orman Quine, in his classic paper "Two Dogmas of Empiricism" offered this instrumentalist statement:

> As an empiricist I continue to think of the conceptual scheme of science as a *tool*, ultimately, *for predicting future experience* in the light of past experience. —(Quine, 1951, p. 44, my italics)

And what about the "real world"? In an earlier paper, "On What There Is", he argued that "to be is to be the value of a bound variable" (Quine, 1948). In other words, if your best theory talks about Xs—that is, postulates the existence of Xs by quantifying over them—then, according to that theory, Xs exist. But, in light of his instrumentalism in the later paper, he made what seems to me to be a more controversial claim. The very next passage after his remarks about prediction (above) is this statement about whether the values of the "bound variables" of a theory "really" exist:

> Physical objects [that is, what we might think of as "external reality"] are conceptually imported into the situation as *convenient intermediaries ... irreducible posits comparable, epistemologically, to the gods of Homer.* For my part I do, qua lay physicist, believe in physical objects and not in Homer's gods; and I consider it a scientific error to believe otherwise. But in point of epistemological footing the physical objects and the gods differ only in degree and not in kind. Both sorts of entities enter our conception only as cultural posits. The myth of physical objects is epistemologically superior to most in that it has proved more efficacious than other myths as a device for working a manageable structure into the flux of experience. —(Quine, 1951, p. 44, my bracketed interpolation and italics)

something else—something more familiar. But if you believe in quantum mechanics there is nothing else" (Bernstein and Holt, 2016, p. 62). On Bernstein's instrumentalist view, quantum mechanics can only be understood in terms of itself.

Something like quantum mechanics that can only be understood in terms of itself and not in terms of anything else is a kind of "base case" of understanding. (Things that can be understood in terms of something else are a kind of "recursive" case of understanding. We'll discuss recursion in §7.5.3.) However, other things might be able to be understood in terms of quantum mechanics: Recent research in cognitive science suggests that quantum-mechanical methods applied at the macrosopic level might provide better explanations of certain psychological findings about human cognition than more "standard" methods (Wang et al., 2013).

4.7 What Is a Scientific Theory?

It is important to distinguish between the everyday sense of 'theory' and the scientific sense. In the *everyday sense*, a "theory" is merely an idea; it may or may not have any evidence to support it. In this everyday sense, 'theory' is constrasted with 'fact'.[5] In the *scientific sense*, a "theory" is a set of statements (1) that describe, explain, or predict some phenomenon, often formalized mathematically or logically (or even computationally, as we'll see in Chapter 15), *and* (2) that are grounded in empirical or logical evidence. (Note that both 'theory' and 'theorem' are etymologically related.) To be "scientific", a theory must be accompanied by confirming evidence, *and* (as we'll see in §4.9.1) its statements must be precise enough to be capable of being falsified.

Anti-evolutionists (both creationists as well as advocates of "intelligent design") sometimes criticize the scientific theory of evolution as "merely a theory". Anyone who does so is confusing the everyday sense (in which 'theory' is opposed to 'fact') with the scientific sense. Evolution *is* a theory in the scientific sense.

We will return to this topic in Chapter 15, where we will consider whether computer programs can be scientific theories.

Further Reading:
For a wonderful discussion of the nature of science with respect to evolution, see the judge's opinion in the celebrated legal case of Kitzmiller v. Dover Area School District (especially §4, "Whether ID [Intelligent Design] Is Science"), http://ncse.com/files/pub/legal/kitzmiller/highlights/2005-12-20_Kitzmiller_decision.pdf.

Also see Adam Gopnik 2015a for a discussion of the nature of 'theory' as it is used in science and in ordinary language. For a discussion of "Why so many people choose not to believe what scientists say", see Willingham 2011.

Actually, the scientific notion of theory comes in (at least) two varieties: syntactic and semantic. We have alredy said a few things about what syntax and semantics are, and we will have a lot more to say later on. For now, let's just say that the *syntactic* approach to scientific theories focuses on an axiomatic treatment of linguistic *sentences*. On this view,

> a theory was conceived of as an axiomatic theory. That means, as a set of sentences, defined as the class of logical consequences of a smaller set, the axioms of that theory. —van Fraassen (1989, p. 220)

By contrast, the *semantic* approach to scientific theories focuses on the *models* that interpret those sentences and that "link their terms with their intended domain" (van Fraassen, 1989, p. 221). Just as there is a syntactic vs. a semantic view of scientific theories, so is there a syntactic vs. a semantic view of computer programs, which we will investigate in Chapter 17.

[5] Philosophers who use the word 'fact' to refer to states of affairs in the world would say that the everyday sense of 'theory' is contrasted with 'factual (or true) statements'.

4.8 "The" Scientific Method

People often talk about "the scientific method". There probably isn't any such thing. First, there are *many* scientific methods of studying something: (Some) biologists and astronomers use (some) different methods from (some) physicists. Second, disciplines besides the natural sciences (notably mathematics and engineering, but also the social sciences and even many of the humanities) also use scientific methods (Blachowicz, 2016), (Ellerton, 2016).

But let's look at one version of a scientific method, a version that is interesting in part because it was described by the mathematician John Kemeny, who was also a computer scientist. (He was the inventor of the BASIC computer programming language and helped develop time sharing. He also worked with Einstein and was president of Dartmouth College.)

His book *A Philosopher Looks at Science* presents the scientific method as a cyclic procedure (Kemeny, 1959, Chs. 5, 10). Because cyclic procedures are called 'loops' in computer programming, I will present Kemeny's version of the scientific method as an infinite loop (an algorithm that does not halt):

Algorithm Scientific-Method[6]
begin
while there is a new fact to observe, **do:**
{That is, **repeat until** there are no new facts to observe.
This will never happen, so we have *permanent inquiry*}
begin

1. *observe* things & events;
   {Express these observations as descriptive statements about particular objects a, b, ...:
   $$\mathrm{P}a \rightarrow \mathrm{Q}a, \quad \mathrm{P}b \rightarrow \mathrm{Q}b, \quad \ldots$$
   Observations may be "theory-laden", that is, based on assumptions}
2. *induce* general statements;
   {make summary descriptions, such as: $\forall x[\mathrm{P}x \rightarrow \mathrm{Q}x]$}
3. *deduce* future observations;
   {make predictions, such as: $\mathrm{P}c / \therefore \mathrm{Q}c$}
4. *verify* predictions against observations;
   {**if** $\mathrm{Q}c$
   then general statement is confirmedor is consistent with theory
   else revise theory (or ...)}

end
end.

[6]This "algorithm" is written in informal pseudocode. Terms in **boldface** are control structures. Expressions in {braces} are comments.

Kemeny's version of the scientific method is a cycle (or "loop") consisting of observations, followed by inductive inferences, followed by deductive predictions, followed by verifications.[7] The scientist begins by making individual observations of specific objects and events, and describes these in language: Object *a* is observed to have property P, object *a* is observed to have property Q, object *a*'s having property P is observed to precede object *a*'s having property Q, and so on. Next, the scientist uses inductive inference to infer from a series of events of the form P*a* precedes Q*a*, P*b* precedes Q*b*, etc., that whenever any object *x* has property P, it will also have property Q. So, the scientist who observes that object *c* has property P will deductively infer (that is, will predict) that object *c* will also have property Q—*before observing whether it does or not*. The scientist will then perform an experiment to see whether Q*c*. If Q*c* is observed, then the scientist's theory that $\forall x[\mathrm{P}x \rightarrow \mathrm{Q}x]$ will be verified; otherwise, the theory will need to be revised in some way (as we suggested in §2.6.1.3; we'll discuss it in more detail in §4.9.1). For Kemeny, an observation is explained by means of a deduction from a theory, following Hempel's deductive-nomological theory (§4.5.2).

Finally, according to Kemeny,

a discipline is a science if and only if it follows the scientific method.

This rules out astrology, on the grounds that astrologers never verify their predictions.[8]

4.9 Alternatives to "The Scientific Method"

This "hypothetical-deductive method" of "formulating theoretical hypotheses and testing their predictions against systematic observation and controlled experiment" is the classical, or popular, view of what science and the scientific method are. And, "at a sufficiently abstract level", all sciences "count as using" it (Williamson, 2011). But there are at least two other views of the nature of science that—while generally agreeing on the distinctions between science as opposed to art, engineering, and pseudo-sciences such as astrology—differ on the nature of science itself.

4.9.1 Falsifiability

4.9.1.1 Science as Conjectures and Refutations

According to philosopher Karl Popper (1902–1994), the scientific method (as propounded by people like Kemeny) is a fiction. The "real" scientific method sees science as a sequence of *conjectures* and *refutations* (Popper 1953; see also (Popper, 1959)):[9]

1. Conjecture a theory (to explain some phenomenon).

2. Compare its predictions with observations
   (that is, perform experiments to test the theory).

[7] Perhaps a better word than 'verification' is 'confirmation'; we'll discuss this in §4.9.1.

[8] Or that their predictions are so vague that they are always trivially verified. See §4.9.1, below.

[9] For Popper's views on engineering, see Popper 1972, cited in the Further Sources of Information at the end of Chapter 5.

Figure 4.1: http://www.gocomics.com/calvinandhobbes/2012/04/20, © 1992 Watterson

3. **If** an observation differs from a prediction,
   then the theory is *refuted* (or *falsified*)
   else the theory is *confirmed*.

It is important to note that 'confirmed' *does not mean "true"*! Rather, it means that we have evidence that is consistent with the theory (recall our discussion of the coherence theory of truth in §2.4.2)—that is, the theory is *not yet falsified*! This is because there might be some *other* explanation for the predicted observation. Just because a theory T predicts that some observation O will be made, and that observation is indeed made, it does not follow that the theory is true! This is because argument (**A**):

(A) $O,\ (T \rightarrow O) \nvdash_D T$

is an invalid argument (called the Fallacy of Affirming the Consequent). If O is true, but T is false, then the second premise is still true, so we could have true premises and a false conclusion. This might also be called the fallacy of circumstantial evidence, where O is the circumstantial evidence that could support T, but there might be *another* theory that *also* predicts O and which *is* true. (Recall our discussion of this in §2.6.1.3.)

So, what is science according to Popper?

A theory or statement is scientific if and only if it is falsifiable.

By 'falsifiable', Popper meant something like "capable of being falsified *in principle*", not "capable of being falsified with the techniques and tools that we *now* have available to us".

For Popper, falsifiability also ruled out astrology (and other superstitions) as a candidate for a scientific theory. It also ruled out Freudian psychotherapy and Marxist economics. The reason why Popper claimed that astrology, etc., were only pseudo-sciences was that they cannot be falsified, because they are too vague. The vaguer a statement or theory is, the harder it is to falsify. As physicist Freeman Dyson once wrote, "Progress in science is often built on wrong theories that are later corrected. It is better to be wrong than to be vague" (Dyson, 2004, p. 16). When I was in college, one of my friends came into my dorm room, all excited about an astrology book he had found that, he claimed, was really accurate. He asked me what day I was born; I said "September 30th". He flipped the pages of his book, read a horoscope to me, and asked

if it was accurate. I said that it was. He then smirked and told me that he had read me a random horoscope, for April 16th. The point was that the horoscope for April 16th was so vague that it also applied to someone born on September 30th! (See Figs. 4.1 and 4.2.)

Further Reading:
For more on the nature of pseudo-science, see Gordin 2012; Pigliucci and Boudry 2013a,b; Curd 2014. On astrology, see Thagard 1978. On Marxist economics, see Hudelson 1980. On Freudian theories, see Grünbaum 1984, Hansson 2015, and http://grunbaum.pitt.edu/wp-content/plugins/downloads-manager/upload/Medicine%20Bibliography%201-13.pdf.

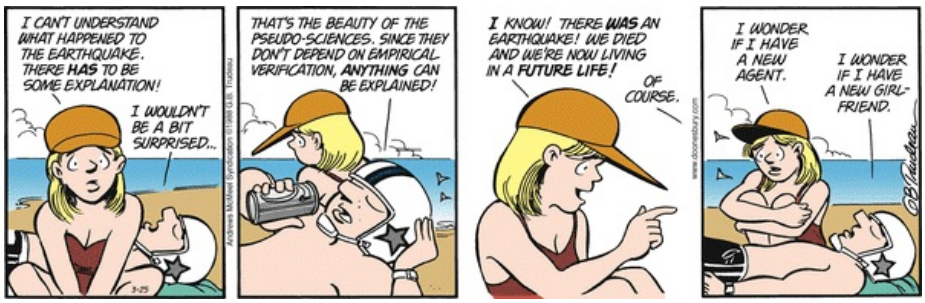


Figure 4.2: http://doonesbury.washingtonpost.com/strip/archive/2017/03/25; ©1988 G.B. Trudeau

4.9.1.2 The Logic of Falsifiability

It is worthwhile to explore the logic of falsifiability a bit more. Although the invalid argument form (**A**), above, seems to describe what goes on, it needs to be made more detailed, because it is not the case that scientists deduce predictions from *theories alone*. There are usually *background beliefs* that are independent of the theory being tested (for example, beliefs about the accuracy of one's laboratory equipment). And one does not usually test a complete theory T but merely one new *hypothesis* H that is being considered as an addition to T. So it is not simply that argument (**A**), above, should have as a premise that theory T predicts observation O. Rather, theory T *conjoined with background beliefs* B, *conjoined with the actual hypothesis* H being tested is supposed to logically predict that O will be observed:

$$(T \ \& \ B \ \& \ H) \rightarrow O$$

Suppose that O is not observed:

$$\neg O$$

What follows from these two premises? By the rule of inference called 'modus tollens', we can infer:

$$\neg(T \ \& \ B \ \& \ H)$$

But, from this, it follows (by DeMorgan's Law) that:

$$\neg T \vee \neg B \vee \neg H$$

That is, either T is false, *or* B is false, *or* H is false, *or any combination of them* is false. What this means is that, if you strongly believe in your theory T that seems to be inconsistent with your observation O, *you do not need to give up* T. Instead, you could give up hypothesis H, or some *part* of T,[10] or (some part of) your background beliefs B (for example, you could blame your measuring devices as being too inaccurate). (Recall our discussion in §2.6.2.1.) As (Quine, 1951) pointed out, you could even give up the laws of logic if the rest of your theory has been well confirmed; this is close to the situation that obtains in contemporary quantum mechanics with the notion of "quantum logic".

Further Reading:

1. On rules of logic such as modus tollens and DeMorgan's Law, see any introductory logic text (such as (Schagrin et al., 1985)), (Rapaport, 1992), or https://en.wikipedia.org/wiki/Propositional_calculus
2. On quantum logic, see http://plato.stanford.edu/entries/qt-quantlog/
3. On the pessimistic meta-induction, see: http://plato.stanford.edu/entries/structural-realism/ and https://en.wikipedia.org/wiki/Pessimistic_induction

4.9.1.3 Problems with Falsifiability

One problem with falisfiability is that not all alleged pseudo-sciences are vague: Is astrology really a Popperian pseudo-science? Although the popular newspaper style of astrology no doubt is (on the grounds of vagueness), "real" astrology, which might be considerably less vague, might actually turn out to be testable and, presumably, falsified, hence falsifiable. But that would make it scientific (albeit false)!

That points to another problem with falsifiability as the mark of science: Are false statements scientific? This is related to the "pessimistic meta-induction" that all statements of science are false. But this isn't quite right: Although it might be the case that any given statement of science that is currently held to be true may turn out to be false, it doesn't follow that all such statements *are* false or will eventually be found to be false. What does follow is that all statements of science are *provisional*:

> Newton's laws of gravity, which we all learn in school, were once thought to be complete and comprehensive. Now we know that while those laws offer an accurate understanding of how fast an apple falls from a tree or how friction helps us

[10] Recall our discussion in §2.6.2.1: T is usually a complex conjunction of claims $A_1, \ldots, A_n$. Consequently, if T is not the case, then at least one of the A_i is not the case. In other words, you need not *give up* a theory; you only need to *revise* it. That is, if prediction O has been falsified, then you only need to give up one of the A_i or H, not necessarily the whole theory T. However, sometimes you *should* give up an entire theory. This is what happens in the case of "scientific revolutions", such as (most famously) when Copernicus's theory that the Earth revolves around the Sun (and not vice versa) replaced the Ptolemaic theory, small revisions to which were making it overly complex without significantly improving it. See §4.9.2, below.

> take a curve in the road, they are inadequate to describe the motion of subatomic particles or the flight of satellites in space. For these we needed Einstein's new conceptions.
>
> Einstein's theories did not refute Newton's; they simply absorbed them into a more comprehensive theory of gravity and motion. Newton's theory has its place and it offers an adequate and accurate description, albeit in a limited sphere. As Einstein himself once put it, "The most beautiful fate of a physical theory is to point the way to the establishment of a more inclusive theory, in which it lives as a limiting case." It is this continuously evolving nature of knowledge that makes science always provisional. (Natarajan, 2014, pp. 64–65)

4.9.2 Scientific Revolutions

Thomas Kuhn (1922–1996), a historian of science, rejected both the classic scientific method and Popper's falsifiability criterion ("a plague on both your houses", he might have said). Based on his studies of the history of science, Kuhn claimed that the real scientific method works as follows (Kuhn, 1962, Ch. 9):

1. There is a period of "normal" science, based on a "paradigm"—roughly, on a generally accepted theory. During that period of normal science, a Kemeny-like or Popper-like scientific method is in operation. (Dyson, 2004, p. 16) refers to the "normal" scientists as "conservatives ... who prefer to lay one brick at a time on solid ground".

2. If that paradigmatic theory is challenged often enough, there will be a "revolution", and a new theory—a new paradigm—will be established, completely replacing the old one. (Dyson, 2004, p. 16) refers to the "revolutionaries" as "those who build grand castles in the air".

3. A new period of normal science follows, now based on the new paradigm, and the cycle repeats.

The most celebrated example was the Copernican revolution in astronomy (Kuhn, 1957). "Normal" science at the time was based on Ptolemy's "paradigm" of an Earth-centered theory of the solar system. But this was so inaccurate that its advocates had to keep patching it up to make it consistent with observations. Copernicus's new paradigm—the heliocentric theory that we now believe—overturned Ptolemy's paradigm.

Other scientific revolutions include those of Newton (who overthrew Aristotle's physics), Einstein (who overthrew Newton's), Darwin (whose theory of evolution further "demoted" humans from the center of the universe), Watson and Crick ("whose discovery of the ... structure of DNA ... changed everything" in biology (Brenner, 2012, p. 1427)), and Chomsky in linguistics (even though some linguists and cognitive scientists today think that Chomsky was wrong (Boden, 2006)).

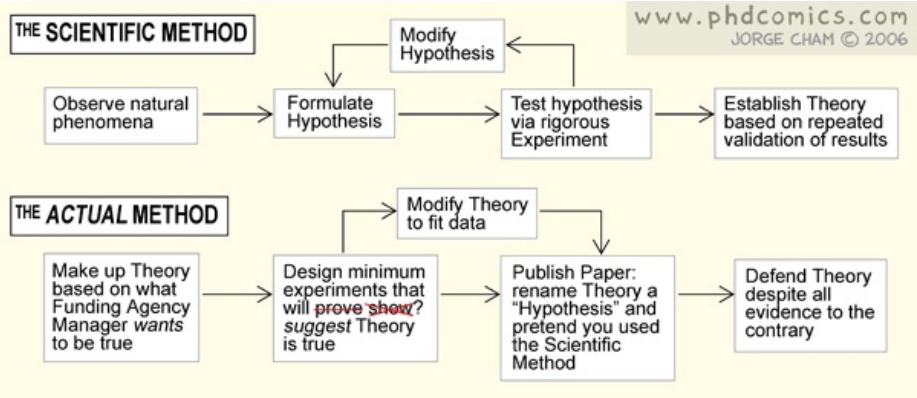


Figure 4.3: http://www.phdcomics.com/comics/archive.php?comicid=761, 16 Sep. 2006, © Jorge Cham

Further Reading:
On scientific revolutions and paradigm shifts, see (Corcoran, 2007) and (Weinberger, 2012).

For a review of the 4th edition (2012) of (Kuhn, 1957), see (Lehoux and Foster, 2012).

On Kuhnian paradigms and philosophy, see (Papineau, 2017).

4.9.3 Other Alternatives

> [T]raditional views about how science is carried out are often idealized or simplistic. Science proceeds in anything but a linear and logical fashion. (Krauss, 2016, p. 85)

Besides the triumverate of Bacon's (or Kemeny's) scientific method, Popper's falsificationism, and Kuhn's scientific revolutions, there are other approaches to the nature of science. For instance, philosopher of science Michael Polanyi argued in favor of science as being "socially constructed", not purely rational or formal (see (Kaiser, 2012) for an overview). And another philosopher of science, Paul Feyerabend, also critiqued the rational view of science, from an "anarchic" point of view (see (Feyerabend, 1975), (Preston, 2012), and compare similar remarks by a computer-scientist-turned-sociologist in (Quillian, 1994, §2.2)).[11] (For a humorous take on the anarchic view, see Figure 4.3.) But exploration of alternatives such as these are beyond our scope.

[11] Quillian's essay is an explanation, in terms of the communication of information, of why the natural sciences are more "effective" than the social sciences. Quillian was one of the early researchers in AI and later became a sociologist of science. Although written in the early days of the World Wide Web, his paper has some interesting implications for the role of social media in political discourse.

4.10 Is CS a Science?

These are only a handful among many views of what science is. Is CS a science according to any of them? This is a question that I will leave to the reader to ponder. But here are some things to consider:

Does CS follow Kemeny's scientific method? For that matter, does *any* science (like physics, chemistry, or biology) really follow it? Does *every* science follow it (what about astronomy or cosmology)? Or is it just an idealized vision of what scientists are supposed to do?

Philosopher Timothy R. Colburn draws an analogy between the scientific method of formulating, testing, and (dis)confirming hypotheses and the problem-solving method of CS consisting of formulating, testing, and accepting-or-rejecting an algorithm (Colburn, 2000, p. 168). Besides suggesting that CS is (at least in part) scientific, this analogizes algorithms to scientific hypotheses or theories. (See Chapter 15 for further discussion.) Even if it is just an idealization, does CS even come close? What kinds of theories are there in CS? How are they tested? If CS is a science, is it "provisional"? Nelson Pole has suggested[12] that "if there is a bug lurking in every moderately complex program, then *all* programs are provisional". Are any computer-science theories ever refuted?

Similarly, (Denning, 2005, p. 28) says that "The scientific paradigm ... is the process of forming hypotheses and testing them through experiments; successful hypotheses become models that explain and predict phenomena in the world." He goes on to say, "Computing science follows this paradigm in studying information processes". For readers who are studying CS, think about your own experiences. Do you agree with Denning that CS follows this scientific method?

Is CS scientific in Kuhn's sense? Have there been any revolutions in CS? Is there even a current Kuhnian paradigm? Here are two issues for you to think about: First, the Church-Turing Computability Thesis identifies the informal notion of computation with formal notions like the Turing machine (as we'll see in more detail in Chapters 7 and 8). "Hypercomputation" is a name given to various claims that the informal notion of computation goes beyond Turing-machine computability. Cockshott and Michaelson 2007, §2.5, p. 235 suggest that the hypercomputation challenges to the Computability Thesis are examples of Kuhnian revolutionary paradigmatic challenges to the "normal" science of CS. (They don't think that the challenges are successful, however. Stepney et al. 2005 offer a long list of paradigms that they think can and should be challenged.) Keep this in mind when you read Chapter 11 on hypercomputation.

Second, two traditions in AI have been logically oriented, knowledge-based AI (sometimes called "Good Old-Fashioned AI", or GOFAI); and connectionist AI, which is based on "artificial neural networks" instead of on logic. Although the former dominated AI research in the early days and, arguably, still has an important role to play (Levesque, 2017; Landgrebe and Smith, 2019a), most AI now is based on the latter. When three connectionist researchers (Geoffrey Hinton, Yann LeCun, and Yoshua Bengio) received the Turing Award, another AI researcher (Oren Etzioni) said, "What we

[12] Private communication, 9 March 2015.

have seen is nothing short of a paradigm shift in the science. History turned their way, and I am in awe" (quoted in Metz 2019b). Keep this in mind when you read Chapter 19 on AI.

And there are other considerations: What about disciplines like mathematics? Math certainly seems to be scientific in some sense, but is it a science like physics or biology? Is CS, perhaps, more like math than like these (other) sciences? This raises another question: Even if CS is a science, what kind of science is it?

4.11 What Kind of Science Might CS Be?

Hempel (1966) distinguished between *empirical* sciences and *non-empirical* sciences.

The former explore, describe, explain, and predict various occurrences in the world. Such descriptions or explanations are empirical statements that need empirical (that is, experimental) support. The empirical sciences include the *natural* sciences (physics, chemistry, biology, some parts of psychology, etc.) and the *social* sciences (other parts of psychology, sociology, anthropology, economics, perhaps political science, perhaps history, etc.). Is CS an empirical science?

The non-empirical sciences are logic and mathematics. Their statements don't need empirical support. Yet they are true of, and confirmed by, empirical evidence (though exactly how and why this is the case is still a great mystery). Is CS a non-empirical science?

CS arose from logic and math. But it also arose from the development of the computer as a tool to solve logic and math problems. (We will explore this twin history of computers and algorithms in Chapter 6.) This brings it into contact with the empirical world and empirical concerns such as space and time limitations on computational efficiency (or "complexity" (Loui, 1996), (Aaronson, 2013b)).

One possible way of adding CS to Hempel's taxonomy is to take a cue from the fact that psychology doesn't neatly belong to just the natural or just the social sciences. So, perhaps CS doesn't neatly belong to just the empirical or just the non-empirical sciences, but that parts of it belong to each. And it might even be the case that the non-empirical aspects of CS are not simply a third kind of non-empirical science, on a par with logic and math, but are themselves parts of both logic and of math.

Or it might be the case that we are barking up the wrong tree altogether. What if CS isn't a science at all? This possibility is what we turn to in the next chapter.

Further Reading:
On "the unreasonable effectiveness of mathematics in science", see Wigner 1960; Hamming 1980a. Wilson and Frenkel 2013 discusses a related question about whether scientists need to study mathematics at all. Burkholder 1999 discusses the difference between "empirical experimental disciplines" (like mechanics, which is a branch of physics) and "*a priori* disciplines" (like mathematics). Tedre and Moisseinen 2014 is a survey of the nature of experiments in science, and whether CS is experimental in nature.

4.12 Questions to Think About

1. Hempel's empirical–non-empirical distinction may be an arbitrary division of a continuous spectrum (§3.3.3.1):

   > The history of science is partly the history of an idea that is by now so familiar that it no longer astounds: the universe, including our own existence, can be explained by the interactions of little bits of matter. *We scientists are in the business of discovering the laws that characterize this matter.* We do so, to some extent at least, by a kind of reduction. The stuff of biology, for instance, can be reduced to chemistry and the stuff of chemistry can be reduced to physics. (Orr, 2013, p. 26, my italics)

   This reductionist picture can be extended at both ends of the spectrum that Orr mentions: At one end,

   > if physics was built on mathematics, so was chemistry built on physics, biology on chemistry, psychology on biology, and … sociology … on psychology …. (Grabiner, 1988, p. 225), citing (Comte, 1830, Vol. I, Ch. 2, Introduction).

   At the other end, mathematics is built on logic and set theory (Quine, 1976) (see Figure 4.4). However, not everyone thinks that this chain of reductions is legitimate (Fodor, 1974).

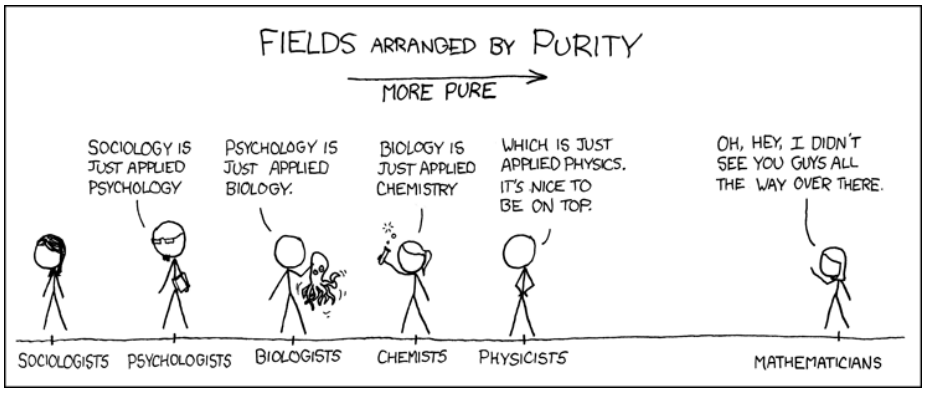


Figure 4.4: ©xkcd.com, http://xkcd.com/435/

Does CS fit into this sequence? If it doesn't, does that mean that it's not part of science? After all, it's not obvious that CS is "in the business of discovering the laws that characterize … matter". We might try to argue that the universe isn't made up of matter, but of information. Then, if you are also willing to say that CS is the science of information (or of information processing), you could conclude that it is a science.

2. (Vardi, 2010) argues that computing (or "computational science") is not a new kind of science or a new way of doing science, but just a more efficient way of doing the same kind of science that humans have always done. Reversing this, George Johnson (2001b) argues that "all science is computer science".

   Try to reconstruct and evaluate their arguments for these propositions.

3. "Web science" is the scientific study of the World Wide Web (Lohr, 2006), (Shneiderman, 2007). But just because something is *called* 'science' doesn't necessarily mean that it *is* a science!

   Is "Web science" a science?
   Whether or not it's a science, how is it related to CS?

4. Read some of the essays cited in §4.9.1.1 that have been critical of the scientific status of disciplines such as Freudian psychoanalysis, economics (Marxist or otherwise!),[13] astrology, etc., and consider whether the arguments that have been used to justify or to challenge their status as a science can be applied to CS.

[13] For non-Marxist economics, you might consider (Rosenberg, 1994), (Leiter, 2004), (Leiter, 2005), (Leiter, 2009), (Chetty, 2013).

4.13 Further Sources of Information

Items in **boldface** are especially important, interesting, or useful.

1. Books and Essays:
   - Chaitin, Gregory J. (1968), "On the Difficulty of Computations", in Gregory J. Chaitin, *Thinking about Gödel and Turing: Essays on Complexity, 1970–2007* (World Scientific Publishing): 3–15, http://www.worldscientific.com/doi/suppl/10.1142/6536/suppl_file/6536_chap01.pdf
     - See esp. §7, on "classical problems of the methodology of science".
   - Dyson, Freeman (2006, 19 October), "Writing Nature's Greatest Book", *New York Review of Books* 53(16).
     - An interesting book review that discusses some of the origins of science. In particular, take a look at fifth and sixth paragraphs, which discuss Galileo and Descartes.
   - Giere, Ronald N. (1984), *Understanding Scientific Reasoning, Second Edition* (New York: Holt, Rinehart & Winston).
   - Lloyd, Seth (2007), "You Know too Much", *Discover Magazine* (28 April).
     - "Most of our exploding collection of science information is junk."
     - We'll look at Lloyd's views on computers and computation in Chapter 9.
   - McCain, Garvin; & Segal, Erwin M. (1969), *The Game of Science* (Belmont, CA: Brooks/Cole).
   - Reed, Dan (2007), "Computing: All Our Constants Are Variables", *Computing Research News* 19(4) (September): 3.
     - Makes an interesting distinction between science as "natural philosophy" and as *scientia* (= knowledge).
   - Rosenberg, Alex (2000), *Philosophy of Science: A Contemporary Introduction* (London: Routledge).
   - **Simon, Herbert A. (1996), *The Sciences of the Artificial, 3rd edition* (Cambridge, MA: MIT Press).**
     - Ch. 1, "Understanding the Natural and the Artificial Worlds", discusses the nature of understanding by simulation.
   - Steedman, Mark (2008), "On Becoming a Discipline", *Computational Linguistics* 34(1) (March): 137–144.
     - Steedman, a computational linguist, has some interesting things to say on the differences between a discipline such as physics and a discipline such as CS (in general) and computational linguistics (in particular), especially in §1, "The Public Image of a Science".
   - Tedre, Matti (2011), "Computing as a Science: A Survey of Competing Viewpoints", *Minds and Machines* 21(3) (August): 361–387.
2. Websites:
   - "[R]ecommendations on texts ... whose subject is the philosophy of science", AskPhilosophers.org, http://www.askphilosophers.org/question/1515
   - Tedre, Matti (2007), "Introduction to the Course", Lecture Notes in the Philosophy of Computer Science, http://cs.joensuu.fi/~mmeri/teaching/2006/philcs/files/lecture_notes1.pdf

Chapter 5

What Is Engineering?

Version of 5 September 2019; DRAFT © 2004–2019 by William J. Rapaport

> [Engineering is] the art of directing the great sources of power in nature for the use and convenience of man [sic].[1]
> — Thomas Tredgold, 1828; cited in (Florman, 1994, p. 175)

> Engineering ... is a great profession. There is the fascination of watching a figment of the imagination emerge through the aid of science to a plan on paper. Then it moves to realization in stone or metal or energy. Then it brings jobs and homes to men [sic]. Then it elevates the standards of living and adds to the comforts of life. That is the engineer's high privilege.
> — Herbert Hoover (1954),[2]
> http://www.hooverassociation.org/hoover/speeches/engineering_as_a_profession.php

> [T]he scientist builds in order to study; the engineer studies in order to build.
> —Frederick P. Brooks (1996, p. 62, col. 1)

> [S]cience tries to understand the world, whereas engineering tries to change it.
> —Mark (Staples, 2015, p. 2)[3]

[1] For a commentary on this, see Mitcham 2009, cited in the Further Sources of Information at the end of this chapter.

[2] Yes; the 31st President of the US.

[3] See the "Philosophical Digression" in §5.7, later in this chapter.

Figure 5.1: http://www.gocomics.com/nonsequitur/2009/09/22, ©2009 Universal UClick

5.1 Readings:

In doing these readings, remember that this chapter is intended to help us determine whether CS is an engineering discipline.

1. Required:
   - Davis, Michael (2009), "Defining Engineering from Chicago to Shantou", *The Monist* 92(3) (July): 325–338, https://www.researchgate.net/publication/271044960_Defining_Engineering_from_Chicago_to_Shantou
   - Petroski, Henry (2003), "Early [Engineering] Education", *American Scientist* 91 (May-June): 206–209.
   - Loui, Michael C. (1987), "Computer Science Is an Engineering Discipline", *Engineering Education* 78(3): 175–178.
2. Very Strongly Recommended:
   - Davis, Michael (1998), "Introduction to Engineering", Part I of *Thinking Like an Engineer: Studies in the Ethics of a Profession* (New York: Oxford University Press), esp.:
     - Introduction (p. 3)
     - Ch. 1, "Science, Technology, and Values" (pp. 3–17)
     - "Who Is an Engineer?" (from Ch. 2, pp. 25–28)
     - Introduction to Ch. 3, "Are 'Software Engineers' Engineers?" (pp. 31–32)
     - "The Standard Definition of Engineer" (from Ch. 3, pp. 32–34)
     - "Three Mistakes about Engineering" (from Ch. 3, pp. 34–36)
     - "Membership in the Profession of Engineering" (from Ch. 3, pp. 36–37)
   - Petroski, Henry (2008), "Scientists as Inventors", *American Scientist* 96(5) (September-October): 368–371.
     - The first two sections argue that all scientists are sometimes engineers and all engineers are sometimes scientists.
   - Brooks, Frederick P., Jr. (1996), "The Computer Scientist as Toolsmith II", *Communications of the ACM* 39(3) (March): 61–68, http://www.cs.unc.edu/∼brooks/Toolsmith-CACM.pdf

5.2 Can We Define 'Engineering'?

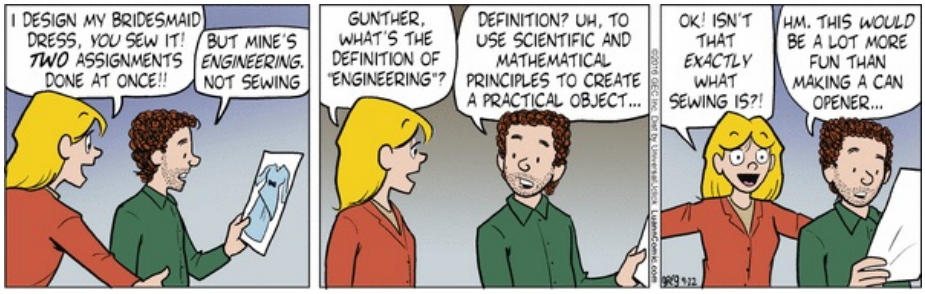


Figure 5.2: http://www.gocomics.com/luann/2016/09/22, ©GEC, Inc.

We began by asking what CS is (Chapter 3), and we considered that it might be what it says it is: a science. So we then looked at what science is (Chapter 4).

We also considered that CS might be a branch of engineering; so now it is time to ask what engineering is. What is the relationship of engineering to science? And what is the relationship of CS to engineering?

The philosophy of engineering is much less well developed than the philosophy of science,[4] and, for some reason, there seem to be fewer attempts to try to define 'engineering'. For instance, if you link to various university websites for schools or departments of engineering, you will rarely find a definition. (Similar points are made by (Koen, 1988, p. 307) and (Pawley, 2009).)

The etymology of 'engineer' is of little help for understanding what engineering is. According to the *Oxford English Dictionary*, 'engineer' comes from 'engine' + '-or' (where '-or' means "agent"), and 'engine', in turn, comes from the Latin '*ingenium*', which had multiple meanings, including "natural disposition", "mental powers", and "clever device"[5]—none of which seems to help: The word has evolved too much for us to be able to figure out what it means from its origins.

Dictionary definitions of 'engineering' are even less helpful than usual. Actually, dictionary definitions are rarely useful: First, different dictionaries don't always agree. Second, some are better than others. Third, dictionaries at best tell you how people *use* a term, but, if people use a term "incorrectly", dictionaries are duty bound to record that.[6] Finally, dictionaries can be misleading: *Webster's Ninth New Collegiate Dictionary* (Mish, 1983, p. 259) defines 'college' as "a body of clergy living together and supported by a foundation"! This may once have been true, and may even still be true in a very limited sense of the term, but why is it listed as the *first* definition? The answer is that Merriam-Webster dictionaries list definitions in *historical order*! So, caution is always advised when citing a dictionary.

[4] Daniel C. Dennett (1995, p. 188) makes a similar observation and singles out the 1969 first edition of Herbert Simon's (1996b) *The Sciences of the Artificial* as a pioneering work in the philosophy of engineering.

[5] http://www.oed.com/view/Entry/62225 and http://www.oed.com/view/Entry/62223

[6] More precisely, if "the meaning" of a word is simply how people use it, then there might be no such thing as an "incorrect" use. Many dictionaries take it as their task merely to record how people use a word, without taking a stand on whether any of those uses are "incorrect".

Nevertheless, it is instructive to see how *Webster's Ninth* defines 'engineering':

1. "The activities or function of an engineer . . . "
2. "The application of science and mathematics . . . [to make] matter and . . . energy . . . useful to people . . . "

The first of these cannot be understood without understanding 'engineer', which is defined thus:

1. "A member of a military group devoted to engineering work."
2. "A person . . . trained in . . . engineering."[7]

Independently of the "military group" condition, both of these definitions of 'engineer' require us to already understand 'engineering'!

As we saw in §3.3.3.1, Richard W. Hamming once pointed out that "the only generally agreed upon definition of mathematics is 'Mathematics is what mathematicians do', which is followed by 'Mathematicians are people who do mathematics' " (Hamming, 1968, p. 4). So this dictionary agrees explicitly with Hamming: Engineering is what engineers do; engineers are people who do engineering!

Only the second definition of 'engineering' in *Webster's Ninth* holds out some hope for independent understanding. Arguably, however, it seems to rule out by definition that CS is engineering, because it is not at all clear that computer scientists "apply science and math to make matter and energy useful". Some might do that (by a stretch of meaning), but surely not all do.

According to the National Research Council's Committee on the Education and Utilization of the Engineer, engineering is, by their definition,

> Business, government, academic, or individual efforts in which knowledge of mathematical and/or natural sciences is employed in research, development, design, maufacturing, systems engineering, or technical operations with the objective of creating and/or delivering systems, products, processes, and/or services of a technical nature and content intended for use.
> (Florman, 1994, pp. 174–175)

Even Florman admits that this is a mouthful! Perhaps it can be simplified to something like this: Efforts in which math and natural science are used in order to produce something useful. If so, then is engineering (merely) applied science?

And Michael Davis, a philosopher of engineering, points out that this definition, because of its vagueness (the overuse of 'and/or'), includes too much (such as accountants, because they use mathematics). He does say that it emphasizes three important "elements" of engineering: the centrality of math and science, the concern with the physical world (which might, therefore, rule out software; but see §12.4, on that topic), and the fact that "unlike science, engineering does not seek to understand the world but to remake it". But he goes on to say that "those three elements . . . do not define" engineering. So, at best, they are necessary but not sufficient conditions (Davis, 1996, p. 98).

[7] For the complete definitions, see (Mish, 1983, p. 412).

Here is another definition-by-committee (note the lists of verbs and nouns):[8]

> **Engineering** is the *knowledge* required, and the *process* applied, to conceive, design, make, build, operate, sustain, recycle or retire, something of significant technical content for a specified purpose;—a concept, a model, a product, a device, a process a system, a technology. (Malpas, 2000, p. 31, my italics)

But it comes down to much the same thing as others have said: designing or building useful things. It emphasizes two aspects to this: One is that the designing or building must be *knowledge-based*. This presumably rules out designing or building that is based, not on scientific knowledge, but on experience alone (what Knuth might call "art"; see §3.14.2). The other aspect is that engineering is a *process*, in the sense of "knowing how" to do something (Malpas, 2000, p. 5). This has an algorithmic flair—after all, algorithms are methods of describing how to do something. (Recall our discussion of this in §3.14.4.)

Finally, Henry Petroski (an engineer) notes that we speak of "the sciences" in the plural (as we do of "the humanities"), but of engineering in the singular, "even though there are many" "engineerings" (Petroski, 2005, p. 304). So determining what engineering is may be as difficult as determining what CS is. More than for science or even CS, it seems that engineering *is* what engineers *do*. In §§5.4 and 5.5, we will consider a variation on this theme—that engineering is what engineers *study*; in §5.6, we will look at what it is that they *do*.

5.3 Could Engineering Be Science?

> The scientist seeks to understand what is; the engineer seeks *to create what never was.* —Theodore von Kármán, cited in (Petroski, 2008a, my italics)[9]

Citing this, (Petroski, 2008a) argues that all scientists are sometimes engineers (for example, when they create a *new theory* that "never was") and that all engineers are sometimes scientists (for example, when they seek to understand how an *existing* bridge works). Could engineering and science be the same discipline? (That would certainly short-circuit the debate about whether CS is one or the other!) Another engineer, Samuel C. Florman, suggested as much (note the italicized phrase!):

> It is generally recognized ... that *engineering is* "*the* art or *science of* making practical application of the knowledge of pure sciences." ... The engineer uses the logic of science to achieve practical results. (Florman, 1994, pp. x–xi, my italics)

One philosopher who has tried to explain engineering—Mario Bunge—also places it among the sciences: First, along with Kemeny (see §4.8), Bunge defines science as any discipline that applies the scientific method. Next, he says that there are two kinds of science: pure and applied. *Pure* sciences apply the scientific method to increasing our knowledge of reality (for example, cell biology). *Applied* sciences apply the

[8] See Ch. 3, §3.17, question 7, footnote 62.

[9] Compare what Brooks said about computers and magic; see §3.14.7.

scientific method to enhancing our welfare and power (for example, cancer research). Among the applied sciences are operations research (mathematics applied to management), pharmacology (chemistry applied to biology), *and engineering* (Bunge, 1974). Given this taxonomy, CS would not necessarily be a *branch* of engineering, though it might be an applied science *alongside* engineering. Yet there is a "pure" component of CS, namely, the mathematical theory of algorithms, computability, and complexity (on the first two, see Chapter 7; on the latter, see (Loui, 1996), (Aaronson, 2013b)).

And Quine said something that suggests that engineering might be a part of science:

> I have never viewed prediction as the main purpose of science,[10] although it was probably the survival value of the primitive precursor of science in prehistoric times. *The main purposes of science are understanding (of past as well as future),* ***technology****, and control of the environment.* (Quine, 1988, my italics and boldface)

If "technology" can be identified with engineering (and why shouldn't it be?—but see §5.6.1), then this puts engineering squarely into the science camp, rendering the science-vs.-engineering debates moot (though still not eliminating the need to ask what engineering—or technology—is).

Further Reading:
On technology, and how it might differ from engineering, see (Bunge, 1974) and (Fiske, 1989).

5.4 A Brief History of Engineering

> Rather than treat software engineering as a subfield of CS, I treat it as an element of the set, {Civil Engineering, Mechanical Engineering, Chemical Engineering, Electrical Engineering, ...}. This is not simply a game of academic taxonomy, in which we argue about the parentage or ownership of the field; *the important issue is the content and style of the education.* (Parnas, 1998, p. 1, my emphasis)

Michael Davis offers an insight into what engineering might be (Davis, 1998).[11] He begins with a history of engineering, beginning some 400 years ago in France, where there were "engines"—that is, machines—and "engineers" who worked with them. These "engineers" were soldiers: either those who used "engines of war" such as catapults and artillery, or those who had been carpenters and stonemasons in civilian life and who continued to ply these trades as soldiers. From this background comes the expression "railroad engineer" and such institutions as the Army Corps of Engineers.

In 1676, the French army created a corps of engineers (separate from the infantry) who were charged with *military* construction. So, at least in 17th-century France, an engineer was someone who did whatever it was that those soldiers did. Forty years later, in 1716, there were *civil* engineers: soldiers who built infrastructure (like bridges and roads) *for civilians.*

A distinction was drawn between engineers and architects. The former were trained in math and physics, and were concerned with reliability and other practical matters.

[10] Recall our discussion of this in Ch. 4, §4.5.3.

[11] (Davis, 1995a) and (Davis, 1996) contain earlier versions of the same considerations.

They were trained as army officers, hence (presumably) more discliplined for larger projects. Architects, on the other hand, were more like artists, chiefly concerned with aesthetics.

Engineers in France at that time were trained at the École Polytechnique ("Polytechnic School"), a university whose curriculum began with a year of science and math, followed gradually by more and more applications to construction (for example, of roads), culminating in a specialization.

So, at this time, engineering was the application of science "for the use and convenience of" people and for "improving the means of production" (Tredgold, as quoted in (Davis, 1998, p. 15)). Engineering was *not* science: Engineers *used* science but didn't *create* new knowledge. Nor was engineering *applied* science: Engineers were concerned with human welfare (and not even with generality and precision), whereas applied scientists are concerned with applying their scientific knowledge.

5.5 Conceptions of Engineering

(Davis, 2011, pp. 31–33) cites four different conceptions of engineering:

1. "engineering as tending engines":

   > This would include railroad engineers and building-superindent engineers. Clearly, neither computer scientists nor software engineers are engineers in this sense, but neither are electrical, civil, mechanical, or chemical engineers.

2. "engineering-as-invention-of-useful-objects":

   > Davis criticizes this sense as both "too broad" (including architects and accountants) and "anachronistic" (applying to inventors of useful objects before 1700, which is about when the modern sense of 'engineer' came into use). Note that this seems to be the sense of engineering used by many who argue that CS *is* engineering; they view engineering as designing and building useful artifacts.

3. "engineering-as-discipline":

   > Here, the issue concerns "the body of knowledge engineers are supposed to learn", which includes "courses concerned with the material world, such as chemistry and statistics". Again, this would seem to rule out both CS and software engineering, on the grounds that neither needs to know any of the "material" natural sciences like chemistry or physics (though both software engineers and computer scientists probably need some statistics) and both need "to know things other engineers do not".

4. "engineering-as-profession":

   > This is Davis's favored sense, which he has argued for in his other writings.

Davis concludes that engineering must be defined by two things: (1) by its professional *curriculum* (by its specific knowledge) and (2) by a professional commitment to use that knowledge consistent with a *code of ethics*. So, rather than saying that engineering is what engineers *do*, Davis says that engineering is what engineers *learn* and how they *ought* (ethically) to use that knowledge. This, of course, raises the question: What is it that engineers learn? Mark Staples[12] observes that Davis's definition of engineering in terms of its curriculum "is circular .... How does engineering knowledge become accepted into engineering curricula?"

There is another question—central to our concerns: Is what engineers learn also something that computer scientists learn? Here, Davis's explicit argument against software engineering (currently) being engineering (and his implicit argument against CS (currently?) being engineering) is that, although both are professions, neither is (currently) part of the profession of engineering as it is taught and licensed in engineering schools. Even CS departments that are academically housed in engineering schools typically do not require their students to take "engineering" courses, their academic programs are not accredited in the same way,[13] nor are their graduates required to become "professional engineers" in any legal senses.

5.6 What Do Engineers Do?

There are two very general tasks that various authors put forth as what engineers do: They *design* things, and they *build* things.

5.6.1 Engineering as Design

The engineer Henry Petroski (Petroski, 2003, p. 206) says that engineering's fundamental activity is *design*. ((Mitcham, 1994) distinguishes between the *engineer* as designer and the *technician* or technologist as builder.) So, engineering is not science, whose fundamental activity is analysis (Petroski, 2003, p. 207), whereas design (along with building) are synthesis activities.

One aspect of design has been picked up by (Hamming, 1968). When one designs something, one has to make choices. Hamming suggests that "science is concerned with *what* is possible while engineering is concerned with *choosing*, from among the many possible ways, *one* that meets a number of often poorly stated economic and practical objectives". This fits well with much of the work—even theoretical work—that is done by computer scientists. As we saw in §§3.6 and 3.7, one definition of CS is that it is concerned with what can be automated (in the sense of "computed"; recall our discussion of this in §3.15.2.1). One way of expressing this is as follows: For what

[12] Personal communication, 2015.

[13] Many of them are accredited, of course, but not as *engineering* curricula.

tasks can there be an algorithm that accomplishes it? But there can be many algorithms all of which accomplish the exact same task. How can we choose among them? We can ask which ones are more efficient: Which use less memory ("space")? Which requires fewer operations (less "time")? So, in addition to asking what can be computed, CS also asks: What can be computed *efficiently*? (As we discussed in §3.15.2.2.) If that is computer *engineering*, so be it, but that would put one of the most abstract, theoretical, and mathematical branches of CS—namely, the theory of computational complexity—smack dab in the middle of computer engineering, and that doesn't seem correct.

Mark Staples[14] points out that, contra Petroski, engineering is more than just design, because architects also design, but are not engineers. (For more on design, see (Denning, 2013a).)

5.6.2 Engineering as Building

We have seen that many people say that what engineers do is to *build* or *create* things. For example, Paul Abrahams argues as follows:

1. Someone who "discover[s] how things work" is a scientist.
2. Someone who "learn[s] how to build things" is an engineer.
3. Therefore, "[c]omputer science is both a scientific discipline and an engineering discipline" (Abrahams, 1987, p. 472).

The conclusion can be made valid by adding two implicit premises:

A. Computer scientists discover how things work.

B. Computer scientists learn how to build things.

Is the argument sound? The explicit premises *seem* to be true. But is premise (1) *really* true? Is life, or the universe, a "thing"? Do scientists really try to learn how the kinds of physical objects that engineers build work (and nothing else)? This seems overly simplistic. Nevertheless, this "analytic vs. synthetic" distinction (that is, a distinction between *analyzing*—taking something apart—in order to learn how it works, on the one hand, and *synthesizing*—putting things together—in order to build something, on the other hand) seems to be a feature of many characterizations of science vs. engineering.

As for implicit premise (A), computer scientists can be said to discover how things work algorithmically. As for (B), computer scientists can be said to build both software (for example, computer programs) and hardware (for example, computers).

Moreover, "engineering ... is an activity that creates things" (Petroski, 2005, p. 304). Note two points: First, it is creative; this is related to claims about engineering as designing and building things. But, second, it is an activity, even grammatically: The word 'engineering' is a gerund—a word that "expresses ... action". Is science also an activity (or is engineering different from science in this respect)? Insofar as science is an activity, it is an activity that produces "knowledge". Engineering is an activity that uses that scientific knowledge to design and build artifacts. Yet one way to discover how things work is to try to build them; so, is all engineering a kind of science?

[14] Personal communication, 2015.

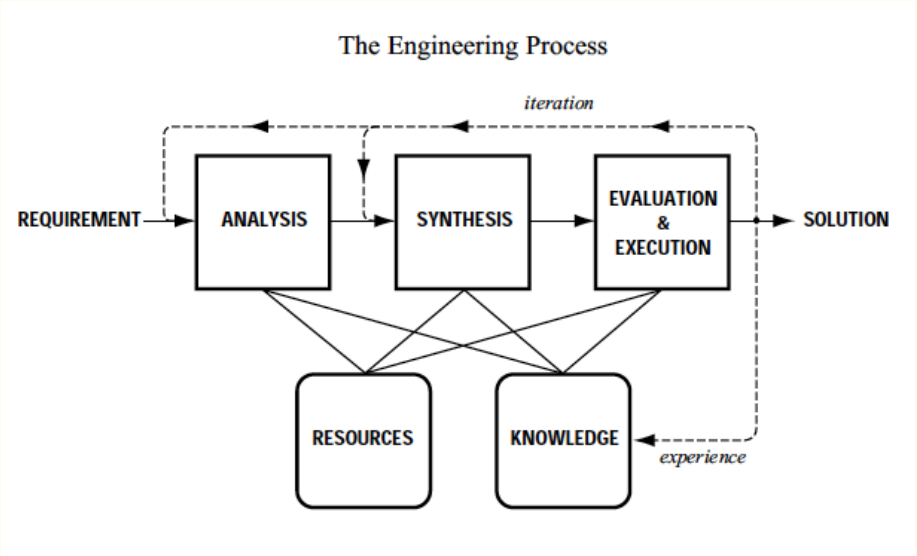


Figure 5.3: Malpas's Engineering Method (Malpas, 2000, p. 35)

5.7 The Engineering Method

Just as there is a "scientific method", some scholars have proposed an "engineering method". Presumably, just as 'science' can be defined as any discipline that follows "the scientific method", so 'engineering' can be defined as any discipline that follows "the engineering method". We saw in Chapter 4 one view of the scientific method, according to which it is a loop that cycles through observation of facts, induction of general statements, deduction of future observations, and verification of the deduced predictions against observations, before cycling back to more observations.

Similarly, (Malpas, 2000) describes the engineering method both linearly and as a cycle. It begins by inputting a set of requirements, followed by analysis, then synthesis, then evaluation and execution, and outputting a solution. The cycle comes in between the input and the output: The evaluation and execution cycles back both to the analysis and to the synthesis, as well as adding to a knowledge base that, along with a set of resources, interact with the analysis, synthesis and evaluation-execution. (See Fig. 5.3.)

But neither this nor the scientific method are carved in stone; they are more like guidelines or even after-the-fact descriptions of behavior rather than rules that must be slavishly followed. Are "engineering methods" significantly different from "scientific methods"? Malpas's engineering method doesn't seem so. (Koen, 2009) seeks a "universal method" (not merely a scientific method or a method used in the humanities); he finds it in the "engineering method", which he identifies with heuristics.

(Koen, 1988) defines the engineering method differently, as:

> the use of engineering *heuristics* to cause the best change in a poorly understood situation within the available resources. (Koen, 1988, p. 308, my italics)

For Koen,

> A heuristic is anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis ... incapable of justification and fallible. It is anything that is used to guide, discover and reveal a possible, but not necessarily, correct way to solve a problem. Though difficult to define, a heuristic has four characteristics that make it easy to recognize: it does not guarantee a solution; it may contradict other heuristics; it reduces the search time for solving a problem; and its acceptance depends on the immediate context instead of on an absolute standard. (Koen, 1988, p. 308).

Further Reading:
Two useful surveys of the various meanings of 'heuristic' are (Romanycia and Pelletier, 1985) and (Chow, 2015).

See also our discussions in §3.15.2.3, and in Chapter 7.

Many other disciplines use heuristics; writers, for example, are often told to "write simply". (See Question 4 at the end of this chapter.)

So, what makes a heuristic an *engineering* heuristic? According to Koen, the first two characteristics differentiate the use of heuristics from science and mathematics. So, they demarcate engineering from science and math. The third and fourth characteristics make their use more practical than at least some scientific or mathematical theories.

(Koen, 1988, p. 309) states "that the *engineering strategy for causing desirable change in an unknown situation within the available resources* and the *use of heuristics* is an absolute identity". First, Koen is saying that what engineers do is to cause changes. This does contrast with science (and math), whose goal is, presumably, to understand things, not to change them, and it is consistent with the quote from Staples cited as an epigraph to this chapter.

Philosophical Digression:
Recall that Marx said that philosophers should change the world, not merely understand it (see §2.6.2.2). Was Marx proposing a discipline of "philosophical engineering"?

Second, Koen's engineering method is not as "formal" as, say, Malpas's, because it is simply the use of heuristics ("the engineering strategy" = "the use of heuristics"). But what kind of heuristics? Much of what Koen says suggests that the kind of heuristic reasoning used by engineers is akin to what Herbert Simon called "bounded rationality" and "satisficing" (which we discussed briefly in §§2.6.1.4 and 3.15.2.3): being satisfied with having a reasonable answer to a question rather than the "correct" one. Bounded rationality is necessary in practical situations, given limits ("bounds") on our time and knowledge. (He offers a partial list, with examples, in the rest of his essay.)

Further Reading:
For a reply to Koen, see (Staples, 2014, §6.2).

5.8 Software Engineering

In addition to the question of whether CS is a kind of engineering, there is the question of the nature of software engineering. Computer scientists (whether or not they consider themselves to be scientists or engineers) often consider software engineering as a branch of CS. Courses in software engineering are often, perhaps even usually, taught in CS departments. But is software engineering engineering?

For Davis, software engineering would be (real?) engineering if and only if there is a professional curriculum for it, along with a code of professional ethics. Interestingly, he also suggests that this might not happen until "real" engineering starts becoming more computational (Davis, 2011, p. 34).

Software engineer David Parnas has a different take on CS's relationship to engineering:

> Just as the scientific basis of electrical engineering is primarily physics, the scientific basis of software engineering is primarily computer science. This paper contrasts an education in a science with an education in an engineering discipline based on the same science. (Parnas, 1998, p. 2).

There are two interesting implications of this. First, it suggests that Parnas views CS as a science, because he takes it to be the scientific basis of a branch of engineering. Second, this view of things is inconsistent with the view advocated by, for instance, Loui and Hartmanis, who take CS (or parts of it) as being a kind of engineering whose scientific basis is primarily mathematics, that is, as mathematical engineering (as we discussed in §3.13). On the other hand, one might argue that if software engineering is based on CS, which, in turn, is based on mathematics, then software engineering must ultimately be based on mathematics, too, which suggests that software engineering would be *mathematical-engineering* engineering!

And that might not be far from the truth, considering that much of *formal* software engineering is based on (discrete) mathematics and logic (such as the formal analysis of computer programs and their development, or the use of program-verification methods in the development of programs; see, for example, (Mili et al., 1986) and our discussion of program verification in Chapter 16). So, is software engineering unique in being a kind of engineering that is based on another kind of engineering rather than on a science? Or is software engineering indeed based on a science, namely CS? Parnas quite clearly believes that CS is a science, *not* an engineering discipline. Why?

Part of the reason concerns his definition of 'engineering': "Engineers are professionals *whose education prepares them* to use mathematics, science, and the technology of the day, *to build products* that are important to the safety and well-being of the public" (Parnas, 1998, p. 2, my italics). This echoes Davis's claim about the central role of *education* in the nature of being an engineer, as well as Brooks's (and others') claim that the purpose of engineering is to *use* science to *build* humanly useful things.

To complete his argument that CS is not engineering, Parnas needs a premise that states that CS education *doesn't* prepare computer scientists to use CS to build things, or perhaps just that computer scientists don't build things. (That leaves open the possibility that CS might be a branch of math or a "technology of the day", but it's pretty clear from the first quote that he thinks that it is a science.) This missing premise is the gist of his entire article. But at least one part of his argument is this: Proper training in software engineering ("designing, building, testing, and 'maintaining' software products" (Parnas, 1998, p. 2)) requires more than a course or two offered in a CS curriculum; rather, it requires an "accredited professional programme ... modelled on programmes in traditional engineering disciplines" (Parnas, 1998, p. 2).[15]

But we still don't have a clear statement as to why he thinks that CS *is* a science and is *not* engineering. As for the latter, it's not engineering, because there is no "rigid accreditation process ... [hence, no] well documented 'core body of knowledge' ... for computer science" (Parnas, 1998, p. 2). Such accreditation might be necessary, but is surely not sufficient: One might force such a core body of knowledge and such an accreditation process on, say, physics, but that wouldn't make physics an engineering discipline.

Some clarity arises here:

> It is clear that two programmes are needed [for example, both physics and electrical engineering, or both computer science and software engineering], not because there are two areas of science involved [for example, physics and electrical engineering], but because there are two very different career paths. *One career path is that of graduates who will be designing products for others to use. The other career path is that of graduates who will be studying the phenomena that interest both groups and extending our knowledge in this area.*
> (Parnas, 1998, p. 3, my emphasis)

So: scientists study phenomena and extend knowledge; engineers design products. So: CS studies phenomena and extends knowledge; software engineers design software products. The distinction between science and engineering, for Parnas, is that between learning and building (Parnas, 1998, p. 4). Note that Parnas agrees with Brooks about the distinction, but draws the opposite conclusion, that CS is *not* engineering!

Questions for the Reader:

1. What phenomena does Parnas think that computer scientists study?
2. Does Parnas consider electrical *engineering* to be an "area of science"!

Further Reading:
For more on software engineering, see (Denning and Riehle, 2009).
On software-engineering education, see (Dewar and Astrachan, 2009).

[15] The spelling in this quote is Canadian-British spelling. 'Programme' is used in the sense of an "academic program", not in the sense of a "computer program".

5.9 Closing Remarks

But a science and an engineering discipline can have the same object: They can be about the same thing. (For example, both chemists and chemical engineers study chemistry.) If so, then what is the common object of computer *science* and computer *engineering*? Is it computers? Algorithms? Information? Perhaps computer *science* studies algorithms and procedures, whereas computer *engineering* studies computers and computer systems. If so, then who studies the relations between these, such as "programmed living machines"? (Recall our discussion of (Newell and Simon, 1976) in §3.7.)

Trying to *distinguish* between science and engineering may be the wrong approach. It is worth recalling W.A. Wulf's cautionary remarks, which we quoted in §3.11:

> Let's remember that *there is only one nature—the division into science and engineering ... is a human imposition, not a natural one*. Indeed, the division is *a human failure*; it reflects *our limited capacity to comprehend the whole.* That failure impedes our progress; *it builds walls* just where the most interesting nuggets of knowledge may lie. (Wulf, 1995, p. 56; my italics)

Is CS a science that tries to understand the world computationally? Or is it an engineering discipline that tries to change the world by building computational artifacts? (Or both? Or neither?) No matter our answer, it has to be the science or engineering (or whatever) *of* something. We have seen at least two possibilities: It studies computers, or it studies computation (algorithms). To further explore which of these might be central to CS, let us begin by asking, "What is a computer?". Later, we will inquire into what computation is.

5.10 Questions to Think About

1. Link to various engineering websites, and try to find a definition of 'engineer' or 'engineering'. Here are two examples:

   (a) "What Is Engineering?",
   Whiting School of Engineering, Johns Hopkins University,
   http://www.jhu.edu/∼virtlab/index.php

   - "This website offers an introduction to basic engineering concepts, allows you to conduct experiments in a virtual laboratory, and describes hands-on projects ... helping you to discover just what engineering is all about."

   (b) "What is engineering and what do engineers do?",
   National Academy of Engineering of the National Academies,
   http://www.nae.edu/About/FAQ/20650.aspx

2. In §3.10, we saw that Brooks argued that CS was not a science, but a branch of engineering, in part because the purpose of engineering is to build things, and that that's what computer scientists do.

   How would you evaluate his argument now that you have thought more deeply about what engineering is?

3. (Loui, 1987, p. 176) said that "The ultimate goal of an engineering project is a product ... that benefits society", giving bridges and computer programs as sample "products". But not all computer programs benefit society—think of computer viruses. Presumably, Loui meant something like "product that *is intended to* benefit society."

   But does that mean, then, that a computer programmer who writes a spreadsheet program *is* an engineer (no matter how sloppily the programmer writes it), whereas a computer programmer who writes a computer virus is *not* an engineer (even if the program was designed according to the best software engineering principles)?

4. If the central feature of engineering is, let's say, the application of scientific (and mathematical) techniques for producing or building something, then surely part of CS is engineering—especially those parts that are concerned with building computers and writing programs. Here's something to think about: Just as (some) computer scientists write programs, so journalists and novelists write essays. Moreover, they use heuristics, such as "write simply", "avoid using the passive voice", and so on. And (Pawley, 2009, p. 310, col. 2) makes a similar point concerning a National Academy of Engineering definition of engineers as "men and women who create new products"; Pawley observes that

> Without knowing how the NAE defines "product," one could argue that an academic who writes a book on how food is portrayed in Victorian novels has created a product (the book) based on abstract ideas (theories about the historical display of food).

Are journalists, novelists, and other writers therefore engineers? Their products are not typically applications of science and math, so perhaps they aren't. But might they not be considered to be, say, language engineers?

5. Evaluate the validity and soundness of the following argument:[16]
   (a) Engineers are cognitive agents who build artifacts for some identifiable purpose.
   (b) Birds build nests for housing their young.
   (c) Beavers build dams because the sound of rushing water annoys them.[17]
   (d) Computer engineers build computers for computation.
   (e) ∴ Birds, beavers, and computer engineers are all engineers.
6. Evaluate the validity and soundness of the following argument:
   (a) Engineers are cognitive agents who build artifacts for some identifiable purpose *and* who know what that purpose is.
   (b) Birds and beavers do not *know* why they build nests and dams, respectively; they are only responding to biological or evolutionary instincts.
   (c) Computer engineers *do* know what the purpose of computation is.
   (d) ∴ Computer engineers are engineers, but birds and beavers are not.
7. 'Design' has a secondary meaning with a slightly negative connotation. Consider the following passage:

   > I am purposely using the word designer instead of animator because [Walt] Disney was always designing things, made designs, and had designs. A designer is someone who indicates with a distinctive mark, and Disney put his mark on everything in his studios. *A designing person is often a crafty person who manages to put his schemes into effect by hook or by crook.* (Zipes, 1995, p. 341, footnote 9, my interpolated brackets and italics)

   Might computer programmers considered as designers in the more-or-less neutral engineering sense also be designers in this other sense? (This is something that we will consider in §18.8.1.)

[16] Thanks to Albert Goldfain for questions 5 and 6.
[17] http://naturealmanac.com/archive/beaver_dams/beaver_dams.html

5.11 Further Sources of Information

1. Books and Essays:

   - Davis, Michael (1995), "Questions for STS [Science & Technology Studies] from Engineering Ethics", talk given at the Society for the Social Study of Science, Charlottesville, VA, October 22, 1995, http://ethics.iit.edu/publication/Questions_for_STS.pdf
     - "What is engineering? What do engineers do? How do engineering decisions get made? What can engineers do?"
   - Florman, Samuel C. (1994), *The Existential Pleasures of Engineering, 2nd edition* (New York: St. Martin's Press).
     - See especially Ch. 14, "The Engineering View", pp. 174–184.
   - Hoare, C.A.R. (2009), "Retrospective: An Axiomatic Basis for Computer Programming", *Communications of the ACM* 52(10) (October): 30–32.
     - Has some interesting comments on the complementary nature of pure academic research (science) and applied industrial research (engineering).
   - Kaag, John; & Bhatia, Sujata K. (2014), "Fools for Tools: Why Engineers Need to Become Philosophers", *The Chronicle [of Higher Education] Review* 61(13) (28 November): B13–B15, http://chronicle.texterity.com/chronicle/20141128b?folio=B13#pg13
     - An essay on engineering ethics.
   - Mitcham, Carl (2009), "A Philosophical Inadequacy of Engineering", *The Monist* 92(3): 339–356.
     - "Engineering is commonly defined as the art or science of 'directing the great sources of power in nature for the use and the convenience of humans' .... But there is nothing in engineering education or knowledge that contributes to any distinct competence in making judgments about what constitutes 'human use and convenience.' Engineering as a profession is analogous to what medicine might be if physicians had no expert knowledge of health or to law if attorneys knew nothing special about justice." (p. 339)
   - Petroski, Henry (2007), "Lab Notes", *American Scientist* 95(2) (March-April): 114–117.
     - How "a theoretician develop[ed] his applied side".
   - Popper, Karl R. (1972), *Objective Knowledge: An Evolutionary Approach* (Oxford: Oxford University Press).
     - Parts of this book discuss the relation of engineering to science.
   - Simons, Peter M.; & Michael, Sir Duncan (eds.) (2009), Special Issue on Philosophy and Engineering, *The Monist* 92(3) (July).
   - Staples, Mark (2014), "Critical Rationalism and Engineering: Ontology", *Synthese* 191(10): 2255–2279, http://www.nicta.com.au/pub?doc=7397
     - Presents a deductive view of theories in engineering, arguing that they "express claims that an artefact ... will perform in a way that satisfies its requirements for use" (§8). Definitions of engineering are discussed in §2. The relation of engineering to science is discussed in §4.

- Staples, Mark (2015), "Critical Rationalism and Engineering: Methodology", *Synthese* 192(1): 337–362, http://www.nicta.com.au/pub?doc=7747
  - A sequel to Staples 2014. Contains a useful "taxonomy" (§3) of ways in which artifacts can fail to conform to specifications (a topic that will also be relevant to our discussion of implementation in Ch. 14).
- Tedre, Matti (2009), "Computing as Engineering", *Journal of Universal Computer Science* 15(8): 1642–1658.
- Vincenti, Walter G. (1990), *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History* (Baltimore: Johns Hopkins University Press).
  - Argues that engineering is a kind of knowledge that is different from scientific knowledge.

Chapter 6

What Is a Computer? A Historical Perspective

Version of 5 July 2019; DRAFT © 2004–2019 by William J. Rapaport

> "Let us now return to the analogy of the theoretical computing machines ... It can be shown that a single special machine of that type can be made to do the work of all. It could in fact be made to work as a model of any other machine. The special machine may be called the universal machine ...
> —Alan Turing (1947), cited in (Davis, 2012)

> "If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence I have ever encountered."
> —Howard Aiken (1956), cited in (Davis, 2012)[1]

> "There is no reason for any individual to have a computer in their home." —Ken Olsen (1974)[2]

> "Many people think that computation is for figuring costs and charges in a grocery store or gas station." (Hill, 2008)

[1] Five years *before* Aiken said this, the Lyons tea company in Great Britain became the first company to computerize its operations (Martin, 2008).

[2] For the citation and history of this quote, see https://quoteinvestigator.com/2017/09/14/home-computer/. That website offers an interesting alternative intepretation: Home *computers* might not be needed if there are home *terminals*, that is, if what is now called "cloud computing" becomes ubiquitous.

6.1 Readings:

1. Highly Desired (if you have enough time to read two full-length books!):
   - Aspray, William (ed.) (1990), *Computing before Computers* (Ames, IA: Iowa State University Press), http://ed-thelen.org/comp-hist/CBC.html
     - This will give you a good background in the "engineering" history of computers.
   - Davis, Martin (2012), *The Universal Computer: The Road from Leibniz to Turing; Turing Centenary Edition* (Boca Raton, FL: CRC Press/Taylor & Francis Group).
     - This will give you a good background in the "logical" history of computers, written by one of the leading mathematicians in the field of theory of computation.
     - At least try to read the Introduction (http://tinyurl.com/Davis00) which is only one page long!
     - See Davis 1987, below, for an article-length version of the story told in this book.
     - Originally published as *Engines of Logic: Mathematicians and the Origin of the Computer* (New York: W.W. Norton, 2000). See (Papadimitriou, 2001), (Dawson, 2001), and (Johnson, 2002b) for reviews of that edition.
2. Required:
   - Browse the linked websites at "A Very Brief History of Computers" http://www.cse.buffalo.edu/∼rapaport/584/history.html
   - O'Connor, J.J., & Robertson, E.F. (1998), "Charles Babbage", http://www-gap.dcs.st-and.ac.uk/∼history/Mathematicians/Babbage.html
   - Simon, Herbert A., & Newell, Allen (1958), "Heuristic Problem Solving: The Next Advance in Operations Research", *Operations Research* 6(1) (January-February): 1–10.
     - Read the brief history of Babbage's work (pp. 1–3); skim the rest.
     - In this paper, Simon and Newell predicted that (among other things) a computer would "be the world's chess champion" (p. 7) within 10 years, that is, by 1968.[3]
     - Adam Smith's pin-factory story (told in Simon & Newell 1958 and discussed below in §6.5.3) is reprinted in:
       - Lawson, Robert (2004), "Division of Labour", http://www.divisionoflabour.com/archives/000006.php
   - Davis, Martin (1987), "Mathematical Logic and the Origin of Modern Computers", *Studies in the History of Mathematics*, reprinted in Rolf Herken (ed.), *Universal Turing Machine: A Half-Century Survey; Second Edition* (Vienna: Springer-Verlag, 1995): 135–158, http://staff.science.uva.nl/∼edoh/COMPHIS/davispaper.pdf
     - Article-length version of Davis 2012.

[3] But it didn't happen till 1997 (https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov). I once asked Simon about this; our email conversation can be found at: http://www.cse.buffalo.edu/∼rapaport/584/S07/simon.txt; see also (Simon, 1977, p. 1191, endnote 1).

- Ensmenger, Nathan (2003), "Bits of History: Review of A.R. Burks's *Who Invented the Computer? The Legal Battle that Changed Computing History*", in *American Scientist* 91 (September-October): 467–468, http://www.americanscientist.org/bookshelf/pub/bits-of-history

3. Very Strongly Recommended:

   - Robinson, J. Alan (1994), "Logic, Computers, Turing, and von Neumann", in K. Furukawa; D. Michie; & S. Muggleton (eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning* (Oxford: Clarendon Press): 1–35.
     - A personal history of the development of computers and the related history of logic, by the developer of the resolution method of automated theorem proving.
   - Browse through:
     - Computer History Museum, http://www.computerhistory.org/
     - Copeland, Jack (2000), "A Brief History of Computing", http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html
     - Lee, J.A.N. (1995–2002), "The History of Computing", http://ei.cs.vt.edu/∼history/
     - Hitmill.com (2004), "History of Computers", http://www.hitmill.com/computers/computerhx1.html
     - Hoyle, Michelle A. (1994–2006), "The History of Computing Science" http://lecture.eingang.org/

6.2 Introduction

Figure 6.1: http://www.gocomics.com/jumpstart/2012/02/13; ©2012 UFS Inc.

Let us take stock of where we are. We began by asking what CS is, and we saw that it might be a science, a branch of engineering, a combination of both, or something *sui generis*. To help us answer that question, we then investigated the nature of science and of engineering.

We also asked, "What is CS the science (or engineering, or study) *of*"? We saw that there are at least three options to consider:

1. The subject matter of CS might be *computers* (the physical objects that compute), as (Newell et al., 1967) suggested; or

2. it might be *computing* (the algorithmic processing that computers do), as (Knuth, 1974b) suggested; or

3. it might be something else (such as the information that gets processed).

In this chapter and Chapter 7, "What Is an Algorithm?", we will begin to examine the first two options. Readers interested in exploring the third option—information—are encouraged to look at some of the readings cited in §3.8.

So our focus now will be to seek answers to the question:

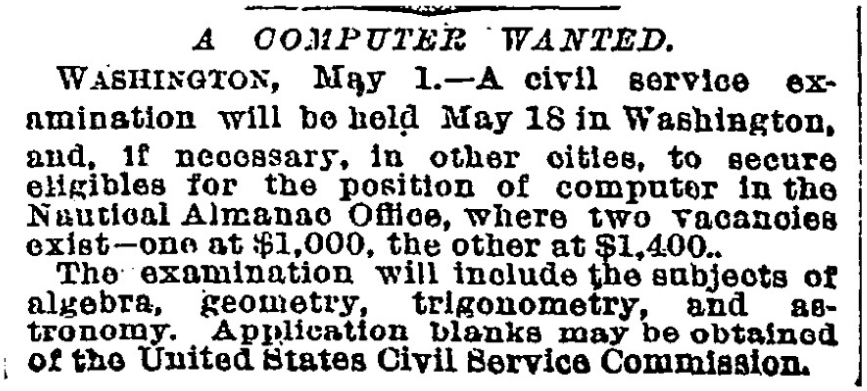
> "What is a computer?"

To help answer this, we will look first at the *history* of computers (in this chapter) and then, in Chapter 9, at some *philosophical* issues concerning the nature of computers.

6.3 Would You Like to Be a Computer? Some Terminology

> Towards the close of the year 1850, the Author first formed the design of rectifying the Circle to upwards of 300 places of decimals. He was fully aware, at that time, that the accomplishment of his purpose would add little or nothing to his fame as a Mathematician, though it might as a Computer; ...
>
> —William Shanks (1853), as cited in Brian Hayes 2014a, p. 342

Let's begin our historical investigation with some terminology. Some 125 years ago, in the May 2, 1892, issue of *The New York Times*, the following ad appeared:



A COMPUTER WANTED.

WASHINGTON, May 1.—A civil service examination will be held May 18 in Washington, and, if necessary, in other cities, to secure eligibles for the position of computer in the Nautical Almanac Office, where two vacancies exist—one at $1,000, the other at $1,400.

The examination will include the subjects of algebra, geometry, trigonometry, and astronomy. Application blanks may be obtained of the United States Civil Service Commission.

The New York Times
Published: May 2, 1892
Copyright © The New York Times

Figure 6.2: http://tinyurl.com/NYT-computer

So, over a century ago, the answer to the question "What is a computer?" was: a human who computes! In fact, until at least the 1940s (and probably the 1950s),[4] that was the meaning of 'computer'. When people wanted to talk about a *machine* that computed, they would use the phrase 'computing *machine*' or (later) '*electronic* (digital) computer'. (In Chapters 8 and 19, when we look at Alan Turing's foundational papers in CS and AI, this distinction will be important.) Interestingly, nowadays when one wants to talk about a *human* who computes, we need to use the phrase '*human* computer' (Pandya, 2013). In this book, for the sake of familiarity, I will use the

[4] And possibly even the 1960s, as in the movie *Hidden Figures* (https://en.wikipedia.org/wiki/Hidden_Figures); see also (Bolden, 2016) and (Natarajan, 2017).

word 'computer' for the machine, and the phrase 'human computer' for a human who computes.

Digression and Further Reading:
For a history of human computers (most of whom were women), see Lohr 2001; Grier 2005; Skinner 2006; Thompson 2019. An interesting website is: "Computer Programming Used to Be Women's Work", *Smart News Blog*,
http://blogs.smithsonianmag.com/smartnews/2013/10/computer-programming-used-to-be-womens-work/

The other kind of human computer, of course, would be mathematicians (of either sex):

> Historians might ... wonder if mathematicians who devised algorithms were programmers .... Modern programmers would ... say no because these algorithms were not encoded for a particular machine.
> —(Denning and Martell, 2015, p. 83)

But they were! They were encoded for humans (human computers)! Curiously, on the very next page, Denning & Martell say exactly that:

> The women who calculated ballistic tables for the Army during World War II were also programmers, although their programs were not instructions for a machine but for themselves to operate mechanical calculators. In effect, they were human processing units.

But why should this be treated merely as a kind of metaphor? These women *were* the computers!

6.4 Two Histories of Computers

There seem to be two histories of computers; they begin in parallel, but eventually converge and intersect:

- The goal of one of these histories was to build a *machine that could compute* (or calculate), that is, a machine that could duplicate—and therefore assist, or even replace, or eventually supersede—human computers. This is an *engineering* goal.

- The goal of the other history was to provide a *foundation for mathematics*. This is a *scientific* (or, at least, a mathematical or logical) goal.

These histories were probably never really parallel but more like a tangled web, with at least two "bridges" connecting them: The first was a person who lived about 330 years ago, and the other was someone who was active much more recently (about 80 years ago)—Gottfried Wilhelm Leibniz (1646–1716) and Alan Turing (1912–1954). (There were, of course, other significant people involved in both histories, as we will see.) Moreover, both histories begin in ancient Greece, the engineering history beginning with the need for computational help for astronomical purposes (including navigation), and the scientific history beginning with Aristotle's study of logic.

6.5 The Engineering History

The engineering history concerns the attempt to create machines that would do certain mathematical computations. The two main reasons for wanting to do this seem to be (1) to make life easier for humans (let a machine do the work) and—perhaps of more importance—(2) to produce computations that are more accurate (both more precise and with fewer errors) than those that humans produce.

It is worth noting that the goal of having a machine perform an intellectual task that would otherwise be done by a human is one of the motivations underlying AI. In this section, we will only sketch some of the highlights of the engineering history.

Further Reading:
In addition to the readings for this chapter (§6.1), see (Goldstine, 1972) (written by one of the early pioneers of computers), (Arden, 1980, pp. 10–13, §"A Brief History"), (Chase, 1980) (an illustrated history of computers, with a useful introduction by the science historian I. Bernard Cohen), (Carlson et al., 1996) (for more detail on the engineering history), (O'Regan, 2008), and (Campbell-Kelly, 2009).

(Sloman, 2002, §2) argues that even the engineering history of computers has "two strands": the "development of machines for controlling physical mechanisms and [the] development of machines for performing abstract operations, e.g. on numbers."

(Husbands et al., 2008) is an overview of attempts to make mechanical minds.

6.5.1 Ancient Greece

The very early history of the attempt to build machines that could calculate can be traced back to at least the second century B.C.E., when a device now known as the Antikythera Mechanism was constructed. This was a device used to calculate astronomical information, possibly for use in agriculture or religion. Although the Antikythera Mechanism was discovered in 1900, a full understanding of what it was and how it worked was not figured out until the 2000s.

Question for the Reader:
In §3.9.5, we asked how someone who didn't know what a computer was would describe a laptop found in the desert. The Antikythera Mechanism is close to a real-life example of the "computer found in the desert".

Does it compute? What does it compute? Is what it computes determined by its creators? Can *we* determine it?

Further Reading:
On the Antikythera Mechanism, see (Freeth, 2006), (Wilford, 2006), (Seabrook, 2007), (Wilford, 2008), and (Freeth, 2009). For a slideshow, see Seabrook, John (2007), "Pieces of History", http://www.newyorker.com/online/2007/05/14/slideshow_070514_antikythera

6.5.2 17th-Century Adding Machines

Skipping ahead almost 2000 years to about 350 years ago, two philosopher-mathematicians are credited with more familiar-looking calculators: Blaise Pascal (1623–1662), who helped develop the theory of probability, also invented an adding (and subtracting) machine that worked by means of a series of connected dials, and Leibniz (who invented calculus, almost simultaneously with, but independently of, Isaac Newton) invented a machine that could add, subtract, multiply, and divide. As we'll see later on, Leibniz also contributed to the scientific history of computing with an idea for something he called a "calculus ratiocinator" (loosely translatable as a "reasoning system").

Further Reading:
"Before electronic calculators, the mechanical slide rule dominated scientific and engineering computation", according to (Stoll, 2006). Note: The slide rule is an *analog* calculator! For more on analog computation, see Rubinoff 1953; Samuel 1953, p. 1224, § "The Analogue Machine"; Jackson 1960; Montague 1960; Pour-El 1974; Moor 1978; Haugeland 1981a; Copeland 1997, "Nonclassical Analog Computing Machines", pp. 699–704; Shagrir 1999; Holst 2000; Piccinini 2004a, 2007d, 2008, 2009, 2010b, 2011, 2012; Care 2007; Zenil and Hernández-Quiroz 2007, especially p. 5; Fortnow 2010; Piccinini and Craver 2011; McMillan 2013; Corry 2017; and §9.7.1, below. For an alternative way to compute with real numbers other than with analog computers, see (Buzen, 2011).

For some good images of early calculating machines, including Pascal's and Leibniz's, see:

1. "Generations of Computers" (blog), http://generationsofcomputers.blogspot.com/2007/12/generation-of-computers.html
2. IBM "Antique Attic", a three-"volume", illustrated exhibit of computing artifacts, http://www-03.ibm.com/ibm/history/exhibits/index.html
3. Tout, Nigel (2000–2012), "Vintage Calculators Web Museum", http://www.vintagecalculators.com/

6.5.3 Babbage's Machines

> We both went to see the *thinking* machine (for such it seems) last Monday.
> —Lady Byron (Ada Lovelace's mother), writing in 1833 about Babbage's Difference Engine (as cited in Stein 1984, p. 38)

Two of the most famous antecedents of the modern electronic computer were due to the English mathematician Charles Babbage, who lived about 185 years ago (1791–1871). The first of the machines he designed was the Difference Engine (1821–1832), inspired in part by a suggestion made by a French mathematician named Gaspard de Prony (1755–1839).

De Prony, who later headed France's civil-engineering college, needed to construct highly accurate, logarithmic and trigonometric tables for large numbers, and was himself inspired by Adam Smith's 1776 text on economics, *The Wealth of Nations*. Smith discussed the notion of the "division of labor": The manufacture of pins could be made more efficient by breaking the job down into smaller units, with each laborer who

worked on one unit becoming an expert at his one job. This is essentially what modern computer programmers call "top-down design" (Mills, 1971) and "stepwise refinement" (Wirth, 1971): To accomplish some task T, analyze it into subtasks $T_1, \ldots, T_n$, each of which should be easier to do than T. This technique can be repeated: Analyze each T_i into sub-subtasks $T_{i_1}, \ldots, T_{i_m}$, and so on, until the smallest sub...subtask is so simple that it can be done without further instruction (this is the essence of "recursion"; see §7.6). De Prony, realizing that it would take him too long using "difference equations" by hand,[5] applied this division of labor to computing the log and trig tables, using two groups of human computers, each as a check on the other.

It should be noted that, besides its positive effects on efficiency, the division of labor has negative ones, too: It "would make workers as 'stupid and ignorant as it is possible for a human creature to be.' This was because no worker needed to know how to make a pin, only how to do his part in the process of making a pin" (Skidelsky, 2014, p. 35), quoting Adam Smith, in *The Wealth of Nations*, Book V, Ch. I, Part III, Article II (https://www.marxists.org/reference/archive/smith-adam/works/wealth-of-nations/book05/ch01c-2.htm). More recently, several writers have pointed out that very few of us know every detail about the facts that we know or the activities that we know how to perform (see, for example, (Dennett, 2017, Ch. 15)). So this negative effect might be unavoidable.

Further Reading:
Babbage was inspired by de Prony, who was inspired by Smith. Smith was, in turn, inspired by the Talmud—the 2500-year-old Jewish commentaries on the Torah. See Kennedy, Gavin (2008, 11 May), "The Talmud on the Division of Labour", *Adam Smith's Lost Legacy* (blog), http://adamsmithslostlegacy.blogspot.com/2008/05/talmud-on-dvision-of-labour.html (note: the misspelling of 'dvision' in that URL is not a typographical error!) and Cowen, Tyler (2008), "Division of Labor in the Babylonian Talmud", *Marginal Revolution* (blog), http://marginalrevolution.com/marginalrevolution/2008/05/division-of-lab.html. See also Stein 1984; Pylyshyn 1992.

The recursive nature of top-down design and stepwise refinement has been identified with the notion of scientific progress by Rosenblueth and Wiener (1945, p. 319, my interpolated brackets): "Scientific progress consists in a progressive opening of ... [closed, that is, "black"] boxes" and subdividing closed boxes into "several samller shut compartments" some of which "may be ... left closed, because they are considered only functionally, but not structurally important."

Babbage wanted a *machine* to replace de Prony's *people*; this was to be his Difference Engine. He later conceived of an "Analytical Engine" (1834–1856), which was intended to be a general-purpose problem-solver (perhaps more closely related to Leibniz's goal for his calculus ratiocinator). Babbage was unable to completely build either machine: The techniques available to him in those days were simply not up to the precision required. However, he developed techniques for what we would today call "programming" these machines, using a 19th-century version of punched cards (based on a technique invented by Joseph Marie Jacquard for use in looms—a sequence of

[5] Difference equations are a discrete-mathematical counterpart to differential equations. They involve taking successive differences of sequences of numbers.

punched cards constituted a "program" for weaving a pattern in the cloth on the loom). Working with Babbage, Lady Ada Lovelace (1815–1852)—daughter of the poet Lord Byron—wrote a description of how to program the (yet-unbuilt) Analytical Engine; she is, thus, considered to be the first computer programmer.

According to Stein 1984, p. 49,

> ... the important difference between the two machines is that the Difference Engine followed an unvarying computational path ..., while the Analytical Engine was to be truly programmable ....

This suggests that the relationship between the Difference Engine and the Analytical Engine was similar to that between a Turing Machine (which can only compute a single function) and a Universal Turing Machine (which can compute any function whose algorithm is stored on its tape).

Further Reading:
Hyman 1982 is a biography of Babbage (reviewed in O'Hanlon 1982). Useful websites on Babbage include:

1. Charles Babbage Institute, http://www.cbi.umn.edu/
2. Lee, J.A.N. (1994), "Charles Babbage", http://ei.cs.vt.edu/~history/Babbage.html
3. Greenemeier, Larry (2008, April 24), "150-Year-Old Computer Brought to Life" (slideshow), http://www.scientificamerican.com/article.cfm?id=150-year-old-computer-babbage
4. Sydell, Laura (2009, December 10), "A 19th-Century Mathematician Finally Proves Himself", http://www.npr.org/templates/story/story.php?storyId=121206408&ft=1&f=1001
5. Johnstone, Adrian (2014), "Babbage's Language of Thought", *Plan 28 Blog* (28 November), http://blog.plan28.org/2014/11/babbages-language-of-thought.html

Robin Gandy (1988, pp. 53, 54), who was Turing's only PhD student, notes that Babbage's Analytic Engine can be considered as a kind of register machine, in which case it is equivalent to a Turing machine, and he considers this statement by Babbage—"the whole of the development and operations of analysis are now capable of being executed by machinery"—to be "Babbage's Thesis" (perhaps on a par with Turing's thesis).

On the Difference Engine, see Park 1996 and Campbell-Kelly 2010. A partial model of the Difference Engine was finally built around 1991 (Swade, 1993), and efforts are underway to build a version of the Analytical Engine (http://www.plan28.org/). For more on the (re-)construction of the Analytical Engine, see Markoff 2011, which contains a diagram of the Analytical Engine, http://www.nytimes.com/interactive/2011/11/07/science/before-its-time-machine.html.

"Was Babbage's Analytical Engine Intended to Be a Mechanical Model of the Mind?"—that question is answered in the negative (at least from Babbage's point of view) in Green 2005.

Ada Lovelace's commentary can be found in her notes to her translation of a description of the Analytic Engine (Menabrea and Lovelace, 1843). For more on Lovelace, see Stein 1984, 1985; Kidder 1985; Kim and Toole 1999; Holt 2001; MacFarlane 2013; Uglow 2018.

For a more historically accurate discussion of the history of programming, see Randell 1994. Lohr 2002 is the story of one of the early computer programmers. For "Reflections on the first textbook on programming", see Campbell-Kelly 2011. Ensmenger 2011 contains "Reflections on recruiting and training programmers during the early period of computing."

For the story of the Jacquard loom, see Keats 2009.

6.5.4 Electronic Computers

The modern history of electronic, digital computers is itself rather tangled and the source of many historical and legal disputes. Here is a brief survey:

1. John Atanasoff (1903–1995) and his student Clifford Berry (1918–1963), working at Iowa State University, built the ABC (Atanasoff-Berry Computer) in 1937–1942. This may have been the first electronic, digital computer, but it was not a general-purpose (programmable) computer, and it was never completed. It was,

however, the subject of a patent-infringement suit, about which more in a moment.

2. Konrad Zuse (1910–1995), in Germany, developed the Z3 computer in 1941, which was programmable.

3. In 1943, the Colossus computer was developed and installed at Bletchley Park, England, for use in cryptography during World War II. Bletchley Park was where Alan Turing worked on cracking the Nazi's code-making machine, the Enigma.

> **Further Reading:**
> For more on Colossus, see: (Sale, nd), (Wells, 2003), and (Copeland et al., 2010).
>
> On the Enigma, see (Kernan, 1990).
>
> (Martin, 2013) is a brief biography of Mavis Batey, a code breaker who worked with Turing at Bletchley Park.

4. Howard Aiken (1900–1973), inspired by Babbage, built the Harvard Mark I computer in 1944; it was designed to compute differential equations.

5. After the war, in 1945, Turing decided to try to implement his "*a*-machine" (what is now called the 'Turing machine'; see §6.6, below, and—for more detail—Chapter 8), and developed the ACE (Automatic Computing Engine) (Copeland, 1999), (Campbell-Kelly, 2012). It was also around this time that Turing started thinking about AI and neural networks.

6. John Presper Eckert (1919–1995) and his student John Mauchly (1907–1980), working at the University of Pennsylvania, built the ENIAC (Electronic Numerical Integrator And Computer) in 1946. This was the first *general-purpose*—that is, programmable—electronic computer. In 1945, with the collaboration of the mathematician John von Neumann (1903–1957)—who outlined an architecture for computers that is still used today—they began to develop the EDVAC (Electronic Discrete Variable Automatic Computer), which used binary arithmetic (rather than decimal). Completed in 1949, it evolved into the first commercial computer: the UNIVAC (UNIVersal Automatic Computer). UNIVAC became famous for predicting, on national TV, the winner of the 1952 US presidential election. The company that made UNIVAC evolved into the Sperry Rand Corporation, which owned the patent rights. The Honeywell Corporation, a rival computer manufacturer, successfully sued Sperry Rand in 1973, on the grounds that Mauchly had visited Atanasoff in 1941, and that it was Atanasoff and Berry—not Eckert and Mauchly—who had "invented" the computer, thus vacating Sperry Rand's patent claims.

Further Reading:
On the ENIAC, see Kennedy 1946; Lohr 1996; Levy 2013. For a short biography of Eckert, see (Baranger, 1995a). For Atanasoff, see (Baranger, 1995b). On Zuse, see (Lee, 1994), (Hyman, 2012), and (Winkler, 2012).

On the ENIAC-ABC controversy, with a discussion of an attempt to replicate the ABC, see (Wheeler, 1997). A useful summary of some of the issues involved can be found in (Ensmenger, 2003), who observes that identifying Atanasoff as "*the* inventor of *the* computer" (my phrasing and italics) is an "answer to what is fundamentally the *wrong question*" (Ensmenger, 2003, italics in original), because "any particular claim to priority of invention must necessarily be heavily qualified: If you add enough adjectives, you can always claim your own favorite". (Note that the subtitle of (Wheeler, 1997) is precisely that question!) For another take on this kind of question, by computer scientist Richard W. Hamming, see (Hamming, 1980b).

(Halmos, 1973) is a very readable, short biography of von Neumann, with a heavy emphasis on the humorous legends that have grown up around him. The story of von Neumann's involvement in the development of computers can be found in (Dyson, 2012b). (And for commentaries on (Dyson, 2012b), see (Holt, 2012) and (Mauchly et al., 2012).) See also (Bacon, 2010). For the original document on the "von Neumann architecture", see (von Neumann, 1945).

6.5.5 Modern Computers

> Where a calculator like ENIAC today is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1000 vacuum tubes and perhaps weigh only $1\frac{1}{2}$ tons. —*Popular Mechanics*, March 1949, cited in Meigs 2012.

> One of our daughters looked at a pile of MacBooks in the living room, and asked, "Can you hand me a computer?" (28 March 2015)

These early computers were large, cumbersome, and expensive, so there weren't very many of them:

> *There are currently over one hundred computers installed in American universities.* Probably two dozen or more will be added this year. In 1955 the number was less than twenty-five. ... [C]onsidering the costs involved in obtaining, maintaining, and expanding these machines, the universities have done very well in acquiring hardware with their limited funds. (Perlis, 1962, p. 181, my italics)

Of course, a university with, say, 5000 students now probably has at least 5000 computers—and probably double that amount if you include smartphones—not to mention the computers owned by the universities themselves! And each one of those 10,000 or more computers is at least as powerful as, if not more so than, the 100 that there were a half-century ago.

Although the early computers were mostly intended for military uses,

> The basic purpose [of computers at universities], at present [that is, in 1962], is to do computations associated with and supported by university research programs, largely government financed. . . . *Sad to state,* ***some*** *uses occur merely because the computer is available, and seem to have no higher purpose than that.*
> —Perlis (1962, p. 182, my italics)

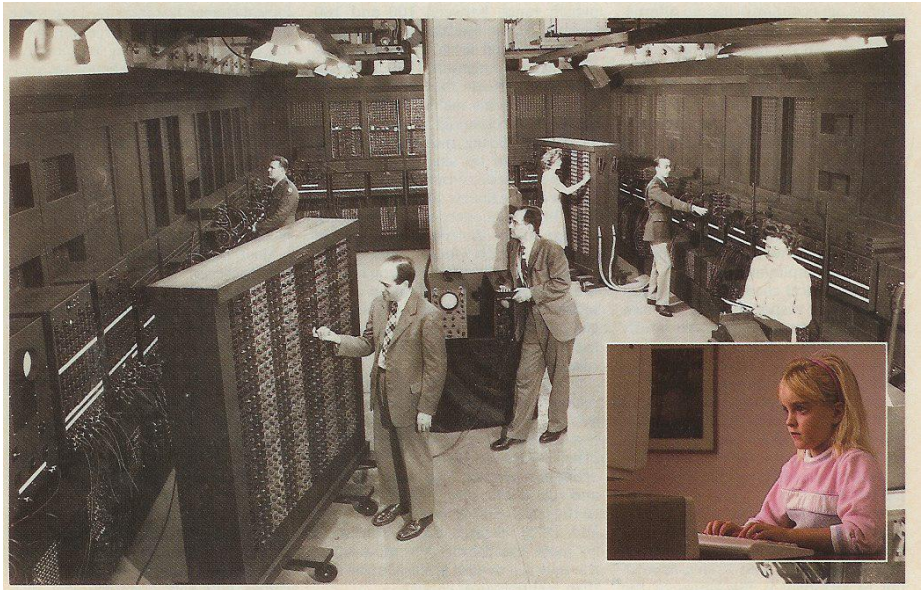
And I wouldn't be surprised if *most* uses (Candy Crush? Skype? Facebook? Twitter? Amazon?) of the 10,000 computers at an average contemporary university "have no higher purpose"! (At this point, you are urged to re-read the chronologically-ordered epigraphs at the beginning of this chapter.)

It is also worth noting the simultaneous *decrease in size* of computers from the 1940s to now, as well as their ease of use, as illustrated in Figures 6.3 and 6.4.

Further Reading:
For the history of personal computers, see (Ryan, 1991) (which tries to predict the future of what is now known as laptop computers, asking "Is the reign of the desktop computer about to end?'), (Press, 1993), (Markoff, 2000) (on the history of Microsoft Basic), (Waldrop, 2001), (Markoff, 2005), and (Lohr, 2010).

On precursors of the Internet and the Web, see (Standage, 1998), (Alden, 1999), and (Wright, 2008) (on a 1934(!) version of a World Wide Web). For a 1909(!) version of the Internet, see (Forster, 1909). For more recent histories of the Internet and the Web, see Brian Hayes 1994, 2000.

(As for "higher purposes", see (Hafner, 2002) :-)



TRAINING necessary to operate a computer has been drastically reduced as a result of advances in both hardware and software. Only highly trained operators could run the first computers, which were developed some 40 years ago. Today's personal computers (which surpass those first machines in both memory and computing power) can be operated by a child.

58 SCIENTIFIC AMERICAN *May 1989*

Figure 6.3: The ENIAC (circa 1946), with Eckert at the controls, and a smaller and more powerful personal computer (circa 1989), with a chid at the controls

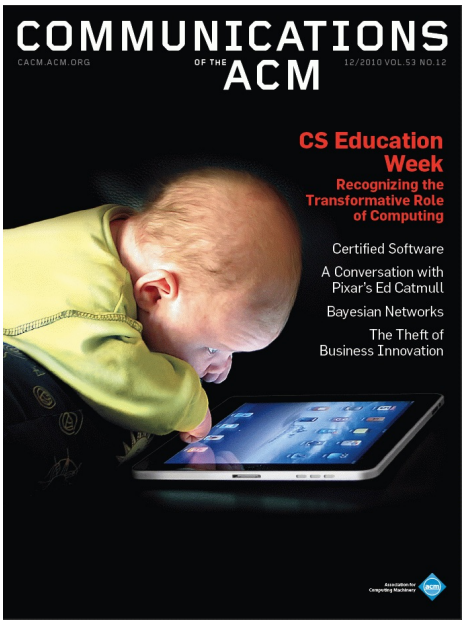


Figure 6.4: Cover of the *Communications of the ACM*, December 2012

6.6 The Scientific, Mathematical, Logical History

> Logic's dominant role in the invention of the modern computer is not widely appreciated. The computer as we know it today was invented by Turing in 1936, an event triggered by an important logical discovery announced by Kurt Gödel in 1930. Gödel's discovery ... decisively affected the outcome of the so-called Hilbert Program. Hilbert's goal was to formalize all of mathematics and then give positive answers to three questions about the resulting formal system: is it consistent? is it complete? is it decidable? Gödel found that no sufficiently rich formal system of mathematics can be both consistent and complete. In proving this, Gödel invented, and used, a high-level symbolic programming language: the formalism of primitive recursive functions. As part of his proof, he composed an elegant modular functional program .... This computational aspect of his work ... is enough to have established Gödel as the first serious programmer in the modern sense. Gödel's computational example inspired Turing ... [who] disposed of the third of Hilbert's questions by showing ... that the formal system of mathematics is not decidable. Although his original computer was only an abstract logical concept, ... Turing became a leader in the design, construction, and operation of the first real computers. (Robinson, 1994, pp. 6–7)[6]

[6]Roughly, a formal system is "consistent" if no false propositions can be proved within it, and it is "complete" if every true proposition *can* be proved within it. What Gödel proved was that, if arithmetic is consistent, then it is incomplete, because an arithmetical version of the English sentence "This sentence is unprovable" is true but unprovable. We will discuss "primitive recursive functions" in Ch. 7 ("What Is

The parallel historical story concerns, not the construction of a physical device that could compute, but the logical and mathematical analysis of what computation itself is.

This story begins, perhaps, with Leibniz, who not only constructed a computing machine, as we have seen, but who also wanted to develop a "calculus ratiocinator": a formalism in a universally understood language (a "characteristica universalis") that would enable its "speakers" to precisely express any possible question and then to rationally calculate its answer. Leibniz's motto (in Latin) was: Calculemus! (Let us calculate!). In other words, he wanted to develop an algebra of thought.

This task was taken up around 180 years later (around 160 years ago) by the English mathematician George Boole (1815–1864), who developed an algebra of logic, which he called *The Laws of Thought* (Boole, 2009). This was what is now called propositional logic. But it lacked a procedure for determining the truth value of a given (atomic) statement.

Boole's work was extended by the German mathematician Gottlob Frege (1848–1925, around 130 years ago), who developed what is now called first-order logic (or the first-order predicate calculus).[7] Frege was a follower of a philosophy of mathematics called "logicism", which viewed mathematics as a branch of logic. Thus, to give a firm foundation for mathematics, it would be necessary to provide a system of logic that itself would need no foundation.

Unfortunately, the English philosopher Bertrand Russell (1872–1970, around 100 years ago), discovered a problem while reading the manuscript of Frege's book *The Foundations of Arithmetic*. This problem, now known as Russell's Paradox, concerned the logic of sets: A set that has as members all and only those sets that do *not* have themselves as members would *both* have itself as a member *and not* have itself as a member. This inconsistency in Frege's foundation for mathematics began a crisis that resulted in the creation of the theory of computation.

Further Reading:
An enjoyable graphic-novel treatment of the Russell-Frege story, written by a well-known computer scientist, is (Doxiadis et al., 2009).

That story continues with work done by the German mathematician David Hilbert (1862–1943, around 115 years ago), who wanted to set mathematics on a rigorous, logical foundation, one that would be satisfactory to all philosophers of mathematics, including "intuitionists" and "finitists". (Intuitionists believe that mathematics is a construction of the human mind, and that any mathematical claim that can only be proved by showing that its assumption leads to a contradiction should not be accepted. Finitists believe that only mathematical objects constructible in a finite number of steps should be allowed into mathematics.) It is worth quoting Hilbert at length:

an Algorithm?"), §7.6.2. For more on Gödel, see Appendix A ("Argument Analysis and Evaluation") and Nagel et al. 2001; Hofstadter 1979; and Franzén 2005. For a discussion of the relationship between Gödel's theorems and Turing machines, see Feferman 2011.

[7] None of these things called 'calculus' (plural: 'calculi') are related to the differential or integral calculus. 'Calculus' just means "system for calculation".

> Occasionally it happens that we seek ... [a] solution [to a mathematical problem] under insufficient hypotheses or in an incorrect sense, and for this reason do not succeed. **The problem then arises: to show the impossibility of the solution under the given hypotheses, or in the sense contemplated.** Such proofs of impossibility were effected by the ancients, for instance when they showed that the ratio of the hypotenuse to the side of an isosceles right triangle is irrational. In later mathematics, the question as to the impossibility of certain solutions plays a preeminent part, and we perceive in this way that old and difficult problems, such as the proof of the axiom of parallels, the squaring of the circle, or the solution of equations of the fifth degree by radicals have finally found fully satisfactory and rigorous solutions, although in another sense than that originally intended. It is probably this important fact along with other philosophical reasons that gives rise to **the conviction (which every mathematician shares, but which no one has as yet supported by a proof) that every definite mathematical problem must necessarily be susceptible of an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts.** Take any definite unsolved problem, ... However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, **the firm conviction that their solution must follow by a finite number of purely logical processes.** Is this **axiom of the solvability of every problem** a peculiarity characteristic of mathematical thought alone, or is it possibly a general law inherent in the nature of the mind, **that all questions which it asks must be answerable**? This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no *ignorabimus* ["We will not know"]. (Hilbert, 1900, pp. 444–445, my boldface)

Further Reading:
For more on impossibility proofs in mathematics, see (Stewart, 2000). On impossibility proofs more generally, see also (Toussaint, 1993). The most famous impossibility proof in CS, of course, is the Halting Problem; see §7.7.

Hilbert proposed the following "Decision Problem" (*Entscheidungsproblem*) for mathematics: to devise a procedure according to which it can be decided by a finite number of operations whether a given statement of first-order logic is a theorem. (We will return to the decision problem, see §8.4.)

Digression: What Is a Theorem?

When you studied geometry, you probably studied Euclid's original presentation of geometry via a modern interpretation as an axiomatic system. Most branches of mathematics (and, according to some philosophers, most branches of science) can be formulated axiomatically. One begins with a set of "axioms"; these are statements that are assumed to be true (or are considered to be so obviously true that their truth can be taken for granted). Then there are "rules of inference" that tell you how to logically infer other statements from the axioms in such a way that the inference procedure is "truth preserving"; that is, if the axioms are true (which they are, by assumption), then whatever logically follows from them according to the rules of inference are also true. Such statements are called 'theorems'. (See Appendix A for more details.)

The Decision Problem:

There are varying versions of the decision problem. Here are some:

> ... determining whether or not a given formula of the predicate calculus is universally valid. —(Hilbert and Ackermann, 1928, p. 112)

> In the broadest sense, the decision problem can be considered solved if we have a method which permits us to decide for any given formula *in which domains of individuals it is universally valid (or satisfiable) and in which it is not.*
> —(Hilbert and Ackermann, 1928, pp. 114–115)

> The Entscheidungsproblem is solved if one knows a procedure that allows one to decide the validity (respectively, satisfiability) of a given logical expression *by a finite number of operations*.
> —(Translation in (Sieg, 1994, p. 77, my italics), possibly of the above passage from (Hilbert and Ackermann, 1928, pp. 114–115).)

> By the Entscheidungsproblem of a system of symbolic logic is here understood the problem to find an effective method by which, given any expression Q in the notation of the system, it can be determined whether or not Q is provable in the system. (Church, 1936a, p. 41, note 6)

An earlier version (dating from 1900) appeared in Hilbert's list of 23 math problems that he thought should be investigated during the 20th century. The 10th problem was this:

> Given a diophantine [sic; usually, this word is capitalized] equation with any number of unknown quantities and with rational integral numerical coefficients: *to devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.*
> —(English translation from http://aleph0.clarku.edu/~djoyce/hilbert/problems.html\#prob10)

Like the Halting Problem, Hilbert's 10th Problem turns out to be non-computable; that is, there is no such process, no such algorithm. (See §7.7.3.1.)

For more on decision problems and the *Entscheidungsproblem*, see Bernhardt 2016, pp. 8–10 and Chs. 2 and 7.

A mathematical statement that was decidable in this way was also said to be "effectively computable" or "effectively calculable", because one could compute, or calculate, whether or not it was a theorem in a finite number of steps.[8]

Many mathematicians took up Hilbert's challenge: In the US, Alonzo Church (1903–1995) analyzed the notion of "function" and developed the lambda-calculus (see below), claiming that any function whoses values could be computed in the lambda-calculus was effectively computable. The Austrian (and later American) logician Kurt Gödel (1906–1978), who had previously proved the incompleteness of arithmetic (and thus became the most respected logician since Aristotle; see footnote 6, above), developed the notion of "recursive" functions, claiming that this was co-extensive with effectively computable functions. Emil Post, a Polish-born American logician (1897–1954), developed "production systems", which also capture the notion of effective computability (Soare, 2009, §5.2, p. 380). And the Russian A.A. Markov (1856–1922) developed what are now known as Markov algorithms. (We will look in more detail at some of these systems in Chapter 7.)

But central to our story was the work of the English mathematician Alan Turing (1912–1954), who—rather than trying to develop a mathematical theory of effectively computable functions in the way that the others approached the subject—gave an analysis of *what human computers did.* Based on that analysis, he developed a formal, mathematical model of a human computer, which he called an "*a*-machine", and which we now call, in his honor, a Turing machine. In his classic paper published in 1936 (Turing, 1936), Turing presented his informal analysis of human computation, his formal definition of an *a*-machine, his claim that functions computable by *a*-machines were all and only the functions that were "effectively computable", a (negative) solution to Hilbert's Decision Problem (by showing that there was a mathematical problem that was *not* decidable computationally, namely, the Halting Problem), a demonstration that a single Turing machine (a "universal Turing machine") could do the work of all other Turing machines, *and*—as if all that were not enough—a proof that a function was computable by an *a*-machine if and only if it was computable in Church's lambda-calculus. (To fully appreciate his accomplishment, be sure to calculate how old he was in 1936!) We will look at Turing's work in much greater detail in Chapter 8.)

Later, others proved that both methods were also logically equivalent to all of the others: recursive functions, production systems, Markov algorithms, etc. Because all of these theories had been proved to be logically equivalent, this finally convinced almost everyone that the notion of "effective computability" (or "algorithm") had been captured precisely. Indeed, Gödel himself was not convinced until he read Turing's paper, because Turing's was the most intuitive presentation of them all. (But, in Chapters 10 and 11, we will look at the arguments of those who are still not convinced.)

[8] For more on this notion of "effectiveness", see Ch. 7, §7.4.

Further Reading:
An excellent, brief overview of the history of logic and the foundations of mathematics that led up to Turing's analysis can be found in Henkin 1962, pp. 788–791. See also Stewart Shapiro 1983; Sieg 1994, §1; Chaitin 2002; Soare 1999; and, especially, Soare 2016, Ch. 17.

For the logical history as written by one of its chief players, see (Kleene, 1981). Martin Davis, another pioneer in the theory of computation, has written a lot on its history: (Davis, 2000) is a somewhat negative review of David Berlinksi, *The Advent of the Algorithm* (Harcourt, 2000), correcting some of the historical errors in that book. (Davis, 2003) is a review of Marcus Giaquinto, *The Search for Certainty: A Philosophical Account of Foundations of Mathematics* (Oxford University Press, 2002). Early sections of (Davis, 2004) contain a good summary of the history of computation.

On Church, see Manzano 1997. For a very elementary introductions to the lambda-calculus, see "PolR" 2009, §"Alonzo Church's Lambda-Calculus" and Alama and Korbmacher 2018.

The role of philosophy in the history of computers is told in George 1983.

For a somewhat controversial take on the history of computing (and the notion of a stored-program computer), see a debate between computer scientist Moshe Vardi and philosopher B. Jack Copeland: Vardi 2013, Copeland 2013, Vardi 2017.

6.7 The Histories Converge

At this point, the engineering and mathematical histories converge:

> ... it is really only in von Neumann's collaboration with the ENIAC team that two quite separate historical strands came together: the effort to achieve high-speed, high-precision, automatic calculation and the effort to design a logic machine capable of significant reasoning.
>
> The dual nature of the computer is reflected in its dual origins: hardware in the sequence of devices that stretches from the Pascaline to the ENIAC, software in the series of investigations that reaches from Leibniz's combinatorics to Turing's abstract machines. Until the two strands come together in the computer, they belong to different histories .... (Mahoney, 2011, p. 26)[9]

The development of programmable, electronic, digital computers—especially the EDVAC, with its von Neumann architecture—began to look like Turing machines, and Turing himself decided to implement a physical computer based on his architecture. (For more on this, see (Carpenter and Doran, 1977).)

[9](Mahoney, 2011, p. 88) also emphasizes the fact that these histories "converged" but were not "coincident".

6.8 What Is a Computer?

The twin histories suggest different answers to our question.

6.8.1 What Is a Computer, Given the Engineering History?

If computers can be defined "historically", then they are "machines which (i) perform calculations with numbers, (ii) manipulate or process data (information), and (iii) control continuous processes or discrete devices ... in real time or pseudo real time" (Davis, 1977, p. 1096). Note that (ii) can be considered a generalization of (i), because numbers are a kind of data and because performing calculations with numbers is a kind of manipulation of data. And, because being continuous or being discrete pretty much exhausts all possibilities, criterion (iii) doesn't really seem to add much. So this characterization comes down to (ii) alone:

> A computer is a machine that manipulates or processes data (information).

Or does it? One possible interpretation of clause (iii) is that the *output* of a computer need not be limited to *data*, but might include instructions to other "processes ... or devices", that is, real-world effects. (We'll look into this in Chapter 17.)

According to (Davis, 1977, pp. 1096–1097), computers had evolved to have the following "key characteristics" (perhaps among others):

1. "digital operation"
   (thus focusing on only the discrete aspect of (iii), above);

2. "stored program capability"
   (here understood as "the notion that the instructions for the computer be written in the same form as the data being used by the computer" and attributed to von Neumann; we will return to this issue in Chapter 8);

3. "self-regulatory or self-controlling capability"
   (this is not merely the automaticity of any machine, but it seems to include the ideas of feedback and "automatic modifiable stored programs");

4. "automatic operation"
   (singled out from the previous characteristic because of its emphasis on operating "independently of human operators and human intervention")

5. "reliance on electronics"
   (this is admitted to be somewhat parochial in the sense that electronic computers were, at the time of writing, the dominant way of implementing them, but Davis recognized that other kinds of computers would eventually exist; recall our mention of quantum, DNA, and other computers in §3.5.4).

So, ignoring the last item and merging the previous two, we come down to a version of our previous characterization:

> A (modern) computer is an automatic, digital, stored-program machine (for manipulating information).

What is the nature of the "information" that is manipulated? Davis said that it is *numbers*. But numbers are abstract entities not susceptible to (or capable of) physical manipulation. Computers really manipulate *numerals*—that is, physical symbols that *represent* numbers—not the (abstract) numbers themselves. So, are computers machines that manipulate physical (concrete) *symbols*, or machines that (somehow) manipulate non-physical (abstract) *numbers*? There are two versions of this question. The first version contrasts *numbers* with numerical *symbols* (that is, *numerals*). The second version contrasts numbers *and* numerical symbols in particular with symbols more generally.

The first question is closely related to issues in the philosophy of mathematics. Is math itself more concerned with numerals than with numbers, or the other way around? "Formalists" and "nominalists" suggest that it is only the symbols for numbers that we really deal with. "Platonists" suggest that it is numbers that are our real concern, but at least some of them admit that the only way that we can directly manipulate numbers is via numerals (although some Platonists, including Gödel, suggest that we have a kind of perceptual ability, called 'intution', that allows us to access numbers directly). There are also related questions about whether numbers exist and, if so, what they are. But these issues are beyond our scope. (For more on the philosophy of mathematics, see the suggested readings in §2.8.)

Computers, pretty clearly, have to deal with numbers via numerals. So, "The mathematicians and engineers then [in the 1950s] responsible for computers [who] insisted that computers only processed *numbers*—that the great thing was that instructions could be translated into numbers" (Newell, 1980, p. 137) were probably wrong. But even if we modify such a claim so that we replace numbers by numerals, we are faced with the second question above. Do computers *only* manipulate numerals (or numbers)? What about all of the things that you use your personal computers for (not to mention your smartphones)—how many of them involve numerals (or numbers)?

An answer to that question will depend in part on how we interpret the symbols that a computer deals with. Certainly, there are ways to build computers that, apparently, can deal with more than merely numerical symbols. The Lisp machines of the 1980s are prime examples: Their fundamental symbols were Lisp lists.[10] But, insofar as any computer is ultimately constructed from physical switches that are either in an "on/up" or "off/down" position, we are left with a symbol system that is binary—hence numerical—in nature. Whether we consider these symbols to be numerals or not may be more a matter of taste or convenience than anything more metaphysical.

[10]Lisp is a programming language whose principal data structure is a "linked list". See, for example, (Shapiro, 1992b).

6.8.2 What Is a Computer, Given the Scientific History?

If the engineering history suggests that a computer is an automatic, digital, stored-program machine (for manipulating information), what does the scientific-mathematical-logical history suggest? Is a computer merely a physical implementation of a Turing machine? But Turing machines are hopelessly inefficient and cumbersome ("register" machines, another Turing-equivalent model of computation, are closer to modern computers; see §9.4.1). As Alan Perlis has observed,

> What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak. —Alan J. Perlis, "Epigrams in Programming", http://www.cs.yale.edu/homes/perlis-alan/quotes.html.

To clarify some of this, it will be necessary for us to look more closely at the nature of "effective computation" and "algorithms", which we will do in the next chapter. Armed with the results of that investigation, we will return to the question of what a computer is (from a philosophical point of view), in Chapter 9.

Further Reading:
One answer to the question "What is a computer?", aimed at radio engineers who—in the early 1950s—might not be familiar with them, is (Samuel, 1953), written by an IBM researcher who later became famous for his work on computer checkers-players;

(Copeland, 2004a, pp. 3–4) discusses "The Birth of the Modern Computer". (Haigh, 2014) discusses the (ir)relevance of the mathematical history of computation to the engineering history. Despite its title ("Histories of Computing"), (Mahoney, 2011) is not so much a history of computing or computers as a history of CS. (Mahoney, 2011, Chs. 10 and 11) are especially good on some of the recent mathematical history.

6.9 Further Sources of Information

1. Lohr, Steve (2017), "Jean Sammet, Co-Designer of a Pioneering Computer Language, Dies at 89", *New York Times* (4 June), https://www.nytimes.com/2017/06/04/technology/obituary-jean-sammet-software-designer-cobol.html
2. Markoff, John (1992), "Rear Adm. Grace M. Hopper Dies; Innovator in Computers Was 85", *New York Times* (3 January), http://www.nytimes.com/1992/01/03/us/rear-adm-grace-m-hopper-dies-innovator-in-computers-was-85.html
3. Mathews, Walter M.; & Reifers, Kaye (1984), "The Computer in Cartoons: A Retrospective from *The Saturday Review*", *Communications of the ACM* 27(11) (November): 1114–1119.
4. Sammet, Jean E. (1992), "Farewell to Grace Hopper—End of an Era!", *Communications of the ACM* 35(4) (April): 128–131, http://dx.doi.org/10.1145/129852.214846

Chapter 7

What Is an Algorithm?

Version of 7 August 2019; DRAFT © 2004–2019 by William J. Rapaport

> Thou must learne the Alphabet, to wit, the order of the Letters as they stand. ... Nowe if the word, which thou art desirous to finde, begin with (a) then looke in the beginning of this Table, but if with (v) looke towards the end. Againe, if thy word beginne with (ca) looke in the beginning of the letter (c) but if with (cu) then looke toward the end of that letter. And so of all the rest. &c.
> —Robert Cawdrey, *A Table Alphabeticall, conteyning and teaching the true writing, and understanding of hard usuall English wordes* (1604), cited in Gleick 2008, p. 78.

> This nation is built on the notion that the rules restrain our behavior ....
> —*New York Times* 2006

> Algorithmic behavior existed long before there was an algorithm. —Janice Min, quoted in Rutenberg 2019

Figure 7.1: http://babyblues.com/comics/february-25-2004/, ©2004 Baby Blues Partnership

Readings:

1. Required:
   - Henkin, Leon (1962), "Are Logic and Mathematics Identical?", *Science* 138(3542) (November 16): 788–794.
     - Read pp. 788–791; skim the rest
     - An excellent, brief overview of the history of logic and the foundations of mathematics that led up to Turing's analysis.
   - Davis, Martin (1987), "Mathematical Logic and the Origin of Modern Computers", *Studies in the History of Mathematics*; reprinted in Rolf Herken (ed.), *Universal Turing Machine: A Half-Century Survey; Second Edition* (Vienna: Springer-Verlag, 1995): 135–158.
     - Overlaps and extends Henkin's history, and provides a useful summary of (Turing, 1936), which we will discuss in great detail in Chapter 8.
   - Herman, Gabor T. (1993), "Algorithms, Theory of", in Anthony S. Ralston & Edwin D. Riley (eds.), *Encyclopedia of Computer Science, 3rd edition* (New York: Van Nostrand Reinhold): 37–39.
     - Discussion of the informal notions of "algorithm" and "effective computability"; good background for (Turing, 1936).
2. Very Strongly Recommended:
   - *Either*:

     Soare, Robert I. (1996), "Computability and Recursion", *Bulletin of Symbolic Logic* 2(3) (September): 284–321.

     or:

     Soare, Robert I. (1999), "The History and Concept of Computability", in E.R. Griffor (ed.), *Handbook of Computability Theory* (Amsterdam: Elsevier): 3–36.
     - "a revised and shortened form" of Soare 1996.
     - Read §§1–3, 4.5–4.6; skim the rest
3. Strongly Recommended:
   - Browse through the "Examples of Algorithms" at:
     http://www.cse.buffalo.edu/~rapaport/584/whatisanalg.html
   - Haugeland, John (1981), "Semantic Engines: An Introduction to Mind Design", in John Haugeland (ed.), *Mind Design: Philosophy, Psychology, Artificial Intelligence* (Cambridge, MA: MIT Press): 1–34.
     - A good description of the syntax and semantics of formal systems and their relationship to Turing Machines.
   - Böhm, C.; & Jacopini, G. (1966), "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules", *Communications of the ACM* 9(5): 366–371.
     - Uses flowcharts to prove that "go to" statements are eliminable from computer programs in favor of sequence, selection, and repetition (loops). An important paper, but not for the faint of heart!

7.1 Introduction

> [C]computer science is not really that much about computers. What computer science is mostly about is *computation*, a certain kind of process such as sorting a list of numbers, compressing an audio file, or removing red-eye from a digital picture. The process is typicaly carried out by an electronic computer of course, but itmight also be carried out by a person or by a mechanical device of some sort.
>
> The hypothesis underlying AI ... is that ordinary *thinking* ... is also a computational process, and one that can be studied without too much regard for who or what is doing the thinking. —Levesque (2017, pp. ix–x)

We have been examining two questions: (1) whether CS is a science (or something else, such as a branch of engineering or some combination of both science and engineering) and (2) what its subject matter is.

Does CS study comput*ers*: (physical) devices that compute—or does it study comput*ing*: the algorithmic processes that computers do? (Or perhaps it studies something else, such as information, or information processing.) In the former case, we need to ask what *computers* are; in the previous chapter, we began that investigation by looking at the history of computers. In this chapter, we ask what *computing* is. Then we will be in a better position to return to our question of what a computer is, looking at it from a philosophical, rather than a historical, point of view. And after *that*, we will return to the question of what computing is, again looking at some philosophical issues.

7.2 What Is 'Computation'?

> Many now view computation as a fundamental part of nature, like atoms or the integers. —(Fortnow, 2010, p. 2)

Although I have been talking about "computing", other terms that are used to describe more or less the same territory are 'computation' and 'algorithms'. It may be worth a brief detour into the etymology of these and some related terms. (We'll look at the etymology of 'algorithm' in §7.4.1.)

7.2.1 'compute'

According to the *Oxford English Dictionary* (*OED*)[1], the verb 'to compute' comes from the Latin verb '*computare*', meaning "to calculate, reckon, to count up". But when people talk about "computing" today, they mean a great deal more than mere counting. Computing has come to include everything we can do with computers, including text processing, watching videos, and playing games. So, clearly, the meaning has been extended to include non-numerical "reckoning".

The Latin word '*computare*', in turn, comes from the Latin morpheme '*com*', meaning "together with", and the Latin word '*putare*', meaning "to cleanse, to prune, to reckon, to consider, think" (and '*putare*' came from a word meaning "clean, pure").

[1] http://www.oed.com/view/Entry/37974

So, in ancient Rome at least, to "compute" seems to have meant, more or less, something like: "to consider or think about things together", "to clear things up together", or maybe "to reckon with (something)".

7.2.2 'reckon'

The verb 'to reckon' originally meant "to give an account of, recount; to tell; to describe", and later came to mean "to count, to calculate". 'Reckon' is from an Indo-European[2] root '*rek*', possibly meaning "to reach" or "to tell, to narrate, to say" (as in "to recite" or "to recount"). These meanings, in turn, may derive from an earlier meaning "to arrange", "to put right", "to move in a straight line".[3]

7.2.3 'count', 'calculate', 'figure'

The origins of 'count', 'calculate', and 'figure' are also interesting.

'Count' also came from '*computare*' and originally meant "to enumerate", "to recite a list" (and, as we just saw, 'recite' is probably related to 'reckon'). Note that when you "count", you "recite" a list of number words.

'Calculate' came from Latin '*calculus*'. This certainly did not mean the contents of a certain high school or college course that studies the branch of mathematics concerned with differentiation and integration, and invented by Newton and Leibniz in the 17th century! Rather, it meant "pebble" or "small stone", since counting was done with stones originally. (See Figure 7.2.) Even today, a "calculus" in medicine is an accumulation of minerals in the body, forming a small, stone-like object. The root '*calc*' came from 'calx', meaning "chalk, limestone", and is related to 'calcium'.

Figure 7.2: http://rhymeswithorange.com/comics/august-22-2011/, ©2011, Hilary B. Price

The *verb* 'to figure' means "to use figures to reckon". The earliest citation in the *OED* for the *noun* 'figure' is from 1225, when it meant "numerical symbol". A citation from 1250 has the meaning "embodied (human) form". And a citation from 1300 has the more general meaning of "shape". (This conversion of the noun 'figure' to a verb

[2] http://en.wikipedia.org/wiki/Indo-European_languages

[3] See http://etymonline.com/?term=reckon
and http://www.utexas.edu/cola/centers/lrc/ielex/R/P1613.html

is an example of what the computer scientist Alan Perlis meant when he joked, "In English, every word can be verbed".)[4]

7.2.4 'computation'

The bottom line seems to be this: 'Computation' originally meant something very closely related to our modern notion of "symbol (that is, shape) manipulation", which is another way of describing syntax—the "grammatical" properties of, and relations among, the symbols of a language. (We'll talk more about syntax in Chapter 22, a preliminary version of which appears as (Rapaport, 2017b).)

Now that we know how we got the *word* 'computation', we'll turn to what the *object* computation is.

Further Reading:
For links to some of these etymologies, see:
http://www.cse.buffalo.edu/~rapaport/584/computetymology.html. On the history of the terms 'computable' vs. 'recursive', see Soare (2009, §11.5, p. 391); we'll discuss recursive functions in §7.6.2.

[4] http://www.cs.yale.edu/homes/perlis-alan/quotes.html

7.3 What Is Computation?

> The question before us—what is computation?—is at least as old as computer science. It is one of those questions that will never be fully settled because new discoveries and maturing understandings constantly lead to new insights and questions about existing models. It is like the fundamental questions in other fields—for example, "what is life?" in biology and "what are the fundamental forces?" in physics—that will never be fully resolved. Engaging with the question is more valuable than finding a definitive answer. —(Denning, 2010)

To understand what computation is, we first need to understand what a (mathematical) *function* is.

7.3.1 What Is a Function?

7.3.1.1 Two Meanings

The English word 'function' has at least two, very different meanings. (1) The *ordinary, everyday meaning* is, roughly, "purpose". Instead of asking, "What is the *purpose* of this button?", we might say, "What is the *function* of this button?" To ask for the function—that is, the purpose—of something is to ask "What does it do?". (2) In this chapter, we will be interested in its *mathematical meaning*, as when we say that some "dependent variable" *is a function of*—that is, depends on—some "independent variable". (We'll return to its other meaning in Chapter 17; see also (Rapaport, 2017a).)

Further Reading:
As it turns out, this technical sense of the word was, first of all, initiated by Leibniz and, second, was an extension of its other meaning; for the history of this, see the *OED* entry on the noun 'function', in its mathematical sense (sense 6; http://www.oed.com/view/Entry/75476). On the history of the concept of "function", see (O'Connor and Robertson, 2005).

7.3.1.2 Functions Described Extensionally

Many introductory textbooks define a (mathematical) function as an "assignment" or "mapping" of *values* (sometimes called "dependent variables") to *inputs* (sometimes called "independent variables"). But they never define what an "assignment" is. Such an "assignment" is not quite the same thing as an assignment of a value to a variable in a programming language or in a system of logic. A better term might be 'association': A value (or dependent variable) *is associated with* an input (or independent variable). A much more rigorous way of defining a function is to give a definition based on set theory, thus explicating the notion of "association". There are two ways to do this: "extensionally" and "intensionally" (recall our discussion of extensionality and intensionality in §3.4).

A **function** described "extensionally" is a set of input-output pairs such that no two of them have the same input (or first element). A "binary relation" is a set of ordered pairs of elements from two sets; so, a function is a certain kind of binary relation. (The

"two" sets can be the same one; you can have a binary relation among the members of a single set.) But a function is a special kind of binary relation in which no *two*, distinct members of the relation have the same first element (but different second elements). That is, the input (or independent variable) of a function must always have the same output (or dependent variable). Here is another way of saying this: Suppose that you have what you *think* are two different members of a function; and suppose that they have the same first element and also have the same second element. Then it only *seemed* as if they were two members—they were really one and the same member, not two different ones. As a rule of thumb, a binary relation is a function if "same input implies same output".

The logically correct way to say this, in mathematical English, is as follows:

> Let A, B be sets. (Possibly, $A = B$.)
> Then f **is a *function* from** A **to** $B =_{def}$ (that is, means by definition)
>
> 1. f is a binary relation from A to B,
>
>   and
>
> 2. **for all** members $a \in A$, and
>   **for all** members $b \in B$, and
>   **for all** members $b' \in B$,[5]
>   **if** (a, b) is a member of f, and
>   **if** (a, b') is *also* a member of f,
>   **then** $b = b'$

Because we are considering a binary relation as a set of ordered pairs, let's write each member of a binary relation from A to B as an ordered pair $\langle a, b \rangle$, where $a \in A$ and $b \in B$. Here are some examples of functions in this extensional sense:

1. $f = \{\langle 0,0\rangle, \langle 1,2\rangle, \langle 2,4\rangle, \langle 3,6\rangle, \ldots\}$
   Using "functional" notation—where f(input) = output—this is sometimes written: $f(0) = 0,\ f(1) = 2,\ \ldots$

2. $g = \{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \ldots\}$
   This is sometimes written: $g(0) = 1,\ g(1) = 2,\ \ldots$

[5] Keep in mind that b' might be the same as b! The best way to think about these sequences of "for all" (or "universal quantifier") statements is this: Imagine that sets are paper bags containing their members. (1) "For all $a \in A$" means: Put your hand in bag A, remove a (randomly chosen) member, look at it to see what it is, *and return it to the bag*. (2) "For all $b \in B$" means: Put your hand in bag B, remove a (randomly chosen) member, look at it to see what it is, *and return it to the bag*. Finally, (3) "For all $b' \in B$" means exactly the same thing as in case (2), which means, in turn, that b' in step (3) might be the same member of B that you removed *but then replaced* in step (2); you might simply have picked it out twice by chance.

3. $E = \{\langle y, \langle m, d\rangle\rangle : \langle m, d\rangle = \langle$
$((((19*(y \mod 19))+(y/100)-((y/100)/4)-(((8*(y/100))+13)/25)+15) \mod 30)-(((y \mod 19)+(11*(((19*(y \mod 19))+(y/100)-((y/100)/4)-(((8*(y/100))+13)/25)+15) \mod 30)))/319)+(((2*((y/100) \mod 4))+(2*((y \mod 100)/4))-((y \mod 100) \mod 4)-(((19*(y \mod 19))+(y/100)-((y/100)/4)-(((8*(y/100))+13)/25)+15) \mod 30)-(((y \mod 19)+(11*(((19*(y \mod 19))+(y/100)-((y/100)/4)-(((8*(y/100))+13)/25)+15) \mod 30)))/319)+32) \mod 7)+90)/25$,

$((((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)-(((y \mod 19)+(11*(((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)))/319)+(((2*((y/100) \mod 4))+(2*((y \mod 100)/4))-((y \mod 100) \mod 4)-(((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)-(((y \mod 19)+(11*(((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)))/319)+32) \mod 7)+(((((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)-(((y \mod 19)+(11*(((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)))/319)+(((2*((y/100) \mod 4))+(2*((y \mod 100)/4))-((y \mod 100) \mod 4)-(((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)-(((y \mod 19)+(11*(((19*(y \mod 19))+(y/100)-(y/100)/4-(((8*(y/100))+13)/25)+15) \mod 30)))/319)+32) \mod 7)+90)/25+19) \mod 32$
$\rangle\}$

This function takes as input a year y and outputs an ordered pair consisting of the month m and day d that Easter falls on in year y (Stewart, 2001).

4. Here is a *finite* function (that is, a function with a finite number of members—remember: a function is a *set*, so it has members):

$$\begin{aligned} h = \{&\langle \text{`yes', print `hello'}\rangle, \\ &\langle \text{`no', print `bye'}\rangle, \\ &\langle \text{input} \neq \text{`yes' \& input} \neq \text{`no', print `sorry'}\rangle\} \end{aligned}$$

The idea behind h is this:

> h prints 'hello', if the input is 'yes';
> h prints 'bye', if the input is 'no';
> and h prints 'sorry', if the input is neither 'yes' nor 'no'.

5. Here is a *partial* function (that is, a function that has no outputs for some possible inputs):

$$k = \{\ldots, \langle -2, \frac{1}{-2}\rangle, \langle -1, \frac{1}{-1}\rangle, \langle 1, \frac{1}{1}\rangle, \langle 2, \frac{1}{2}\rangle, \ldots\}$$

Here, $k(0)$ is *undefined*.

6. Another example of a partial function is:

$$h' = \{\langle \text{`yes'}, \text{print `hello'}\rangle, \\ \langle \text{`no'}, \text{print `bye'}\rangle\}$$

Here, h'(‘yeah’), h'(‘nope’), and h'(‘computer’) are all *undefined*.

A function defined extensionally *associates* or *relates* its inputs to its outputs, but does not show how to *transform* an input into an output. For that, we need a “formula” or an “algorithm” (but these are not the same thing, as we will soon see).

7.3.1.3 Interlude: Functions Described as Machines

Sometimes, functions are characterized as “machines” that accept input into a “black box” with a “crank” that mysteriously transforms the input into an output, as in Figure 7.3.

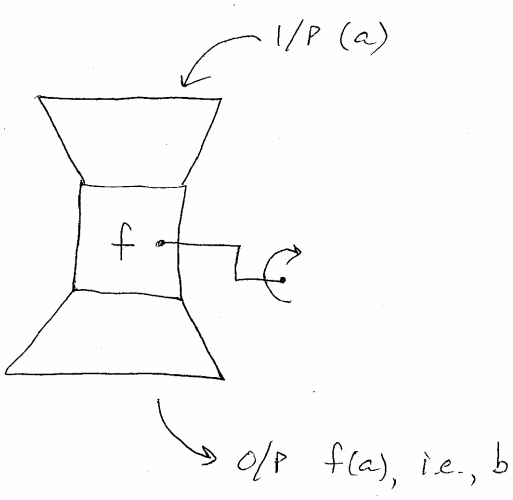


Figure 7.3: A function “machine” f that transforms input a into output $b = f(a)$.

In Figure 7.3, f is a machine into which you put a; you then turn a crank (clockwise, let’s suppose); f then grinds away at the input by means of some mysterious mechanism; and finally the machine outputs b (that is, $f(a)$). But this view of a function as being something “active” or “dynamic” that *changes* something is incorrect.[6]

Despite what you may have been told elsewhere (I was told this in high school), this “machine” is ***NOT*** what a function is! A function, as we saw in §7.3.1.2, is merely the *set* of input-output pairs. So, what is the machine? **It is a *computer*!** And the

[6] See http://www.askphilosophers.org/question/1877.

mysterious "gears" hidden inside the black box implement an *algorithm* that computes the function.

Interestingly, Gödel made this observation in the 1930s in an unpublished paper!

> [Turing] has shown that the computable functions defined in this way [that is, in terms of Turing machines] are exactly those for which you can construct a machine with a finite number of parts which will do the following thing. If you write down any number $n_1, \ldots, n_r$ on a slip of paper and put the slip into the machine and turn the crank, then after a finite number of turns the machine will stop and the value of the function for the argument $n_1, \ldots, n_r$ will be printed on the paper.
> —(Gödel, 1938, p. 168)

So, the machine pictured in Figure 7.3 is a Turing machine! In fact, one problem with this machine metaphor for a function, as we will see, is that not all functions can be computed by algorithms; that is, there are functions for which there are no such "function machines".

7.3.1.4 Functions Described Intensionally

> Editor: We are making this communication intentionally short to leave as much room as possible for the answers. 1. Please define "Algorithm." 2. Please define "Formula." 3. Please state the difference. T. WANGSNESS, J. FRANKLIN *TRW Systems, Redondo Beach, California* —(Wangsness and Franklin, 1966).[7]

Sometimes, functions are described "intensionally" by **formulas**. But—unlike an extensional description—this is not a unique way to describe them, because two different formulas can describe the same function. Here are some examples (using the same function names from §7.3.1.2):

1. $f(i) = 2i$
2. $g(i) = i + 1$
3. $g'(i) = 2i - i + 7/(3+4)$

   Note that g and g' use two different forumulas to describe the same function; that is, $g = g'$, even though their formulas are different.

> **Exercise for the Reader:**
> How would you state the fact that the two *formulas* are different?
> Note that you cannot do this by saying "$i + 1 \neq 2i - i + 7/(3+4)$".

4. $h(i) = \begin{cases} \text{'hello'}, & \textbf{if } i = \text{'yes'} \\ \text{'bye'}, & \textbf{if } i = \text{'no'} \\ \text{'sorry'}, & \textbf{otherwise} \end{cases}$
5. **if** $i \neq 0$, **then** $k(i) = \frac{1}{i}$.

[7] For the published answers, see the Further Reading box at the end of this section.

A function described *extensionally* is like a black box; we know the inputs and outputs, but not how they are related. (To continue the machine metaphor, we don't know what (if anything) goes on inside the machine.) A function described *intensionally* via a *formula* is less opaque and gives us more understanding of the relationship between the input and the outputs.

Further Reading:
For more on this notion of understanding in terms of the internal workings of a black box, see (Strevens, 2013), which also suggests an analogy between computation and causation, a topic that we will return to in Chapters 10, 12, 14, 16, and 21.

A function described intensionally via an *algorithm* gives us even more understanding, telling us not only what the relationship is, but giving explicit instructions on how to make the conversion from input to output.

Although formulas may look a lot like algorithms, they are not the same thing. Consider, for example, the formula '$2+4*5$': Without an explicit statement of a rule telling you whether to multiply first or to add first, there is no way of knowing whether the number expressed by that formula is 30 or 22. Such a rule, however, would be part of an *algorithm* telling you how to calculate the value of the formula.

Or consider the formula '$2x+1$': Should you first calculate $2x$ and then add 1 to it? Or should you store 1 somewhere (say, by writing it on a piece of paper), then calculate $2x$, and finally add $2x$ to 1? And how should you calculate $2x$? Take 2, and then multiply it by x? Or take x, and then multiply it by 2? One of these might be easier to do than the other; for instance, 2×1000 might take only 1 step, whereas 1000×2 might take 999 steps. Of course, the commutative laws of addition and multiplication tell us that, in this case, as far as the output is concerned, it doesn't matter in which order you compute the value of the formula; however, clearly one of these algorithms might be more efficient than the other. In any case, here we have a clear case of only one formula but at least two (and possibly 4) distinct algorithms.

Perhaps an even clearer example is function E, above—the one that tells you when Easter occurs. I dare you to try to use this formula to find out when Easter will occur next year! Where would you even begin? To use it, you would need an *algorithm*, such as the one at http://tinyurl.com/yb9jvbpl.[8] (A related matter is knowing whether the formula is even correct! We'll explore this issue in Chapter 16.)[9]

Some functions expressed as formulas might be seen as containing an implicit algorithm for how to compute them:

> [A] term in the series for arctan 1/5 can be written either as $(1/5)^m/m$ or as $1/(m5^m)$. Mathematically these expressions are identical, but they imply different computations. In the first case you multiply and divide long decimal fractions;

[8] http://techsupt.winbatch.com/webcgi/webbatch.exe?techsupt/nftechsupt.web+WinBatch/How~To+Easter~finder.txt. If neither of these links work, do the following: Link to http://techsupt.winbatch.com/; then search for "Easter finder".

[9] I created this formula by working backwards from the algorithm given in (Stewart, 2001), so it's quite possible that I introduced a typographical error! Even if I didn't, I am assuming that (Stewart, 2001)'s algorithm is correct. And that could be a big assumption.

> in the second you build a large integer and then take its reciprocal.
> —Brian Hayes (2014a, p. 344)

But these formulas can only be interpreted as algorithms with additional information about the order of operations (roughly, do things in innermost parentheses first, then do exponentiations, then multiplication and division from left to right, then addition and subtraction from left to right).

Further Reading:

1. For a good discussion of the difference between formulas and algorithms, see the question asked in the epigraph to this section, and the answers in (Huber, 1966) and (Knuth, 1966). Knuth's answer is a commentary on Huber's. Huber's answer, roughly, is that an algorithm is a set of instructions for computing the value of a function by "executing" (or carrying out, or following) the instructions, whereas a formula is an expression describing the value of a function; it can be "evaluated" (that is, the value of the function can be determined from the formula) but not executed (because a formula does not come equipped with an algorithm for telling you *how* to evaluate it). In a sense, a formula is "static", whereas an algorithm is (potentially) "dynamic".

2. Turing-award winner Judea Pearl ...

   > ... considers the difference between two representations of simple algebraic relations, which he terms "equations versus diagrams," contrasting:
   >
   > $$Y = 2XZ = Y + 1$$
   >
   > with
   >
   > $$X \rightarrow \times 2[Y] \rightarrow +1[Z]$$
   >
   > The former describes relations between variables; the latter specifies a simple computer program, in the form of a flowchart, indicating the order in which operations are carried out.
   > —(Chater and Oaksford, 2013, p. 1172), citing (Pearl, 2000)

   The interpretation of the flowchart version is something like this:

   (a) input X

   (b) multiply X by 2; store in Y

   (c) add 1 to Y; store in Z

   Note that this algorithm does not appear to have an output! See §7.4 for discussion of this.

Functions describable by formulas are not the only kind of functions. There are functions without formulas for computing them. (To revert to our machine metaphor, there are functions such that the "gears" of their "machines" work by magic!) For example, there are "table look-up" functions, where the only way to identify the correct output for a given input is to look it up in a table (rather than to compute it); usually, this is the case when there is no lawlike pattern relating the inputs and the outputs. Of course, there are non-computable functions, such as the Halting Problem (we'll have more to say on what this in §7.7). And there are random functions.

As we saw in §3.15.2, one of the central purposes—perhaps *the* central question—of CS is to figure out which functions *do* have algorithms for computing them! (If functions defined extensionally are "magic", then functions defined intensionally are "magic *tricks*".) This includes "non-mathematical" functions, such as the (human) cognitive "functions" that take as input sensory information from the environment and produce as output (human, cognitive) behavior. To express this another way, the subfield of CS known as AI can be considered as having as its purpose figuring out which such cognitive functions are computable.

7.3.1.5 Computable Functions

So we have two central concepts: function and algorithm. We have given a careful definition of the mathematical notion of function. We have not yet given a careful definition of the mathematical notion of algorithm, but we have given some informal characterizations (and we will look at others in §7.4, below). We can combine them as follows:

> A function f **is computable** will mean, roughly, that there is an "algorithm" that *computes* f.

This is only a rough definition or characterization because, for one thing, we haven't yet defined 'algorithm'. But, assuming that an algorithm is, roughly, a set of instructions for computing the output of the function, then it makes sense to define a function as being computable if we can ... well ... compute it! So:

> A function f **is computable** iff there is an algorithm A_f such that, for all inputs i, $A_f(i) = f(i)$.

That is, a function f is computable by an algorithm A_f if both f and A_f have the same input-output "behavior" (that is, if both define the same binary relation, or set of input-output pairs). Moreover, A_f must specify *how* f's inputs and outputs are related. So, whereas a function only *shows* its input-output pairs but is silent about *how* they are related, an algorithm for that function must say more. It must be a procedure, or a mechanism, or a set of intermediate steps or instructions that transforms the input into the output, or shows you explicitly how to find the output by starting with the input or how to get from the input to the output. Algorithms shouldn't be magic or merely arbitrary.[10]

[10] Except possibly in the "base case", where the "algorithm" is so simple or basic that it consists merely in giving you the output directly, without any intermediate processing. (See the box in §7.5.3, below, for an explanation of "base case".)

It seems easy enough to give examples of algorithms for some of the functions listed earlier:

1. $A_f(i) =$

   input i;
   multiply i by 2;
   output result.

2. $A_g(i) =$

   input i;
   add 1 to i;
   output result.

3. $A_{g'}(i) =$

   input i;
   multiply i by 2;
   call the result x;
   subtract i from x;
   add 3 to 4;
   call the result y;
   divide 7 by y;
   add x to y;
   output result.

4. For $E(m,d)$, see the English algorithm in (Stewart, 2001) or the computer program online at the URL given in §7.3.1.4, above. Note that, even though that algorithm may not be easy to follow, it is certainly much easier than trying to compute the output of E from the formula. (For one thing, the algorithm tells you where to begin!)

5. $A_k(i) =$

   if $i \neq 0$
   then
   begin
   divide 1 by i;
   output result
   end.

   Note that this algorithm doesn't tell you what to do if $i = 0$, because there is no "**else**"-clause. So, what would happen if you input 0? Because the algorithm is silent about what to do in this case, anything might happen! If it were implemented on a real computer, it would probably "hang" (that is, do nothing), or crash, or go into an infinite loop.

Question for the Reader:
The philosopher Richard Montague suggests that—for a more general notion of computation than a mere Turing machine (one that would apply to both digital and analog computation)—a computer needs an output signal that indicates when the computation is finished (Montague, 1960, p. 433). As we will see in Chapter 8, in Turing's theory of computation, the machine simply halts.

How do you know that a machine has halted rather than merely being in an infinite loop? What is the difference between a program *halting* and a program *hanging*?

Good programming technique would require that the program be rewritten to make it "total" instead of "partial", perhaps with an error handler like this:

$A'_k(i) =$
if $i \neq 0$
then
begin
divide 1 by i;
output result
end
else output "illegal input".

Question for the Reader:
Is $A'_k(i)$ merely a different algorithm for function k,
or is it really an algorithm for a *different function* (call it k')?

Can this notion of algorithm be made more precise? *How?*

7.4 'Algorithm' Made Precise

(This section is adapted from (Rapaport, 2012b, Appendix).)

> The meaning of the word *algorithm*, like the meaning of most other words commonly used in the English language, is somewhat vague. In order to have a *theory of algorithms*, we need a mathematically precise definition of an algorithm. However, in giving such a precise definition, we run the risk of not reflecting exactly the intuitive notion behind the word. —(Herman, 1983, p. 57)

7.4.1 Ancient Algorithms

Before anyone attempted to define 'algorithm', many algorithms were in use by mathematicians—for example, ancient Babylonian procedures for finding lengths and for computing compound interest (Knuth, 1972a), Euclid's procedures for construction of geometric objects by compass and straightedge (Toussaint, 1993), and Euclid's algorithm for computing the greatest common divisor of two integers. And algorithms were also used

by ordinary people—for example, the algorithms for simple arithmetic with Hindu-Arabic numerals (Robertson, 1979). In fact, the original, eponymous use of the word referred to those arithmetic rules as devised by Abu Abdallah Muhammad ibn Musa Al-Khwarizmi, a Persian mathematician who lived around 780–850 CE (about 1200 years ago).

Further Reading:
What looks as if it might be his last name—'Al-Khwarizmi'—really just means something like "the person who comes from Khwarizm", a lake that is now known as the Aral Sea (Knuth, 1985, p. 171). See (Crossley and Henry, 1990) and (O'Connor and Robertson, 1999) for more on Al-Khwarizmi and his algorithms.

Were the ancient Babylonians really creating algorithms? Insofar as what they were doing fits our informal notion of algorithm, the answer looks to be: yes. But CS historian Michael Mahoney cautions against applying 20th-century insights to ancient times:

> When scientists study history, they often use their modern tools to determine what past work was "really about"; for example, the Babyonian mathematicians were "really" writing algorithms. But that's precisely what was not "really" happening. What was really happening was what was possible, indeed imaginable, in the intellectual environment of the time; what was really happening was what the linguistic and conceptual framework then would allow. The framework of Babylonian mathematics had no place for a metamathematical notion such as algorithm.
> —(Mahoney, 2011, p. 39)

Mahoney cites computer scientist Richard Hamming as making the same point in an essay on the history of computing, that "we would [like to] know what they thought when they did it": What were Babylonian mathematicians thinking when they created what *we now* call "algorithms"? But is that fair? Yes, it would be nice to know what they were really thinking, but isn't it also the case that, whatever *they* thought they were doing, *we* can describe it in terms of algorithms?

7.4.2 "Effectiveness"

When David Hilbert investigated the foundations of mathematics, his followers began to try to make the notion of algorithm precise, beginning with discussions of "effectively calculable", a phrase first used by Jacques Herbrand in 1931 (Gandy, 1988, p. 68) and later taken up by Alonzo Church (Church, 1936b) and his student Stephen Kleene (Kleene, 1952), but left largely undefined, at least in print.

Further Reading:
(Church, 1956) calls 'effective' an "informal notion": See p. 50 (and §10.4.1, below); p. 52 (including note 118 ["an effective method of calculating, especially if it consists of a sequence of steps with later steps depending on results of earlier ones, is called an *algorithm*"] and note 119 ["an effective method of computation, or algorithm, is one for which it woud be possible to build a computing machine", by which he means a Turing machine]); p. 83; p. 99, note 183 ["a procedure ... should not be called effective unless there is a predictable upper bound of the number of steps that will be required"]; and p. 326, note 535). See also (Manzano, 1997, pp. 219–220) and Sieg 1997. Another explication of 'effective' is in (Gandy, 1980, p. 124), which we'll return to in Chapter 10. For another take on 'effective', see (Copeland, 2000).

Another of Church's students, J. Barkley Rosser made an effort to clarify the contribution of the modifier 'effective':

> "Effective method" is here used in the rather special sense of a method each step of which is [1] *precisely predetermined* and which is [2] *certain to produce the answer* [3] in a *finite number of steps*.
> —(Rosser, 1939, p. 55, my italics and enumeration)

But what, exactly, does 'precisely predetermined' mean? And does 'finite number of steps' mean (a) that the written statement of the algorithm has a finite number of instructions or (b) that, when executing them, only a finite number of tasks must be performed? In other words, what gets counted: written steps or executed instructions? One written step—"**for** $i := 1$ to 100 **do** $x := x+1$"—can result in 100 executed instructions. And one written step—"**while** true **do** $x := x+1$"—can even result in infinitely many executed instructions! Here is what Hilbert had to say about finiteness:

> It remains to discuss briefly what general requirements may be justly laid down for the solution of a mathematical problem. I should say first of all, this: that it shall be possible to establish the correctness of the solution by means of a finite number of steps based upon a finite number of hypotheses which are implied in the statement of the problem and which must always be exactly formulated. This requirement of logical deduction by means of a finite number of processes is simply the requirement of rigor in reasoning. —(Hilbert, 1900, pp. 440–441)

7.4.3 Three Attempts at Precision

> Leibniz isolated some general features of algorithmic procedures .... [A]n algorithmic procedure must *determine* completely what actions have to be undertaken by the computing agent. ... the instructions of a calculation procedure can be viewed as prescribing *operations on symbolic expressions* in general, and not just on numerical expressions. ... only physical properties of symbols—such as their shape and arrangement—and not, for example, their meaning, play a role in a calculation process. ... only elementary intellectual capabilities are required on the part of the executor of a calculation procedure ... .
> —(Spruit and Tamburrini, 1991, pp. 7–8).

Much later, *after* Turing's, Church's, Gödel's, and Post's precise formulations and *during* the age of computers and computer programming, slightly less vague, though still informal, characterizations were given by A.A. Markov (a Russian mathematician), Stephen Kleene, and Donald Knuth.

7.4.3.1 Markov

According to (Markov, 1954, p. 1), an algorithm is a "computational process" satisfying three (informal) properties:

1. being "determined"
   ("carried out according to a precise prescription ... leaving no possibility of arbitrary choice, and in the known sense generally understood"),

2. having "applicability"
   ("The possibility of starting from original given objects which can vary within known limits"), and

3. having "effectiveness"
   ("The tendency of the algorithm to obtain a certain result, finally obtained for appropriate original given objects").

These are a bit obscure: Being "determined" may be akin to Rosser's "precisely predetermined". But what about being "applicable"? Perhaps this simply means that an algorithm must not be limited to converting one specific input to an output, but must be more general. And Markov's notion of "effectiveness" seems restricted to only the second part of Rosser's notion, namely, that of "producing the answer". There is no mention of finiteness, unless that is implied by being computational.

7.4.3.2 Kleene

In his logic textbook for undergraduates, (Kleene, 1967) elaborates on the notions of "effective" and "algorithm". He identifies "effective procedure" with "algorithm" (Kleene, 1967, p. 231), characterizing an algorithm as

1. a "procedure" (that is, a "finite" "set of rules or instructions") that ...
2. "in a finite number of steps" answers a question, where ...
3. each instruction can be "followed" "mechanically, like robots; no insight or ingenuity or invention is required", ...
4. each instruction "tell[s] us what to do next", and ...
5. the algorithm "enable[s] us to recognize when the steps come to an end" (Kleene, 1967, p. 223).

And, in a later essay, Kleene writes:

> [a] ... a method for answering any one of a given infinite class of questions ... is given by a set of rules or instructions, describing a procedure that works as follows. [b] *After* the procedure has been described, [then] if we select *any* question from the class, the procedure will then tell us how to perform successive steps, so that after a finite number of them we will have the answer to the question selected. [c] In particular, immediately after selecting the question from the class, the rules or instructions will tell us what step to perform first, unless the answer to the question selected is immediate. [d] After our performing any step to which the procedure has led us, the rules or instructions will *either* enable us to recognize that now we have the answer before us and to read it off, *or else* that we do not yet have the answer before us, in which case they will tell us what step to perform next. [e] In performing the steps, we simply follow the instructions like robots; no ingenuity or mathematical invention is required of us.
> —(Kleene, 1995, p. 18, enumeration added)

So, for Kleene in 1995, an algorithm (informally) is:

a A *set* of *rules or instructions* that describes *a procedure*. The procedure is one thing; its description is another: The latter is a set of imperative sentences.

b Given a class of questions Q and a procedure P_Q for answering any member of Q: $(\forall q \in Q)[P_Q$ gives a *finite sequence* of steps (described by its rules) that answers $q]$. So, the finiteness occurs in the *execution* of P_Q (not necessarily in P_Q itself). And P_Q does not depend on q, only on Q, which suggests, first, that the algorithm must be *general*, and not restricted to a single question. (An algorithm for answering '$2+3=$?' must also be able to answer all questions of the form '$x+y=$?'.) And, second, it suggests that an algorithm has a goal, purpose, or "function" (in the sense of §7.3.1.1, above). That is, the algorithm must not just be a set of instructions that *happens* to answer the questions in Q; it must be designed for that purpose, because it *depends on* what Q is.

Philosophical Digression:
Such a goal, purpose, or function is said to be an "intentional" property. We'll come back to this important issue in Chapter 17 (see also (Rapaport, 2017a)). "Inten**t**ionality" spelled with a 't' is distinct from—but related to—"inten**s**ionality" spelled with an 's'; see (Rapaport, 2012a).

c The algorithm takes question q as input, and either outputs q's answer ("base case"), or else outputs the first step to answer q ("recursive case").[11]

d If it is the "recursive case", then, presumably, q has been reduced to a simpler question and the process repeats until the answer is produced as a base case.

Moreover, the answer is immediately recognizable. That does not necessarily require an intelligent mind to recognize it. It could be merely that the algorithm halts with a message that the output is, indeed, the answer. In other words, the output is of the form: The answer to q is q_i, where q_i is the answer and the algorithm halts, or q_i is a one-step-simpler question and then the algorithm tells us what the next step is. In a so-called "trial-and-error machine" (to be discussed in §11.4.5), the output is of the form: My current guess is that the answer to q is q_i and then the algorithm tells us what the next step is. (We'll see such an algorithm in §7.7.)

e The algorithm is "complete" or "independent", in the sense that it contains all information for executing the steps. We, the executor, do not (have to) supply anything else. In particular, we do not (have to) accept any further, unknown or unforeseen input from any other source (that is, no "oracle" or interaction with the external world.) We'll return to these ideas in Chapters 11 and 17.

7.4.3.3 Knuth

Donald Knuth goes into considerably more detail, albeit still informally (Knuth, 1973, "Basic Concepts: §1.1: Algorithms", pp. xiv–9, esp. pp. 1–9). He says that an algorithm is "a finite set of rules which gives a sequence of operations for solving a specific type of problem", with "five important features" (Knuth, 1973, p. 4):

1. "*Finiteness*. An algorithm must always terminate after a finite number of steps" (Knuth, 1973, p. 4).

   > Note the double finiteness: A finite number of rules in the text of the algorithm *and* a finite number of steps to be carried out. Moreover, algorithms must halt. (Halting is not guaranteed by finiteness; see point 5, below.)

   Interestingly, Knuth also says that an algorithm is a *finite* "computational method", where a "computational method", more generally, is a "procedure", which only has the next four features (Knuth, 1973, p. 4):

[11] See the box in §7.5.3, below, for an explanation of these scare-quoted terms.

Further Reading:
Hopcroft and Ullman (1969, pp. 2–3) distinguish a "procedure", which they vaguely define (their terminology!) as "a finite sequence of instructions that can be mechanically carried out, such as a computer program" (to be formally defined in their chapter on Turing machines) from an "algorithm", which they define as "a procedure which always terminates".

2. "*Definiteness*. Each step ... must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified ..." (Knuth, 1973, p. 5).

   This seems to be Knuth's analogue of the "precision" that Rosser and Markov mention. (For humorous takes on precision and unambiguousness, see Figures 7.4, 7.5, and 7.6.)

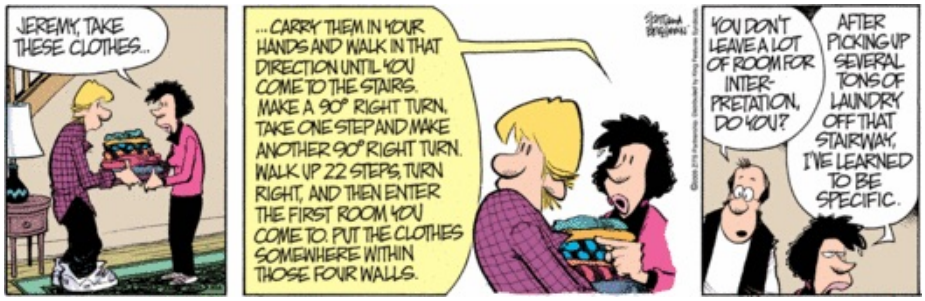


Figure 7.4: ©14 March 2009, Zits Partnership

3. "*Input.* An algorithm has zero or more inputs" (Knuth, 1973, p. 5).

   Curiously, only Knuth and Markov seem to mention this explicitly, with Markov's "applicability" property suggesting that there must be at least *one* input. Why does Knuth say *zero* or more? If algorithms are procedures for computing functions, and if functions are sets of input-output *pairs*, then wouldn't an algorithm *always* have to have input? Presumably, Knuth wants to allow for the possibility of a program that simply outputs some information. Perhaps Knuth has in mind the possibility of the input being internally stored in the computer rather than having to be obtained from the external environment. An example of this[12] would be an algorithm for computing the nth digit in the decimal expansion of a real number: There do not need to be any explicit inputs; the algorithm can just generate each digit in turn. Or perhaps this is how constant functions (functions whose output is constant, no matter what their input is) are handled. (We'll come back to this in §11.4.3.1.) It is worth noting, however, that

[12] Which Matti Tedre reminded me of (personal communication, 2018).

Figure 7.5: ©1992 King Features Syndicate

(Hartmanis and Stearns, 1965, p. 288)—the founding document of the field of computational complexity—allows their multi-tape Turing machines to have at most one tape, which is an output-only tape; there need not be any input tapes. And, if there is only *at most* one output tape, there need not be *any* input or output at all!

4. "*Output.* An algorithm has one or more outputs" (Knuth, 1973, p. 5).

That there must be at least one output echoes Rosser's property (2) ("certain to produce the answer") and Markov's notion (3) of "effectiveness" ("a certain result"). But Knuth characterizes outputs as "quantities which have a specified relation to the inputs" (Knuth, 1973, p. 5): The "relation" would no doubt be the functional relation between inputs and outputs, but, if there is no input, what kind of a relation would the output be in?

Further Reading:
For an example of an algorithm that has an input but no output, see the box in §7.3.1.4, above. See also (Copeland and Shagrir, 2011, pp. 230–231).

Figure 7.6: A real-life example of an ambiguous instruction. (Whose head should be removed?)

Others have noted that, while neither inputs nor outputs are *necessary*, they are certainly useful:

> There remains, however, the necessity of getting the original definitory information from outside into the device, and also of getting the final information, the results, from the device to the outside.
> —(von Neumann, 1945, §2.6, p. 3).

> Do computations have to have inputs and outputs? The mathematical resources of computability theory can be used to define 'computations' that lack inputs, outputs, or both. But the computations that are generally relevant for applications are computations with both inputs and outputs.
> —(Piccinini, 2011, p. 741, note 11)

The computer has to have something to work on ("definitory information", or input), and it has to let the human user know what it has computed ("the final information, the results", or output). It shouldn't just sit there silently computing. In other words, there has to be input and output if the computer is not to be "solipsistic".

Philosophical Digression:
Solipsism is, roughly, the view that I am the only thing that exists, or that I (or my mind) cannot have knowledge of the external world. So, a computer with no input or output would only have "knowledge" of things "inside" itself. For more on solipsism, see http://www.iep.utm.edu/solipsis/ or https://en.wikipedia.org/wiki/Solipsism

Allen Newell has suggested that there must be input iff there is output:

> **Read** is the companion process to **write**, each being necessary to make the other useful. **Read** only obtains what was put into expressions by **write** at an earlier time; and a **write** operation whose result is never **read** subsequently might as well not have happened. —(Newell, 1980, p. 163)

However, there are circumstances where **read** would take input from the external world, not necessarily from previous output. And the last clause suggests that, while output is not necessary, it is certainly useful!

5. "*Effectiveness*. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man [sic] using pencil and paper" (Knuth, 1973, p. 6).

   Note, first, how the term 'effective' has many different meanings among all these characterizations of "algorithm", ranging from it being an unexplained term, through being synonymous with 'algorithm', to naming very particular—and very different—properties of algorithms.

   Second, it is not clear how Knuth's notion of effectiveness differs from his notion of definiteness; both seem to have to do with the preciseness of the operations.

   Third, Knuth brings in another notion of finiteness: finiteness in time. Note that an instruction to carry out an infinite sequence of steps in a finite time could be accomplished by doing each step twice as fast as the previous step; or each step might only take a finite amount of time, but the number of steps required might take longer than the expected life of the universe (as in computing a perfect, non-losing strategy in chess (Zobrist, 2000, p. 367)). These may have interesting theoretical implications (which we will explore in Chapter 11) but do not seem very practical. (Knuth, 1973, p. 7) observes that "we want *good* algorithms in some loosely-defined aesthetic sense. One criterion of goodness is the length of time taken to perform the algorithm . . . ."

   Finally, the "gold standard" of "a [hu]man using pencil and paper" seems clearly to be an allusion to Turing's analysis (Turing, 1936), which we will examine in great detail in the next chapter.

7.4.3.4 Summary

We can summarize these informal observations as follows:

An algorithm (for executor E to accomplish goal G) is:

1. a procedure, that is, a finite set (or sequence) of statements (or rules, or instructions), such that each statement is:
   (a) composed of a finite number of symbols (or marks) from a finite alphabet
   (b) and unambiguous for E—that is,
      i. E knows how to do it
      ii. E can do it
      iii. it can be done in a finite amount of time
      iv. and, after doing it, E knows what to do next—
2. which procedure takes a finite amount of time (that is, it halts),
3. and that ends with G accomplished.

But the important thing to note is that the more one tries to make precise these *informal* requirements for something to be an algorithm, the more one recapitulates Turing's motivation for the formulation of a Turing machine. In Chapter 8, we will look at exactly what Turing did.

But first we are going to look a bit more deeply into the current view of computation.

Further Reading:
For more on the attempts to make the notion of "algorithm" precise, see (Korfhage, 1993), (Sieg, 1994) (which contains a detailed history and analysis of the development of the formal notions of algorithm in the 1930s and after) (Copeland, 1996), (Copeland, 1997) (an essay on hypercomputation (or "non-classical" computation), but the introductory section (pp. 690–698) contains an enlightening discussion of the scope and limitations of Turing's accomplishments), (Sieg, 1997), (Moschovakis, 1998), (Moschovakis, 2001), (Copeland, 2004a), (Blass and Gurevich, 2003), (Chazelle, 2006), (Sieg, 2008), (Gurevich, 2011), and (Hill, 2013).

In Chapter 13, we will be looking at whether computer programs can be copyrighted or patented. In order to answer this question, many legal experts have tried to give a definition of 'algorithm'. One such attempt is (Chisum, 1986).

(Farkas, 1999) contains advice on how to write informal procedures.

7.5 Five Great Insights of CS

In this section, we will revisit in detail the five great insights of CS that were introduced in §3.15.2.1.1. The first three help make precise the vague notion of algorithm that we have been looking at. The fourth links the vague notion with a precise one. Together, they define the smallest possible language in which you can write any procedure for any computer. (And by 'computer' here, I merely mean anything—machine or human—that can execute an algorithm.) The fifth brings in engineering concerns.

7.5.1 Bacon's, Leibniz's, Morse's, Boole's, Ramsey's, Turing's, and Shannon's *Representational Insight*

The first great insight is this:

> **All the information about any computable problem can be represented using only *two nouns*: '0' and '1'**

Rather than enter into a complicated and controversial historical investigation of who is responsible for this insight, I will simply list the names of some of the people who contributed to it:

- Sir Francis Bacon, around 1605, developed an encoding of the alphabet by any objects "capable of a twofold difference".[13] And, of course, once you've represented the *alphabet* in a binary coding, then anything capable of being represented in *text* can be similarly encoded (Quine, 1987, "Universal Library", pp. 223–235, https://urbigenous.net/library/universal_library.html).
- Our old friend Leibniz gave an "Explanation of Binary Arithmetic" in 1703.[14]
- Famously, Samuel F.B. Morse not only invented the telegraph but also (following in Bacon's footsteps) developed his eponymous, binary code in the mid-1800s.[15].
- Going beyond language, the philosopher Frank P. Ramsey, in a 1929 essay on "a language for discussing ... facts"—perhaps something like Leibniz's *characteristica universalis* (which we discussed in §§3.17 and §6.6)—suggested that "all [of the terms of the language] may be best symbolized by numbers. For instance, colours have a structure, in which any given colour may be assigned a place by three numbers .... *Even smells may be so treated* ..." (Ramsey, 1929, pp. 101–102, my italics). (For more examples, see http://www.cse.buffalo.edu/~rapaport/111F04/greatidea1.html)
- In 1936, as we will see in Chapter 8, Turing made essential use of '0' and '1' in the development of Turing machines.

[13] Bacon, *Advancement of Learning*, http://home.hiwaay.net/~paul/bacon/advancement/book6ch1.html; for discussion, see (Cerf, 2015, p. 7).

[14] http://www.leibniz-translations.com/binary.htm

[15] http://en.wikipedia.org/wiki/Morse_code. Arguably, however, Morse code (traditionally conceived as having only two symbols, "dot" and "dash") is not strictly binary, because there are "blanks", or time-lapses, between those symbols (Gleick 2011, p. 20, footnote; Bernhardt 2016, p. 29

- Finally, the next year, Claude Shannon (in his development of the mathematical theory of information) used "The symbol 0 ... to represent ... a closed circuit, and the symbol 1 ... to represent ... an open circuit" (Shannon, 1937, p. 4), and then showed how propositional logic could be used to represent such circuits. Moreover,

> Up until that time [that is, the time of publication of Shannon's "Mathematical Theory of Communication" (Shannon, 1948), everyone thought that communication was involved in trying to find ways of communicating written language, spoken language, pictures, video, and all of these different things—that all of these would require different ways of communicating. Claude said no, *you can turn all of them into binary digits.* And then you can find ways of communicating the binary digits. —Robert Gallager, quoted in (Soni and Goodman, 2017)

There is nothing special about the *symbols* '0' and '1'. As Bacon emphasized, any other bistable[16] pair suffices, as long as they can flip-flop between two easily distinguishable states, such as the *numbers* 0 and 1, "on/off", "magnetized/de-magnetized", "high voltage/low voltage", etc.

Digression:
Bacon used 'a' and 'b', but he also suggested that coding could be done "by Bells, by Trumpets, by Lights and Torches, by the report of Muskets, and any instruments of like natures", http://home.hiwaay.net/~paul/bacon/advancement/book6ch1.html

Strictly speaking, these can be used to represent *discrete* things; *continuous* things can be approximated to any desired degree, however.

This limitation to *two* nouns is not necessary: Turing's original theory had no restriction on how many symbols there were. There were only restrictions on the nature of the symbols (they couldn't be too "close" to each other; that is, they had to be distinguishable) and that there be only finitely many.

But, if we want to have a minimal language for computation, having only two symbols *suffices*, and making them '0' and '1' (rather than, say, 'a' and 'b'—not to mention "the report of Muskets"!) is *mathematically convenient*.

Further Reading:
Chaitin 2006a "discuss[es] mathematical and physical arguments against continuity and in favor of discreteness".

See (Cerf, 2014) for some interesting comments that are relevant to the insight about binary representation of information.

On Shannon, see Horgan 1990, G. Johnson 2001c, Cerf 2017, and Soni and Goodman 2017.

[16] That is, something that can be in precisely one of two states; http://en.wikipedia.org/wiki/Bistability.

7.5.2 Turing's *Processing Insight*

So we need only two *nouns* for our minimal language. Turing is also responsible for providing the *verbs* of our minimal language. Our second great insight is this:

> Every algorithm can be expressed in a language for a computer (namely, a Turing machine) consisting of:
>
> - an arbitrarily long, paper tape divided into squares (like a roll of toilet paper, except you never run out (Weizenbaum, 1976)),
> - with a read/write head,
> - whose only nouns are '0' and '1',
> - **and whose only five verbs (or basic instructions) are:**
>
> 1. **move-left-1-square**
> 2. **move-right-1-square**
> 3. **print-0-at-current-square**
> 4. **print-1-at-current-square**
> 5. **erase-current-square**

The exact verbs depend on the model of Turing machine.[17] The two "move" instructions could be combined into a single verb that takes a direct object (that is, a function that takes a single input argument): move(location). And the "print" and "erase" instructions could be combined into another single transitive verb: print(symbol), where "symbol" could be either '0', '1', or 'blank' (here, erasing would be modeled by printing a blank). (We'll see Turing do something similar, when we look at Turing machines in the next chapter (§8.13).)

Further Reading:
Wang (1957, p. 80) notes that

> there are many things which we can do when we permit erasing but which we cannot do otherwise. Erasing is dispensable only in the sense that all functions which are computable with erasing are also computable without erasing. For example, if we permit erasing, ... only the ... answer appears on the tape at the end of the operation, everything else having been erased.

Deciding how to count the number of verbs is an interesting question. In the formulation above, do we have 3 nouns ('0', '1', 'blank') and only 1 transitive verb ('print(symbol)')? Or do we have only 2 nouns ('0', '1') but 2 verbs ('print(symbol)', 'erase')? (Gurevich, 1999, pp. 99–100) points out that

[17] The ones cited here are taken from John Case's model described in (Schagrin et al., 1985, Appendix B, http://www.cse.buffalo.edu/~rapaport/Papers/schagrinetal85-TuringMachines.pdf).

> at one step, a Turing machine may change its control state, print a symbol at the current tape cell[,] and move its head. ... One may claim that every Turing machine performs only one action a time, but that action [can] have several parts. The number of parts is in the eye of the beholder. You counted three parts. I can count four parts by requiring that the old symbol is erased before the new symbol is printed. Also, the new symbol may be composed, e.g. '12'. Printing a composed symbol can be a composed action all by itself.

And Fortnow (2018b) suggests that there are four verbs: move left, move right, read, write.

In any case, we can certainly get by with only two (slightly complex) verbs or five (slightly simpler) verbs. But either version is pretty small.

Further Reading:
(Wang, 1957, p. 63) "offer[s] a theory which is closely related to Turing's but is more economical in the basic operations. ... [A] theoretically simple basic machine can be ... specified such that all partial recursive functions (and hence all solvable computation problems) can be computed by it and that only four basic types of instruction are employed for the programs: shift left one space, shift right one space, mark a blank space, conditional transfer. ... [E]rasing is dispensable, one symbol for marking is sufficient, and one kind of transfer is enough. The reduction is ... similar to ... the definability of conjunction and implication in terms of negation and disjunction ...."

A version of Wang's machine, with many examples, is given in Dennett 2013a, Ch. 24.

7.5.3 Böhm & Jacopini's *Structural Insight*

We have two nouns and perhaps only two verbs. Now we need some grammatical rules to enable us to put them together. The software-engineering concept of "structured programming" does the trick. This is a style of programming that avoids the use of the 'go to' command. In early programming languages, programmers found it useful to "go to"—or to "jump" to—another location in the program, sometimes with the ability to return to where the program jumped *from* (but not always). This resulted in what was sometimes called "spaghetti code", because, if you looked at a flowchart of the program, it consisted of long, intertwined strands of code that were hard to read and harder to ensure that they were correct (see Figures 7.7 and 7.8). Edsger W. Dijkstra wrote a letter to the editor of the *Communications of the ACM*, that was headlined "Go To Statement Considered Harmful" (Dijkstra, 1968), arguing against the use of such statements. This resulted in an attempt to better "structure" computer programs so that the use of 'go to' could be minimized: Corrado Böhm and Giuseppe Jacopini showed how it could be completely eliminated (Böhm and Jacopini, 1966). This gives rise to the third insight (and the third item needed to form our language):

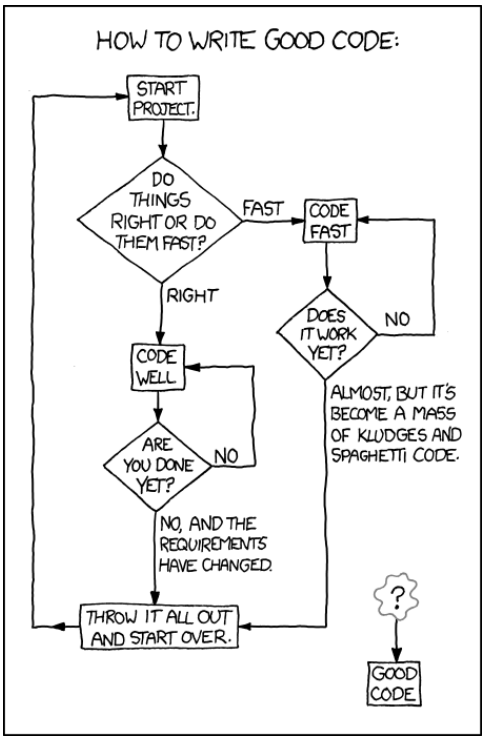


Figure 7.7: https://xkcd.com/844/

Only three *rules of grammar* are needed to combine any set of basic instructions (verbs) into more complex ones:

1. **sequence**:
   - first do this; then do that
2. **selection** (or choice):
   - **if** such-&-such is the case,
     then do this
     then do that
3. **repetition** (or looping):
   - **while** such-&-such is the case **do** this

. . . where "this" and "that" can be:

- any of the basic instructions, or
- any complex instruction created by application of any grammatical rule.

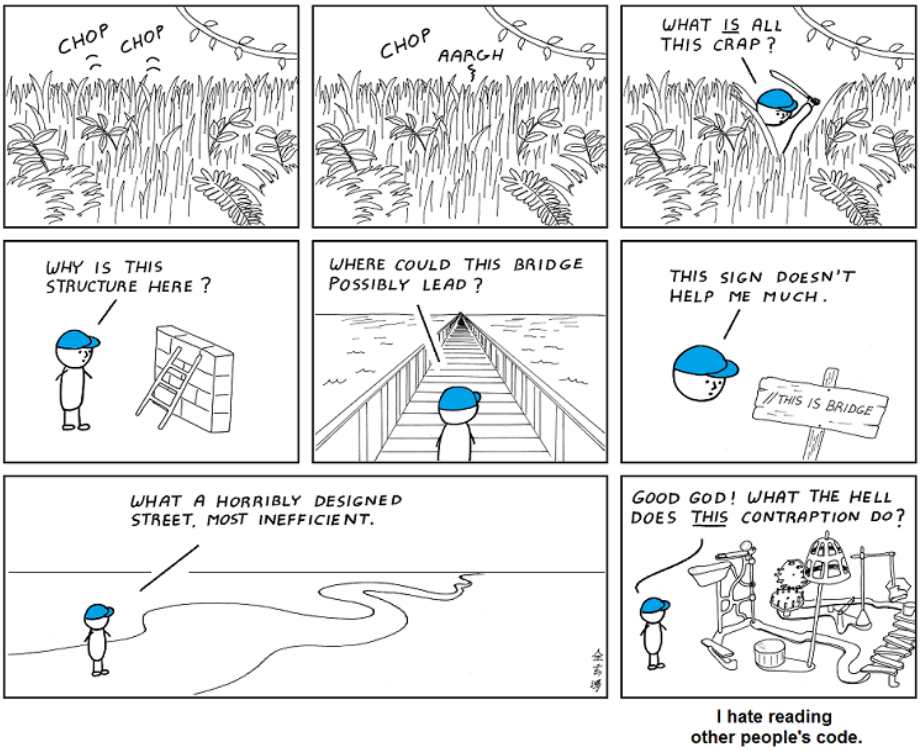


Figure 7.8: http://abstrusegoose.com/432

Digression—Recursive Definitions:
This third insight can be thought of as a "recursive" definition of "instruction".

A recursive definition of some concept *C* consists of two parts. The first part, called the "base case", gives you some explicit examples of *C*. These are not just any old examples, but are considered to be the simplest, or most basic or "atomic", instances of *C*—the building blocks from which all other, more complex instances of *C* can be constructed.

The second part of a recursive definition of *C* consists of rules (algorithms, if you will!) that tell you how to construct those more complex instances of *C*. But these rules don't simply tell you how to construct the complex instances from just the base cases. Rather, they tell you how to construct the complex instances of *C from any instances of C that have already been constructed*. The first complex instances, of course, will be constructed directly from the base cases. But others, even more complex, will be constructed from the ones that were constructed directly from the base cases, and so on. What makes such a definition "recursive" is that simpler instances of *C* "recur" in the definitions of more complex instances.

So, the base case of a recursive definition tells you how to *begin*. And the recursive case tells you how to *continue*.

Recursive definitions can be found outside of CS. Here are two examples:

1. According to some branches of Judaism, a person *p* is Jewish if (a) *p* was converted to Judaism (base case) or (b) *p*'s mother was Jewish (recursive case).
2. "Organisms originate either through synthesis of non-living materials [base case] or through reproduction, either sexual or asexual [recursive case]" (Northcott and Piccinini, 2018, p. 2, my bracketed interpolations).

Recursive definitions sometimes *seem* to be circular: After all, we seem to be defining instances of *C* in terms of instances of *C*! But really we are defining "new" (more complex) instances of *C* in terms of *other*, "older" (that is, already constructed), or simpler instances of *C*, which is not circular at all. (It would only be circular if the *base cases* were somehow defined in terms of themselves. But they are not "defined"; they are given, by fiat.) (For a humorous example of recursion, with "mustard" as the base case, see Figure 7.9.)

So, the structural insight above is a recursive definition of the notion of an "instruction": The base cases of the recursion are the primitive verbs of the Turing machine ('move(location)' and 'print(symbol)'), and the recursive cases are given by sequence, selection, and repetition.

As an analogy, a crossword puzzle can be solved recursively: Begin by filling in those words (or phrases) whose answers you know (for example, a 10-letter word for "first president of the US"). Recursive steps consist in using these "axioms" to "prove theorems", that is, to use the letters from already-filled-in answers as additional clues (or "premises") for new words. This analogy needs to be taken with a grain of salt, however: Some answers that you might know "axiomatically" might also be filled in as "provable theorems". On the other hand, even formal systems can have different axiomatizations, such that an axiom of one formalization might be a theorem of another. What about "cheating" in the sense of looking up an answer? That's just the application of (semantic) table look-up! It is also worth noting that jigsaw puzzles can be solved recursively: The base case of the recursion consists in building the frame. A recursive step is to form a "molecular" piece that consists of two "atomic" pieces that fit together. Further recursions consist of finding two molecular pieces that fit together. (For a different analogy with jigsaw puzzles, see the Digression on Syntax, Semantics, and Puzzles in §14.3.2.3.)

For more on recursion (and its close cousin, induction), see http://www.cse.buffalo.edu/~rapaport/191/recursion.html and Silver 2016 (which is a nice history of mathematical induction and its relation to recursion, although it tends to conflate mathematical induction with inductive logic).

Figure 7.9: ©1994 North America Syndicate

Digression and Further Reading:
Recursion is considered to be the core of the concept of *computation*. It has been argued that it is also the core of the concept of *language*, in particular, and of cognition more generally:

> … the work of Alonzo Church, Kurt Gödel, Emil Post, and Alan Turing, who established the general theory of computability … demonstrated how a finite object like the brain could generate an infinite variety of expressions.
> —(Chomsky, 2017)

> [The] faculty of language … in the narrow sense … only includes recursion and is the only uniquely human component of the faculty of language.
> —(Hauser et al., 2002)

However, this is a highly controversial claim; to follow the debate, see
(Pinker and Jackendoff, 2005), (Fitch et al., 2005), and (Jackendoff and Pinker, 2005).
For another take on this debate, see (Corballis, 2007).

There are optional, additional instructions and grammatical rules:

1. *An explicit "halt" instruction*:

   This is not strictly necessarily, because it can always be simulated by having the program execute a command that does nothing and does not have a "next"; step. We will see such a program when we look at Turing machines, in Chapter 8. However, a "halt" instruction can sometimes make a program simpler or more elegant.

2. *An "exit" instruction*:

   This allows a program to exit from a loop under certain conditions, before the body of the loop is completed. Again, this can provide simplicity or elegance.

3. *Abstraction:*

   A structured programming language ... must provide a mechanism whereby the language can be extended to contain the abstractions which the user requires. A language containing such a mechanism can be viewed as a general-purpose, indefinitely-high-level language. —Liskov and Zilles (1974, p. 51)

   There are two varieties worth noting:

   (a) *Procedural abstraction (or named procedures)*:

   Define new (typically, complex) actions by giving a single name to a (complex) action. This is even more optional than "exit", but it is very powerful in terms of human readability and comprehension, and even in terms of machine efficiency.

   Further Reading:
   We'll see examples of procedural abstraction when we discuss Turing machines (§8.12) and when we discuss the relation of programs to the world (§17.8.2.1). For more information, see Dijkstra 1972, Pylyshyn 1992, and Conery 2010, p. 3. One of the best introductions to it is Pattis et al. 1995; see also Rapaport 2017a, §5.2.2. (And see Figure 7.10 for a humorous version.)

   For more on the power of abstraction, see the first few paragraphs of (Antoy and Hanus, 2010).

   (b) *Abstract Data Types*:

   Procedural abstraction allows the programmer to define new *verbs* in terms of "old" ones. A related technique is the use of abstract data types, which allows the programmer to define new *nouns* in terms of "old" ones (Liskov and Zilles, 1974). Moreover, as is especially clear in object-oriented programming, new "nouns" require new "verbs":

Figure 7.10: http://www.shoecomics.com/, © 5/5/1984? Jefferson Communications

> [A] consequence of the concept of abstract data types is that most of the abstract operations in a program will belong to the sets of operations characterizing abstract types. —Liskov and Zilles (1974, p. 52)

We'll return to abstraction in Chapter 14.

4. *Recursion*:

   > Recursion can be an elegant replacement for repetition: A recursive instruction tells you how to compute the output value of a function in terms of previously computed *output* values instead of in terms of its *input* value. Of course, the base case (that is, the output value of the function for its initial input) has to be given to you in a kind of table-lookup. (We'll look at recursion more closely in §7.6.)

Further Reading:
For a history of the Böhm-Jacopini Theorem, see Harel 1980. For a discussion of structured programming based on sequence, selection, and repetition, along with top-down design and stepwise refinement, with several examples worked out in detail, see Dijkstra 1972, especially §7.

7.5.4 The Church-Turing Computability Thesis

We now have our language: Any algorithm for any computable problem can be expressed in this language (for a Turing machine) that consists of the two nouns '0' and '1', the two verbs 'move(location)' and 'print(symbol)', and the three grammatical rules of sequence, selection, and repetition.

But is it a *minimal* language? In other words, is that really all that is needed? Can your interaction with, say, a spreadsheet program or Facebook be expressed in this simple (if not "simple-minded"!) language? There's no doubt that a spreadsheet

program, for example, that was written in this language would be very long and very hard to read. But that's not the point. The question is: Could it be done? And the answer is our next great insight. In one word, 'yes':

> **The informal notion of computability can be identified with (anything logically equivalent to) Turing-machine computability.**

Another way to put this is to say:

> **Nothing besides our two nouns, two verbs, and three grammar rules are necessary.**

Such a statement, as part of a recursive definition, is sometimes called a "closure" clause.[18]

That is, an algorithm is definable as a program expressible in (anything equivalent to) our minimal language.

This idea was almost simultaneously put forth both by (Church, 1936b) in terms of his lambda calculus and by (Turing, 1936). Consequently, some people call it 'Church's Thesis'; others call it 'Turing's Thesis'; and, as you might expect, some call it 'the Church-Turing Thesis', in part because Turing proved that Church's lambda calculus is logically equivalent to Turing machines. For this reason, Robert Soare (Soare, 2009) has advocated calling it, more simply and more neutrally, the 'Computability Thesis'.

But it is only a proposed definition of 'computable' or 'algorithm': It proposes to identify an *informal*, intuitive notion of effective computability or algorithm with the *formal*, mathematically precise notion of a Turing machine.

Further Reading:
To be clear, I have *not* given such a formal, mathematically precise notion here. For that, see any textbook on computability theory, such as Kleene 1952, Davis 1958, Kleene 1967, Minsky 1967, Rogers 1967, Hopcroft and Ullman 1969, Boolos and Jeffrey 1974, Clark and Cowell 1976, Kfoury et al. 1982, Davis and Weyuker 1983, Cooper 2004, Homer and Selman 2011, or Soare 2016.

How do we know that Turing-machine computability captures (all of) the intuitive notion(s) of effective computability? After all, there are other analyses of computation: For instance, there is Church's analysis in terms of the lambda calculus (see §6.6). Should one of these be preferred over the other? There are two reasons for preferring Turing's over Church's: First, Turing's is easier to understand, because it follows from his analysis of how humans compute. Second—and this is "merely" an appeal to authority—Gödel preferred Turing's analysis, not only to Church's, but also to his own!

Church's theory (the lambda calculus, which John McCarthy later used as the basis of the Lisp programming language) had as its basic, or atomic, steps formal operations on function formulas that some people—Gödel in particular—did not find to be intuitively computable. The same could be said even for Gödel's own theory of recursive functions. But Turing's basic operations were, by design, simple things that any human

[18] http://faculty.washington.edu/keyt/InductiveDefinitions.pdf

could easily do: put a mark at specific location on a piece of paper, and shift attention to a different location.

Further Reading:
For discussion of the appeal to Gödel's authority, see (Shagrir, 2006), which explores why Gödel believed both that "the correct definition of mechanical computabilty was established beyond any doubt by Turing" (Gödel, 1938, p. 168) and that "this definition ... rest[s] on the dubious assumption that there is a finite number of states of mind" (Shagrir, 2006, §1)). (Copeland and Shagrir, 2013) explores both Gödel's interpretations of Turing's work and the relation of the human mind to Turing machines. See also (Sieg, 2006), as well as (Soare, 2009, §2, "Origins of Computabilty and Incomputablity"), which contains a good summary of the history of both Turing's and Gödel's accomplishments.

For more on Gödel, see §6.6.

But the lambda calculus and Turing machines are not the only ones; here is a list of some others:

- Post machines
  (like Turing machines, but treating the tape as a queue;
  see https://en.wikipedia.org/wiki/Post%E2%80%93Turing_machine)
- Markov algorithms
  (later used as the basis of the Snobol programming language;
  see https://en.wikipedia.org/wiki/Markov_algorithm)
- Post productions
  (later used as the basis of production systems in AI;
  see (Post, 1941), (Post, 1943), and (Soare, 2012, p. 3293))
- Herbrand-Gödel recursion equations
  (later used as the basis of the Algol family of programming languages;
  see §7.6.2, below)
- μ-recursive functions (see §7.6.2, below)
- register machines (Shepherdson and Sturgis, 1963)
- any programming language, including, besides those already mentioned, Pascal, C, C+, Java, etc.[19]

There are two major reasons to believe the Computabiity Thesis:

[19] But *not* languages like HTML, which are not "Turing-complete"—that is, not logically equivalent to a Turing machine—usually because they lack one or more of the three grammar rules. Such languages are *weaker* than Turing machines. The question of whether there are models of computation that are *stronger* than Turing machines is the topic of Ch. 11.

1. *Logical evidence:*
   All of the formalisms that have been proposed as precise, mathematical analyses of computability are not only *logically* equivalent (that is, any function that is computable according to one analysis is also computable according to each of the others), but they are *constructively* equivalent (that is, they are inter-compilable, in the sense that you can write a computer program that will translate (or compile) a program in any of these languages into an equivalent program in any of the others). Here is how Turing expressed it in a paper published the year after his magnum opus:[20]

   > Several definitions have been given to express an exact meaning corresponding to the intuitive idea of 'effective calculability' as applied for instance to functions of positive integers. The purpose of the present paper is to show that the computable functions introduced by the author are identical with the λ-definable functions of Church and the general recursive functions due to Herbrand and Gödel and developed by Kleene. It is shown that every λ-definable function is computable and that every computable function is general recursive. ... If these results are taken in conjunction with an already available proof that every general recursive function is λ-definable we shall have the required equivalence of computability with λ-definability ....
   >
   > The identification of 'effectively calculable' functions with computable functions is possibly *more convincing than* an identification with the λ-definable or general recursive functions. For those who take this view the formal proof of equivalence provides a justification for Church's calculus, and allows the 'machines' which generate computable functions to be replaced by the more convenient λ-definitions. —(Turing, 1937, p. 153, my italics)

   Turing cites (Church, 1936b) for the definition of lambda-definability, (Kleene, 1936a) for the definition of general recursiveness, and (Kleene, 1936b) for the proof of their equivalence. Gödel was among those who found Turing's computable functions to be "more convincing than" the other formalisms.

   Further Reading:
   For statements of equivalence of general recursive, μ-recursive, lambda-definable, etc., see Soare 2012, p. 3284. Kleene (1995) shows how to "compile" (or translate the language of) recursive functions into (the language of) Turing machines, that is, how a Turing machine can compute recursive functions.

2. *Empirical evidence:*
   All algorithms that have been devised so far can be expressed as Turing machines; that is, there are no intuitively effective-computable algorithms that are not Turing-machine computable.

But this has not stopped some philosophers and computer scientists from challenging the Computability Thesis. Some have advocated forms of computation that "exceed"

[20] 'λ' is the lower-case, Greek letter "lambda".

Turing-machine computability. We will explore some of these options in Chapters 10 and 11.

Another "objection" to the Computability Thesis (especially in the way that I have presented it) is that

> [C]onflating algorithms with Turing machines is a misreading of Turing's 1936 paper .... Turing's aim was to define *computability*, not algorithms. His paper argued that every function on natural numbers that can be computed by a human computer ... can also be computed by a Turing machine. There is no claim in the paper that Turing machines offer a general model for algorithms. —(Vardi, 2012)

Vardi goes on to cite Gurevich's idea that algorithms are "abstract state machines", whose "key requirement is that one step of the machine can cause only a bounded local change on ... [a] state" (which is an "arbitrary data structure"). This corresponds to Turing's analysis and to our analysis in §7.4. He also cites Moschovakis's idea that algorithms are "recursors": "recursive description[s] built on top of arbitrary operations taken as primitives." This corresponds to recursive functions and the lambda calculus, as we will discus in §7.6, below. And he then observes—in line with the proofs that Turing machines, the lambda calculus, recursive functions, etc., are all logically equivalent—that these *distinct* notions are analogous to the wave-particle duality in quantum mechanics: "An algorithm is *both* an abstract state machine and a recursor, and neither view by itself fully describes what an algorithm is. This *algorithmic duality* seems to be a fundamental principle of computer science."

Can the Computability Thesis be proved? Most scholars say 'no', because any attempt to prove it mathematically would require that the *informal* notion of computability be formalized for the purposes of the proof. Then you could prove that *that formalization* was logically equivalent to Turing machines. But how would you prove that that formalization was "correct"? This leads to an infinite regress.

Further Reading:
Kreisel 1987 is a paper by a well-known logician arguing that Church's Thesis *can* be proved. Similar arguments are made in Stewart Shapiro 1993 and Dershowitz and Gurevich 2008.

For more on the Church-Turing Computability Thesis, see: Stewart Shapiro 1983 (which discusses the notion of computability from a historical perspective, and contains a discussion of Church's thesis), Mendelson 1990, Bringsjord 1993 (which is a reply to Mendelson), Folina 1998, and Piccinini 2007c. Soare 2016, §17.3.3 argues that the Computability Thesis should properly be understood as a "claim with demonstration" and *not* as proposition "in need of continual verification".

Rescorla 2007 argues that *Church's* Thesis (that a number-theoretic function is intuitively computable iff it is recursive) is *not* the same as *Turing's* Thesis (that a number-theoretic function is intuitively computable iff a corresponding string-theoretic function [that represents the number-theoretic one] is computable by a Turing machine). (On representing numbers by strings, see Stuart C. Shapiro 1977.) In an essay on Church's analysis of effective calculability, Sieg 1997 argues that "Turing's analysis is neither concerned with *machine computations* nor with general *human mental processes*. Rather, it is *human mechanical computability* that is being analyzed ..." (p. 171).

Rey 2012 distinguishes between Turing's *thesis* and the Turing *test*.

7.5.5 Turing's, Kay's, Denning's, and Piccinini's *Implementation Insight*

Before turning our attention to a somewhat more mathematical outline of structured programming and recursive functions, after which we will ask whether there are functions that are *non*-computable, there is one more insight:

The first three insights can be physically implemented.

That is, Turing machines can be physically implemented. And, presumably, such a physical implementation would be a computer.

In fact, as Matti Tedre (personal communcation, 2018) pointed out to me, not only can the previous insights be physically implemented, but *this can be done using only one kind of "logic gate"* (either a NOR-gate or a NAND-gate).[21] And, as we have seen, there does not appear to be any limitation on the "medium" of implementation: Most computers today are implemented electronically, but there is work on DNA, optical, etc., computers, and there have even been some built out of Tinker Toys (http://www.computerhistory.org/collections/catalog/X39.81).

Digression and Further Reading:
This was what Turing attempted when he designed the ACE; see §6.5.4. This insight (first suggested to me by Peter Denning, personal communication, 2014), is discussed in great detail in (Piccinini, 2015) and (Piccinini, 2017).

[21] "Nor" and "nand" are connectives of propositional logic, each of which suffices by itself in the sense that all other connectives ("and", "or", "if-then", etc.) can be defined in terms of them. Typically, however, as Tedre pointed out, real computers use several different kinds of gates, for the sake of efficiency.

This brings in the engineering aspect of CS. But it also brings in its train limitations imposed by physical reality: limitations of space, time, memory, etc. Issues concerning what is feasibly or efficiently computable in *practice* (as opposed to what is theoretically computable in *principle*)—complexity theory, worst-case analyses, etc.—and issues concerning the use of heuristics come in here.

Turing Award-winner Alan Kay divides this insight into a "triple whammy of computing":

> 1. Matter can be made to remember, discriminate, decide and do
> 2. Matter can remember descriptions and interpret and act on them
> 3. Matter can hold and interpret and act on descriptions that describe anything that matter can do. —(Guzdial and Kay, 2010)

He later suggests that the third item is the most "powerful", followed by the first, and then the second, and that issues about the limits of computability and multiple realizability are implicit in these.

7.6 Structured Programming and Recursive Functions

(The material in this section is based on lectures given by John Case at SUNY Buffalo around 1983, which in turn were based on (Clark and Cowell, 1976).)

7.6.1 Structured Programming

In §7.5.3, I mentioned "structured programming"—a style of programming that avoids the use of "jump" commands, or the 'go to' command, which Böhm & Jacopini proved could be completely eliminated. Let's see how this can be done.

7.6.1.1 Structured Programs

We can begin with a (recursive) definition of 'structured program': As with all recursive definitions, we need to give a base case (consisting of two "basic programs") and a recursive case (consisting of four "program constructors"). We will use the capital and lower-case Greek letters 'pi' (Π, π) to represent programs.

1. Basic programs:
   (a) The *empty program* π = **begin end.** is a basic (structured) program.
   (b) Let F be a "primitive operation" that is (informally) computable. Then the 1-operation program π = **begin** F **end.** is a basic (structured) program.

   > Note that this definition does not specify which operations are primitive; they will vary with the programming language. One example might be an assignment statement (which will have the form "$y \leftarrow c$", where y is a variable and c is a value that is assigned to y). Another might be the *print* and *move* operations of a Turing machine.
   >
   > Compare the situation with Euclidean geometry: If the primitive operations are limited to those executable using only compass and straightedge, then an angle cannot be trisected. But, of course, if the primitive operations also include measurement of an angle using a protractor, then calculating one-third of an angle's measured size will do the trick.
   >
   > That means that structured programming is a *style* of programming, not a particular programming language. It is a style that can be used with *any* programming language. As we will see when we examine Turing's paper in Chapter 8, he spends a lot of time justifying his choice of primitive operations.

2. Program constructors:

   > The recursive case for structured programs specifies how to construct more complex programs from simpler ones. The simplest ones, of course, are the basic programs: the empty program and the

1-operation programs. So, in good recursive fashion, we begin by constructing slightly more complex programs from these. Once we have both the basic programs and the slightly more complex programs constructed from them, we can combine them—using the recursive constructs below—to form even more complex ones, using the following techniques:

Let π, π' be (simple or complex) programs, each of which contains exactly 1 occurrence of **end**.

Let P be a "Boolean test".

A Boolean test, such as "$x > 0$", is sometimes called a 'propositional function' or 'open sentence'.[22] The essential feature of a Boolean test is that it is a function whose output value is "true" or else it is "false". P must also be (informally) computable (and, again, Turing spends a lot of time justifying his choices of tests).

And let y be an integer-valued variable.

Then the following are also (more complex, structured) programs:

(a) Π = **begin** π; π' **end.** is a (complex) structured program.

Such a Π is the "*linear concatenation*" of π followed by π'. It is Böhm & Jacopini's "sequence" grammar rule.

(b) Π =
```
begin
   if P
      then π
      else π′
end.
```
is a (complex) structured program.

Such a Π is a "*conditional branch*": If P is true, then π is executed, but, if P is false, then π' is executed. It is Böhm & Jacopini's "selection" grammar rule.

(c) Π =
```
begin
   while y > 0 do
      begin
         π;
         y ← y − 1
      end
end.
```
is a (complex) structured program.

[22] It is a "propositional function" because it can be thought of as a function whose input is a proposition, and whose output is a truth value. It is an "open sentence" in the sense that it contains a variable (in English, that would be a pronoun) instead of a constant (in English, that would be a proper name).

Such a Π is called a "*count loop*" (or "for-loop", or "bounded loop"): The simpler program π is repeatedly executed while (that is, as long as) the Boolean test "$y > 0$" is true (that is, until it becomes false). Eventually, it will become false, because each time the loop is executed, y is decremented by 1, so eventually y must become equal to 0. Thus, an infinite loop is avoided. This is one kind of Böhm & Jacopini's "repetition" grammar rule.

(d) Π = **begin**
while *P* **do** π
end.
is a (complex) structured program.

Such a Π is called a "*while-loop*" (or "free" loop, or "unbounded" loop): The simpler program π is repeatedly executed while (that is, as long as) the Boolean test P is true (that is, until P is false). Note that, unlike the case of a count loop, a while loop can be an infinite loop, because there is no built-in guarantee that P will eventually become false (because, in turn, there is no restriction on what P can be, as long as it is a Boolean test). In particular, if P is the constantly-true test "true"—or a constantly-true test such as "1=1"—then the loop will be guaranteed to be infinite. This is a more powerful version of repetition.

7.6.1.2 Two Kinds of Structured Programs

We can classify structured programs based on the above recursive definition:

1. π *is a* ***count-program***
(or a "for-program", or a "bounded-loop program") $=_{def}$
   (a) π is a basic program, or
   (b) π is constructed from count-programs by:
      - linear concatenation, or
      - conditional branching, or
      - count looping
   (c) Nothing else is a count-program.

2. π *is a* ***while-program***
(or a "free-loop program", or an "unbounded-loop program") $=_{def}$
   (a) π is a basic program, or
   (b) π is constructed from while-programs by:
      - linear concatenation, or
      - conditional branching, or
      - count-looping, or

- while-looping

(c) Nothing else is a while-program.

The inclusion of count-loop programs in the construction-clause for while-programs is not strictly needed, because all count-loops are while-loops (just let the P of a while-loop be "$y > 0$" and let the π of the while-loop be the linear concatenation of some other π' followed by "$y \leftarrow y - 1$"). So count-programs are a proper subclass of while-programs: While-programs include all count-programs *plus* programs constructed from while-loops that are not also count-loops.

Informal Mathematical Digression:

1. A function is "*one-to-one*" (or "injective") $=_{def}$ if two of its outputs are the same, then their inputs must have been the same (or: if two inputs differ, then their outputs differ). For example, $f(n) = n+1$ is a one-to-one function. However,

   $$g = \{\langle a, 1\rangle, \langle b, 1\rangle\}$$

   is not a one-to-one function (it is, however, a "two-to-one" function).

2. A function is "*onto*" (or "surjective") $=_{def}$ *everything* in the set of possible outputs "came from" something in the set of possible inputs. For example, $h(n) = n$ is an onto function. However, the one-to-one function $f(n)$ above is not onto if its inputs are restricted to non-negative numbers, because 0 is not the result of adding 1 to any non-negative number, so it is not in the set of actual outputs.

3. A function is a "*one-to-one correspondence*" (or "bijective") $=_{def}$ it is *both* one-to-one *and* onto. For example, the onto function $h(n)$ above is also one-to-one.

For more formal definitions and more examples, see
http://www.cse.buffalo.edu/~rapaport/191/F10/lecturenotes-20101103.html

7.6.2 Recursive Functions

Now let's look at one of the classic analyses of computable functions: a recursive definition of non-negative integer functions that are intuitively computable—that is, functions whose inputs are non-negative integers, also known as "natural numbers". But, first, what is a "natural number"?

7.6.2.1 A Recursive Definition of Natural Numbers

Informally, the set $\mathbb{N}$ of natural numbers = $\{0, 1, 2, \ldots\}$. They are the numbers defined (recursively!) by *Peano's axioms*.

P1 Base case: $0 \in \mathbb{N}$

That is, 0 is a natural number.

P2 Recursive case:

If $n \in \mathbb{N}$, then $S(n) \in \mathbb{N}$,
where S is a one-to-one function from $\mathbb{N}$ to $\mathbb{N}$ such that $(\forall n \in \mathbb{N})[S(n) \neq 0]$.

$S(n)$ is called "the *successor* of n". So, the recursive case says that every natural number has a successor that is also a natural number. The fact that S is a *function* means that each $n \in \mathbb{N}$ has only *one* successor. The fact that S is *one-to-one* means that no two natural numbers have the *same* successor. And the fact that 0 is not the successor of any natural number means both that S is not an "onto" function and that 0 is the "first" natural number.

P3 Closure clause: Nothing else is a natural number.

We now have a set of natural numbers:

$$\mathbb{N} = \{0,\ S(0),\ S(S(0)),\ S(S(S(0))),\ \ldots\}$$

and, as is usually done, we define $1 =_{def} S(0)$, $2 =_{def} S(S(0))$, and so on. The closure clause guarantees that there are no other natural numbers besides 0 and its successors: Suppose that there *were* an $m \in \mathbb{N}$ that was neither 0 nor a successor of 0, nor a successor of any of 0's successors; without the closure clause, such an m could be used to start a "second" natural-number sequence: $m,\ S(m),\ S(S(m)),\ \ldots$. So, the closure clause ensures that no proper *superset* of $\mathbb{N}$ is also a set of natural numbers. Thus, in a sense, $\mathbb{N}$ is "bounded from above". But we also want to "bound" it from *below*; that is, we want to say that $\mathbb{N}$ is the *smallest* set satisfying (P1)–(P3). We do that with one more axiom:

P4 Consider an allegedly proper (hence, smaller) subset **M** of $\mathbb{N}$. Suppose that:

1. $0 \in \mathbf{M}$

   and that

2. for all $n \in \mathbb{N}$, if $n \in \mathbf{M}$, then $S(n) \in \mathbf{M}$.

Then $\mathbf{M} = \mathbb{N}$.

Stated thus, (P4) is the axiom that underlies the logical rule of inference known as "mathematical induction":

> **From** the fact that 0 has a certain property **M**
> (that is, if 0 is a member of the class of things that have property **M**),
> **and**
> **from** the fact that, if any natural number that has the property **M** is such that its successor also has that property,
> **then** it may be inferred that *all* natural numbers have that property.

Further Reading:
Peano's axioms were originally proposed in (Peano, 1889) and (Dedekind, 1890). For more on what are also sometimes called the "Dedekind-Peano" axioms, see (Kennedy, 1968), (Joyce, 2005), and https://en.wikipedia.org/wiki/Giuseppe_Peano.

For further discussion of (P4), see
http://www.cse.buffalo.edu/~rapaport/191/F10/lecturenotes-20101110.html

7.6.2.2 Recursive Definitions of Recursive Functions

There are various kinds of recursive functions. To define them, we once again begin with "basic" functions that are intuitively, clearly computable, and then we recursively construct more complex functions from them. In this section, we will define these basic functions and the ways that they can be combined. In the next section, we will define the various kinds of recursive functions.

1. Basic functions:

   Let $x, x_1, \ldots, x_k \in \mathbb{N}$.

   (a) ***successor:*** $S(x) = x + 1$

   That is, $x+1$ is the successor of x. You should check to see that S satisfies Peano's axiom (P2).

   (b) ***predecessor:*** $P(x) = x \dot{-} 1$, where

$$a \dot{-} b =_{def} \begin{cases} a - b, & \text{if } a \geq b \\ 0, & \text{otherwise} \end{cases}$$

The odd-looking arithmetic operator is a "minus" sign with a dot over it, sometimes called "monus". So, the predecessor of x is $x-1$, except for $x=0$, which is its own predecessor.

(c) ***projection:***[23] $P_k^j(x_1,\ldots,x_j,\ldots,x_k) = x_j$

That is, P_k^j picks out the jth item from a sequence of k items.

The basic functions (a)–(c) can be seen to correspond in an intuitive way to the basic operations of a Turing machine: (a) The successor function corresponds to move(right), (b) the predecessor function corresponds to move(left) (where you cannot move any further to the left than the beginning of the Turing-machine tape), and (c) the projection function corresponds to reading the current square of the tape.[24]

Digression:
An alternative to predecessor as a basic function is the family of *constant* functions $C_q(x_1,\ldots,x_k) = q$ for each $q \in \mathbb{N}$ (Kleene 1952, p. 219; Soare 2009, §15.2, p. 397; Soare 2016, p. 229).

Both predecessor and monus can be defined recursively: Where $n,m \in \mathbb{N}$, let

$$P(0) = 0$$
$$P(S(n)) = n$$

and let

$$n \dot{-} 0 = n$$
$$n \dot{-} S(m) = P(n \dot{-} m)$$

For more details, see https://en.wikipedia.org/wiki/Monus#Natural_numbers.

And, while we're at it, we can define addition recursively, too:

$$n + 0 = n$$
$$n + S(m) = S(n+m)$$

[23] Sometimes called 'identity' (Kleene 1952, p. 220; Soare 2012, p. 3280; Soare 2016, p. 229).

[24] We'll return to this analogy in Ch. 8, §8.11.2. An analogous comparison in the context of "register machines" is made in (Shepherdson and Sturgis, 1963, p. 220).

2. Function constructors:

   Let $g, h, h_1, \ldots, h_m$ be (basic or complex) recursive functions.
   Then the following are also (complex) recursive functions:

   (a) *f is defined from $g, h_1, \ldots, h_m$ by* ***generalized composition*** $=_{def}$
   $f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$

   This can be made a bit easier to read by using the symbol $\overline{x}$ for the sequence $x_1, \ldots, x_k$. If we do this, then generalized composition can be written as follows:

   $$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_m(\overline{x})),$$

   which can be further simplified to:

   $$f(\overline{x}) = g(\overline{h(\overline{x})})$$

   Note that $g(h(x))$—called "function composition"—is sometimes written '$g \circ h$'. So, roughly, if g and h are recursive functions, then so is their (generalized) composition $g \circ h$.

   This is analogous to structured programming's notion of linear concatenation (that is, sequencing): First compute h; then compute g.

   (b) *f is defined from g, h, i by* ***conditional definition*** $=_{def}$

   $$f(x_1, \ldots, x_k) = \begin{cases} g(x_1, \ldots, x_k), & \text{if } x_i = 0 \\ h(x_1, \ldots, x_k), & \text{if } x_i > 0 \end{cases}$$

   Using our simplified notation, we can write this as:

   $$f(\overline{x}) = \begin{cases} g(\overline{x}), & \text{if } x_i = 0 \\ h(\overline{x}), & \text{if } x_i > 0 \end{cases}$$

   This is analogous to structured programming's notion of conditional branch (that is, selection): If a Boolean test (in this case, "$x_i = 0$") is true, then compute g, else compute h.[25]

   (c) *f is defined from g, h by* ***primitive recursion*** $=_{def}$

   $$f(x_1, \ldots, x_k, y) = \begin{cases} g(x_1, \ldots, x_k), & \text{if } y = 0 \\ h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, y-1)), & \text{if } y > 0 \end{cases}$$

   Using our simplified notation, this becomes:

   $$f(\overline{x}, y) = \begin{cases} g(\overline{x}), & \text{if } y = 0 \\ h(\overline{x}, f(\overline{x}, y-1), & \text{if } y > 0 \end{cases}$$

[25] Note, by the way, that "$x_i = 0$" can be written: $P^i_k(x_1, \ldots, x_k) = 0$

Note, first, that the "$y = 0$" case is the base case, and the "$y > 0$ case is the recursive case. Second, note that this combines conditional definition with a computation of f based on f's value for its *previous* output. This is the essence of recursive definitions of functions: Instead of computing the function's output based on its *current input*, the output is computed on the basis of the function's *previous output*.

Further Reading:
For a useful discussion of this, see (Allen, 2001).

This is analogous to structured programming's notion of a count-loop: while $y > 0$, decrement y and then compute f.

(d) *f is defined from $g, h_1, \ldots, h_k, i$ by* ***while-recursion*** $=_{def}$

$$f(x_1,\ldots,x_k) = \begin{cases} g(x_1,\ldots,x_k), & \text{if } x_i = 0 \\ f(h_1(x_1,\ldots,x_k),\ldots,h_k(x_1,\ldots,x_k)), & \text{if } x_i > 0 \end{cases}$$

Again, using our simplified notation, this can be written as:

$$f(\overline{x}) = \begin{cases} g(\overline{x}), & \text{if } x_i = 0 \\ f(h(\overline{x})), & \text{if } x_i > 0 \end{cases}$$

This is analogous to structured programming's notion of while-loop (that is, repetition): While a Boolean test (in this case, "$x_i > 0$") is true, compute h, and loop back to continue computing f, but, when the test becomes false, then compute g.

An Example of a Function Defined by While-Recursion:

The Fibonacci sequence is:

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

where each term after the first two terms is computed as the sum of the previous two terms. This can be stated recursively:

- The first two terms of the sequence are 0 and 1.
- Each subsequent term in the sequence is the sum of the previous two terms.

This can be defined using while-recursion as follows:

$$f(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x = 1 \\ f(x-1) + f(x-2), & \text{if } x > 1 \end{cases}$$

We can make this look a bit more like the official definition of while-recursion by taking $h_1(x) = P(x) = x \dot{-} 1$ and $h_2(x) = P(P(x)) = P(x \dot{-} 1) = (x \dot{-} 1) \dot{-} 1 = x \dot{-} 2$. In other words, the two base cases of f are projection functions, and the recursive case uses the predecessor function twice (the second time, it is the predecessor of the predecessor).

(e) *f is defined from h by the* ***μ-operator*** [pronounced: "mu"-operator] $=_{def}$

$$f(x_1,\ldots,x_k) = \mu z[h(x_1,\ldots,x_k,z) = 0]$$

where:

$\mu z[h(x_1,\ldots,x_k,z) = 0] =_{def}$

$$\begin{cases} min\{z : \begin{cases} h(x_1,\ldots,x_k,z) = 0 \\ \textbf{and} \\ (\forall y < z)[h(x_1,\ldots,x_k,y) \text{ has a non-0 value}] \end{cases} \}, & \textbf{if such } z \text{ exists} \\ \text{undefined}, & \textbf{if no such } z \text{ exists} \end{cases}$$

This is a complicated notion, but one well worth getting an intuitive understanding of. It may help to know that it is sometimes called "unbounded search" (Soare, 2012, p. 3284).

Let me first introduce a useful notation. If $f(x)$ has a value—that is, if it is defined (in other words, if an algorithm that computes f halts)—then we will write: $f(x)\downarrow$. And, if $f(x)$ is undefined—that is, if it is only a "partial" function (in other words, if an algorithm for computing f goes into an infinite loop)—then we will write: $f(x)\uparrow$. Now, using our simplified notation, consider the sequence

$$h(\bar{x},0),\ h(\bar{x},1),\ h(\bar{x},2),\ \ldots,\ h(\bar{x},n),\ h(\bar{x},z),\ \ldots,\ h(\bar{x},z')$$

Suppose that each of the first $n+1$ terms of this sequence halts with a non-zero value, but thereafter each term halts with value 0; that is:

$$\begin{gathered} h(\bar{x},0)\downarrow\ \neq\ 0 \\ h(\bar{x},1)\downarrow\ \neq\ 0 \\ h(\bar{x},2)\downarrow\ \neq\ 0 \\ \ldots \\ h(\bar{x},n)\downarrow\ \neq\ 0 \\ \text{but:} \\ h(\bar{x},z)\downarrow\ =\ 0 \\ \ldots \\ h(\bar{x},z')\downarrow = 0 \end{gathered}$$

The μ-operator gives us a description of that smallest or "min"imal z (that is, the first z in the sequence) for which h halts with value 0. So the definition of μ says, roughly:

> $\mu z[h(\bar{x},z) = 0]$ is the smallest z for which $h(\bar{x},y)$ has a non-0 value for each $y < z$, **but** for which $h(\bar{x},z) = 0$, **if** such a z exists;
> **otherwise** (that is, if no such z exists), $\mu z[h(\bar{x},z)]$ is undefined.

So, f is defined from h by the μ-operator if you can compute $f(\bar{x})$ by computing the smallest z for which $h(\bar{x},z) = 0$.

If h is intuitively computable, then, to compute z, we just have to compute $h(\bar{x},y)$, for each successive natural number y, until we find z. So definition by μ-operator is also intuitively computable.

7.6.2.3 Classification of Recursive Functions

Given these definitions, we can now classify computable functions:

1. *f is a **while-recursive function*** $=_{def}$
   (a) f is a basic function, or
   (b) f is defined from while-recursive functions by:
      i. generalized composition, or
      ii. conditional definition, or
      iii. while-recursion
   (c) Nothing else is while-recursive.

   This is the essence of the Böhm-Jacopini Theorem: Any computer program (that is, any algorithm for any computable function) can be written using only the three rules of grammar: sequence (generalized composition), selection (conditional definition), and repetition (while-recursion).

2. *f is a **primitive-recursive** function* $=_{def}$
   (a) f is a basic function, or
   (b) f is defined from primitive-recursive functions by:
      i. generalized composition, or
      ii. primitive recursion
   (c) Nothing else is primitive-recursive.

   The *primitive-recursive* functions and the *while-recursive* functions overlap: Both include the basic functions and functions defined by generalized composition (sequencing).

   The *primitive-recursive* functions also include the functions defined by primitive recursion (a combination of selection and count-loops), but nothing else.

   The *while-recursive* functions include (along with the basic functions and generalized composition) functions defined by conditional definition (selection) and those defined by while-recursion (while-loops).

3. *f is a* ***partial-recursive*** *function* $=_{def}$

   (a) f is a basic function, or

   (b) f is defined from partial-recursive functions by:

      i. generalized composition, or

      ii. primitive recursion, or

      iii. the μ-operator

   (c) Nothing else is partial-recursive.

4. *f is a* ***recursive*** *function* $=_{def}$

   (a) f is partial-recursive, and

   (b) f is a total function
   (that is, defined for all elements of its domain)

Further Reading:
Unfortunately, the terminology varies with the author. For example, primitive recursive functions were initially called just "recursive functions"; now, it is the while-recursive functions that are usually just called "recursive functions", or sometimes "*general* recursive functions" (to distinguish them from the *primitive* recursive functions); and partial recursive functions are sometimes called "μ-recursive functions" (because they are the primitive recursive functions augmented by the μ-operator). For the history of this and some clarification, see Soare (2009, §§2.3–2.4, p. 373–373; and §15.2, pp. 396–297) and http://mathworld.wolfram.com/RecursiveFunction.html

How are all of these notions related? First, here are the relationships among the various kinds of recursive functions: As we saw, there is an overlap between the *primitive-recursive* functions and the *while-recursive* functions, with the basic functions and the functions defined by generalized composition in their intersection.

The *partial-recursive* functions are a superset of the *primitive-recursive* functions. The partial-recursive functions consist of the primitive-recursive functions together with the functions defined with the μ-operator.

The *recursive* functions are a subset of the *partial-recursive* functions: The recursive functions are the partial-recursive functions that are also total functions.

Second, here is how the *recursive* functions and the *computable* functions are related:

> f is *primitive-recursive* if and only if f is *count-program-computable*.
>
> f is *partial-recursive* iff f is *while-program-computable*.
>
> *And both of these* (partial-recursive and while-program-computable) *are logically equivalent to being Turing-machine computable, lambda-definable, Markov-algorithmic, etc.*

7.7 The Halting Problem

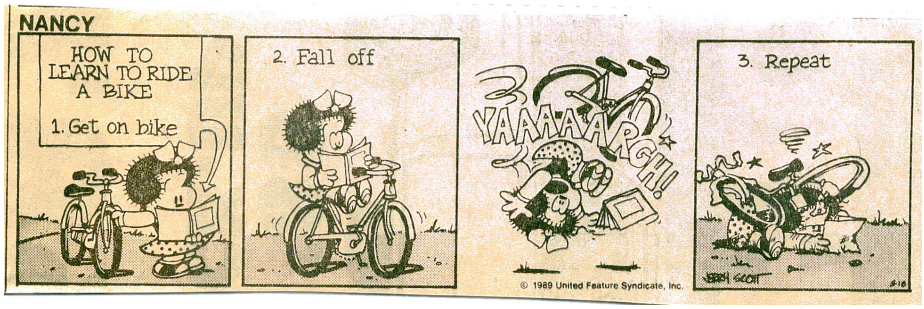


Figure 7.11: ©1989 United Feature Syndicate

7.7.1 Introduction

> You can build an organ which can do anything that can be done, but you cannot build an organ which tells you whether it can be done.
> —(von Neumann, 1966), cited in (Dyson, 2012a).

Have we left anything out? That is, are there any other functions besides these? Yes! The "Halting Problem" provides an example of a *non-computable* function, that is, a function that cannot be defined using any of the mechanisms of §7.6.2. Recall that a function *is* computable if and only if there is an algorithm (that is, a computer program) that computes it. So, the Halting Problem asks whether there is an *algorithm* (for example, a program for a Turing machine)—call it the "Halting Algorithm", A_H—that computes the following *function* $H(C,i)$ (call it the "Halting Function"):

$H(C,i)$ takes as input *both*:

1. an algorithm (or computer program) C
   (which we can suppose takes an integer as input),

   and

2. C's input i
   (which would be an integer)

and $H(C,i)$ outputs:

- "halts", if C halts on i
- "loops", if C loops on i.

A *formula* for H is:

$$H(C,i) = \begin{cases} \text{"halts"}, & \text{if } C(i)\downarrow \\ \text{"loops"}, & \text{if } C(i)\uparrow \end{cases}$$

And our question is: Is there an *algorithm* A_H that computes H? Can we write such a program?

In terms of the "function machine" illustration from §7.3.1.3, we are asking whether there is a "function machine" (that is, a computer) whose internal mechanism (that is, whose program) is A_H. When you input the pair $\langle C, i \rangle$ to this "function machine" and turn its "crank", it should output "halts" if another function machine (namely, the function machine for C) successfully outputs a value when you give it input i, and it should output "loops" if the function machine for C goes into an infinite loop and never outputs any final answer.[26]

Here's another way to think about this: A_H is a kind of "super"-machine that takes as input, not only an integer i, but also another *machine* C. When you turn A_H's "crank", A_H first feeds i to C, and then A_H turns C's "crank". If A_H detects that C has successfully output a value, then A_H outputs "halts"; otherwise, A_H outputs "loops".

This would be very useful for introductory computer-programming teachers or software engineers in general! After all, one thing you never want in a computer program is an unintentional infinite loop. Sometimes, you might want an *intentional* one, however: You *don't* want an automated teller machine to halt—you *do* want it to behave in an infinite loop so that it is always ready to accept new input from a new customer. It would, however, be very useful to have a *single*, handy program that could quickly check to see if *any* program that someone writes has an infinite loop in it. But no such program can be written! In a moment, we will see why.

Before doing so, however, it is important to be clear that it *can* be possible to write a program that will check if another program has an infinite loop. In other words, given a program C_1, there might be another program H_1 that will check whether C_1—*but not necessarily any other program*—has an infinite loop. What *cannot* be done is this: To write a single program A_H that will take *any program C whatsoever* and tell you whether C will halt or not.

Logical Digression and Further Reading:
To be more precise, the difference between these two situations has to do with the order of the logical quantifiers: The Halting Problem asks this question: Does *there exist* a *single* program A_H, such that, *for any* program C, A_H outputs "halts" if C halts, else it outputs "loops"? The answer to this question is "no"; no such program A_H exists. The other question, for which there *can* be a positive answer, is this: *Given any* program C, does *there exist* a program H_C (which will depend on what C is!) that outputs "halts" if C halts, else it outputs "loops"? Note that different Cs might have different H_Cs. The answer to *this* question can be "yes", at least for some Cs:

> In contrast to popular belief, proving termination is not always impossible. ... [M]any have drawn too stong of a conclusion [from Turing's proof that Halting is non-computable] ... and falsely believe we are *always unable to* prove termination, rather than more benign consequence that we are *unable to always* prove termination. ... In our new problem statement we will still require that a termination proving tool always return answers that are correct, but we will not necessarily require an answer. —(Cook et al., 2011, my italics)

[26] It may, however, output some messages, but it never halts with an answer to $C(i)$.

Note that we can't answer the question whether C halts on i by just running C on i: If it halts, we know that it halts. But if it loops, how would we know that it loops? After all, it might just be taking a long time to halt.

There are two ways that we might try to write A_H.

1. You can imagine that $A_H(C,i)$ works as follows:

   - A_H gives C its input i, and then runs C on i.

     (If A_H is a "function machine", then its mechanism includes a miniature version of C's function machine: You input both *i and C's function machine* to A_H and turn the crank; A_H then inputs i to C's function machine and turns *its* crank.)

   - If C halts on i, then A_H outputs "halts";
     otherwise, A_H outputs "loops".

   So, we might write A_H as follows:

   algorithm $A_H^1(C,i)$:
   begin
     if $C(i)\downarrow$
       then output 'halts'
       else output 'loops'
   end.

   This matches our *formula* for function H.

2. But here's another way to write A_H:

   algorithm $A_H^2(C,i)$:
   begin
     output 'loops'; {that is, make an initial *guess* that C loops}
     if $C(i)\downarrow$
       then output 'halts'; {that is, *revise* your guess}
   end.

   "Trial-and-error" programs like A_H^2 will prove useful in our later discussion (in Chapter 11) of hypercomputation (that is, computation that, allegedly, cannot be modeled by Turing machines). But it won't work here, because we're going to need to convert our program for H to another program called A_H^*, and A_H^2 can't be converted that way, as we'll see. More importantly, A_H^2 doesn't really do the required job: It doesn't give us a *definitive* answer to the question of whether C halts, because its initial answer is not really "loops", but something like "not yet".

The answer to our question about whether such an algorithm A_H exists or can be written is negative: There is no program for $H(C,i)$. In other words, $H(C,i)$ is a non-computable function. Note that it *is* a function: There exists a perfectly good set of

input-output pairs that satisfies the extensional definition of 'function' and that looks like this:

$$\{\langle C_1, i_1\rangle, \text{“halts”}\rangle\ ,\ \ldots\ , \langle C_j, i_k\rangle, \text{“loops”}\rangle\ ,\ \ldots\}$$

The next section sketches a proof that H is not computable. The proof takes the form of a "*reductio ad absurdum*" argument.

Logical Digression:
A "*reductio ad absurdum*" argument is one that "reduces" a claim to "absurdity" in order to refute the claim. If you want to show that a claim P is *false*, the strategy is to *assume*—"for the sake of the argument"—that P is *true*, and then to derive a contradiction C (that is, an "absurdity") from it. If you can thus show that $P \rightarrow C$, then—because you know that $\neg C$ is the case (after all, C is a contradiction, hence false; so $\neg C$ must be true)—you can conclude that $\neg P$, thus refuting P.

The rule of inference that sanctions this is "modus tollens":

> From $A \rightarrow B$
> and $\neg B$,
> you may infer $\neg A$

For more on this, see any introductory logic text, or the citations in §4.9.1.2.

So, for our proof, we will assume that H *is* computable, and derive a contradiction. If an assumption implies a contradiction, then—because no contradiction can be true—the assumption must have been wrong. So, our assumption that H is computable will be shown to be false.

7.7.2 Proof Sketch that H Is Not Computable

7.7.2.1 Step 1

Assume that function H *is* computable.

So, there is an *algorithm* A_H that computes *function* H.

Now consider another algorithm, A_H^*, that works as follows: A_H^* is just like algorithm A_H, except that:

- if C *halts* on i, then A_H^* *loops*

  (Remember: If C halts on i, then, by C's definition, A_H does *not* loop, because A_H outputs "halts" and then halts.)

  and

- if C *loops* on i, then A_H^* outputs "loops" and halts (just like A_H does).

Here is how we might write A_H^*, corresponding to the version of A_H that we called 'A_H^1' above:

algorithm $A_H^{1*}(C, i)$:
begin
  if $C(i) \downarrow$
    then while true **do begin end**
    else output 'loops'
end.

Here, 'true' is a Boolean test that is always true. (As we noted earlier, you could replace it by something like '1=1', which is also always true.)

Note that we cannot write a version of A_H^2 that might look like this:

algorithm $A_H^{2*}(C, i)$:
begin
  output 'loops'; {that is, make an initial guess that C loops}
  if $C(i) \downarrow$
    then while true **do begin end** {that is, if C halts, then loop}
end.

Why not? Because if *C halts*, then the only output we will ever see is the message that says that *C loops*! That initial, incorrect guess is never revised. So, we'll stick with A_H (that is, with A_H^1) and with A_H^* (that is, with A_H^{1*}).

Note that if A_H exists, so does A_H^*. That is, we can turn A_H into A_H^* as follows: If A_H were to output "halts", then let A_H^* go into an infinite loop. That is, replace A_H's "output 'halts' " by A_H^*'s infinite loop. This is important, because we are going to show that, in fact, A_H^* does *not* exist; hence, neither does A_H.

7.7.2.2 Step 2

Returning to our proof sketch, the next step is to code C as a number, so that it can be treated as input to itself.

What? Why do that? Because this is the way to simulate the idea of putting the C "machine" into the A_H machine and then having the A_H machine "turn" C's "crank".

So, how do you "code" a program as a number? This is an insight due to Kurt Gödel. To code *any* text (including a computer program) as a number in such a way that you could also *de*code it, begin by coding each symbol in the text as a unique number (for example, using the ASCII code). Suppose that these numbers, in order, are $L_1, L_2, L_3, \ldots, L_n$, where L_1 codes the first symbol in the text, L_2 codes the second, ..., and L_n codes the last symbol.

Then compute the following number:

$$2^{L_1} \times 3^{L_2} \times 5^{L_3} \times 7^{L_4} \times \ldots \times p_n^{L_n}$$

where p_n is the nth prime number, and where the ith factor in this product is the ith prime number raised to the L_ith power.

By the "Fundamental Theorem of Arithmetic",[27] the number that is the value of this product can be uniquely factored, so those exponents can be recovered, and then they can be decoded to yield the original text.

Further Reading:
Gödel numbering is actually a bit more complicated than this. For more information, see "Gödel Number", http://mathworld.wolfram.com/GoedelNumber.html
or "Gödel Numbering", http://en.wikipedia.org/wiki/Godel_numbering

Turing has an even simpler way to code symbols; we'll discuss his version in detail in the next chapter (§8.13).

For a comparison of the two methods, see (Kleene, 1987, p. 492).

7.7.2.3 Step 3

Now consider $A_H^*(C,C)$. This step is called "diagonalization". It looks like a form of self-reference, because it looks as if we are letting C take itself as input to itself—but actually C will take its own Gödel number as input. That is, suppose that you (1) code up program C as a Gödel number, (2) use it as input to the program C itself (after all, the Gödel number of C is an integer, and thus it is in the domain of the function that C computes, so it is a legal input for C), and (3) then let A_H^* do its job on *that* pair of inputs.

By the definition of A_H^*:

> if program C *halts* on input C, then $A_H^*(C,C)$ *loops*;
>
> and
>
> if program C *loops* on input C, then $A_H^*(C,C)$ *halts* and outputs "loops".

7.7.2.4 Step 4

Now code A_H^* by a Gödel number! And consider $A_H^*(A_H^*, A_H^*)$. This is another instance of diagonalization. Again, it may look like some kind of self-reference, but it really isn't, because the first occurrence of 'A_H^*' names an algorithm, but the second and third occurrences are just numbers that happen to be the code for that algorithm.[28]

In other words, (1) code up A_H^* by a Gödel number, (2) use it as input to the program A_H^* itself, and then (3) let A_H^* do its job on that pair of inputs.

Again, by the definition of A_H^*:

> if program A_H^* *halts* on input A_H^*, then $A_H^*(A_H^*, A_H^*)$ *loops*;
>
> and
>
> if program A_H^* *loops* on input A_H^*, then $A_H^*(A_H^*, A_H^*)$ *halts* and outputs "loops".

[27] http://mathworld.wolfram.com/FundamentalTheoremofArithmetic.html

[28] My notation here, cumbersome as it is(!), is nonetheless rather informal, but—I hope—clearer than it would be if I tried to be even more formally precise.

7.7.2.5 Final Result

But A_H^* outputting "loops" means that A_H^* halts!

So, if A_H^* *halts* (outputting "loops"), then it *loops*, and, if A_H^* *loops*, then it *halts*. In other words, *it loops if and only if it halts*; that is, it *does* loop if and only if it does *not* loop!

But that's a contradiction!

So, there is no such program as A_H^*. But that means that there is no such program as A_H. In other words, *the Halting Function H is not computable.*

Further Reading:
On the history of the Halting Problem, see the Further Reading box in §8.10.3.3.

(Chaitin, 2006a) (and (Chaitin, 2006b), which is aimed at a more general audience) discusses the Halting Problem, the non-existence of real numbers(!), and the idea that "everything is software, God is a computer programmer, ... and the world is ... a giant computer"(!!). On the non-"reality" of "real" numbers, see also (Knuth, 2001, pp. 174–175). On the relationship of random numbers to the Halting Problem, see footnote 29, below.

For a humorous take on the Halting Problem, see (Rajagopalan, 2011).

(Vardi, 2011b) argues that the Halting Problem is not the absolute limitation that it appears to be.

7.7.3 Other Non-Computable Functions

> [A] function is a set of ordered pairs ... [satisfying the unique output condition]. ... A computable function ... is a mapping [that is, a function] that can be specifed in terms of some *rule* or other, and is generally characterized in terms of what you have to do to the first element to get the second [where the rule must satisfy the constraints of being an algorithm]. ... [A] noncomputable function ... is an infinite set of ordered pairs for which no rule can be provided, not only now, but in principle. Hence **its specification consists simply and exactly in the list of ordered pairs.**
> —(Churchland and Sejnowski, 1992, p. 62, italics in original, my boldface)[29]

The Halting Function is not the only non-computable function. There are many more; in fact, there are infinitely many of them. Moreover, there are more non-computable functions than computable ones.

Digression:
There are also infinitely many *computable* functions, but "only" *countably* infinitely many, whereas there are *uncountably* infinitely many *non*-computable functions. For more on this, see https://simple.wikipedia.org/wiki/Countable_set, https://en.wikipedia.org/wiki/Countable_set, and https://cs.stackexchange.com/questions/9633/why-are-there-more-non-computable-functions-than-computable-ones

[29] Note that this "specification" is almost the same as the definition of "random number" given in (Chaitin, 2006a). See the Further Reading box at the end of §7.7.2.5.

7.7.3.1 Hilbert's 10th Problem

Two other famous non-computable functions are Hilbert's 10th Problem and the Busy Beaver function.

The first of these was the 10th problem in a famous list of math problems that Hilbert presented in his 1900 speech as goals for 20th century mathematicians to solve (Hilbert, 1900). It concerns Diophantine equations, that is, equations of the form $p(x_1,\ldots,x_n) = 0$, where p is a polynomial with integer coefficients. Hilbert's 10th Problem says:

> Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: *To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.* —(Hilbert, 1900, p. 458)

In the early 1970s, Yuri Matiyasevich, along with the mathematicians Martin Davis and Julia Robinson, and the philosopher Hilary Putnam, proved that there was no such "process" (that is, no such algorithm).

Further Reading:
For more information, see: http://en.wikipedia.org/wiki/Hilbert's_tenth_problem and "Hilbert's Tenth Problem page" at: http://logic.pdmi.ras.ru/Hilbert10/

7.7.3.2 The Busy Beaver Problem

The Busy Beaver function has been described as follows:

> $\Sigma(n)$ is defined to be the largest number which can be computed by an n-state Turing machine .... —(Chaitin, 1987, p. 2)

A slightly more intuitive description is this:

> Given an n-state Turing machine with two-symbol alphabet $\{0,1\}$, what is the maximum number of 1s the machine may print on an initially blank (0-filled) tape before halting? ... The busy beaver problem cannot be solved in general by a computer since the function $\Sigma(n)$ grows faster than any computable function $f(n)$. —(Dewdney, 1989, pp. 241–242)

It was first described, and proved to be non-computable, by Tibor Radó (Radó, 1962).

Further Reading:
(Radó, 1962) is a wonderfully readable introduction to the Busy Beaver "game". §II gives a very simple example, aimed at students, of a Turing machine, and §§I–III and, especially, §VIII are amusing and well worth reading!

For more information, see (Suber, 1997d) and http://en.wikipedia.org/wiki/Busy_beaver

7.8 Summary

Let's take stock of where we are. We asked whether CS is the science of *computing* (rather than the science of *computers*). In order to answer that, we asked what computing, or computation, is. We have now seen one answer to that question: Computation is the process of executing an algorithm to determine the output value of a function, given an input value. We have seen how to make this informal notion precise, and we have also seen that it is an interesting notion in the sense that not all functions are computable.

But this was a temporary interruption of our study of the history of computers and computing. In the next chapter, we will return to that history by examining Alan Turing's formulation of computability.

7.9 Questions for the Reader

1. To the lists of features of algorithms in §7.4, above, (Gurevich, 2012, p. 4) adds "isolation":

   > Computation is self-contained. No oracle is consulted, and nobody interferes with the computation either during a computation step or in between steps. The whole computation of the algorithm is determined by the initial state.

   Is this related to Markov's "being determined" feature, or Kleene's "followed mechanically" feature, or Knuth's "definiteness" feature?

   Does "isolation" mean that a program that asks for input from the external world (or from a user, who, of course, is in the external world!) is not doing computation? (We'll discuss this in Chapters 11 and 17, but you should start thinking about this now.)

2. Gurevich has another "constraint": "Computation is symbolic (or digital, symbol-pushing)" (p. 4). That is, computation is syntactic. (See Chapter 22 for a discussion of what that means.)

   Does that mean that computation is not mathematical (because mathematics is about *numbers*, not numerals)? Does it mean that computers cannot have real-world effects? (We'll return to these topics in Chapter 17.)

3. Harry Collins described an "experimenter's regress":

   > [Y]ou can say an experiment has truly been replicated only if the replication gets the same result as the original, a conclusion which makes replication pointless. Avoiding this, and agreeing that a replication counts as "the same procedure" even when it gets a different result, requires recognising the role of tacit knowledge and judgment in experiments. —(*The Economist*, 2013)

   If you consider an experiment as a mathematical binary relation whose input is, let's say, the experimental set-up and whose output is the result of the experiment, then, if a replication of the experiment always gets the same result, then the relation is a function.

   Can scientific experiments be considered as (mathematical) functions? In that case, does it make any sense to replicate an experiment in order to confirm it?

4. Should other verbs be added to the Processing Insight? Is "read" a verb on a par with the ones cited? (Is "read" even needed?) Should Boolean tests be included as verbs?

5. > Computability is a relative notion, not an absolute one. All computation, classical or otherwise, takes place relative to some set or other or primitive capabilities. The primitives specified by Turing in 1936 occupy no privileged position. One may ask whether a function is computable relative to these primitives or to some superset of them. —(Copeland, 1997, p. 707); see also (Copeland and Sylvan, 1999, pp. 46–47).

   In §7.6.1.1, definition (1b), I said that primitive operations had to be computable, at least in an informal sense. After all, there we were trying to define what it meant to be computable. But another way to proceed would be to say that primitive operations are computable *by definition*.

   But does this allow *anything* to be a primitive operation, even something that really shouldn't be (informally) computable? What if the primitive operation is, in fact, *non*-computable? Could we have a kind of "computation" in which the recursive portions are based on a non-computable (set of) primitive operation(s)?

   For more on relative computability, see (Soare, 2009, 2016); (Homer and Selman, 2011, Ch. 7); and Chapter 11, below.

6. A research question:

   > ... every physical process instantiates a *computation* insofar as it progresses from state to state according to dynamics prescribed by the laws of physics, that is, by systems of differential equations. —(Fekete and Edelman, 2011, p. 808)

   This suggests the following very odd and very liberal definition: Something is a *computation* $=_{def}$ it is a progression from state to state that obeys a differential equation. This definition is liberal, because it seems to go beyond the limitations of a Turing-machine-like algorithm. That's not necessarily bad; for one thing, it subsumes both analog and discrete computations under one rubric.

   Are Turing-machine algorithms describable by differential equations?

7.10 Further Sources of Information

Items in **boldface** are especially interesting.

1. Books and Essays:

   (a) Harnad, Stevan (ed.) (1994), Special Issue on "What Is Computation?", *Minds and Machines* 4(4), containing:

   - Harnad, Stevan, "Preface", pp. 377–378.
   - **Harnad, Stevan, "Computation Is Just Interpretable Symbol Manipulation; Cognition Isn't", pp. 379–390**, http://users.ecs.soton.ac.uk/harnad/Papers/Harnad/harnad94.computation.cognition.html
   - **Chalmers, David J., "On Implementing a Computation", pp. 391–402.**
   - Chrisley, Ronald L., "Why Everything Doesn't Realize Every Computation", pp. 403–420.
   - MacLennan, Bruce J., "Words Lie in Our Way", pp. 421–437.
   - Kentridge, Robert W., "Symbols, Neurons, Soap-Bubbles, and the Neural Computation Underlying Cognition", pp. 439–449.
   - Boyle, C. Franklin, "Computation as an Intrinsic Property", pp. 451–467.
   - **Bringsjord, Selmer, "Computation, among Other Things, Is beneath Us", pp. 489–490**; portions online at: http://homepages.rpi.edu/~brings/SELPAP/beneath.ab.html

   (b) Shagrir, Oron (1999), "What Is Computer Science About?", *The Monist* 82(1): 131–149.

   - Despite its title, this paper is more about what computers are and what computation is.

   (c) **Smith, Brian Cantwell (2002), "The Foundations of Computing", in Matthias Scheutz (ed.), *Computationalism: New Directions* (Cambridge, MA: MIT Press): 23–58.**

   (d) Vardi, Moshe Y. (2011), "Solving the Unsolvable", *Communications of the ACM* 54(7) (July): 5, http://www.inf.unibz.it/~calvanese/teaching/tc/material/solving-the-unsolvable-CACM-2011-07.pdf

   - See also: Ledgard, Henry; & Vardi, Moshe Y. (2011), "Solved, for All Practical Purposes" (letter to the editor, and reply), *Communications of the ACM* 54(9) (September): 7, http://tinyurl.com/LedgardVardi2011

2. Website:

   - PolR (2009), "An Explanation of Computation Theory for Lawyers", *Groklaw* (11 November), http://www.groklaw.net/article.php?story=20091111151305785, http://www.groklaw.net/staticpages/index.php?page=20091110152507492, and http://www.groklaw.net/pdf2/ComputationalTheoryforLawyers.pdf

Chapter 8

Turing's Analysis of Computation

Version of 28 June 2019; DRAFT © 2004–2019 by William J. Rapaport

> Turing's 'Machines'. These machines are *humans* who calculate.
> —(Wittgenstein, 1980, p. 191e, §1096)

> [A] human calculator, provided with pencil and paper and explicit instructions, can be regarded as a kind of Turing machine. —(Church, 1937)

> [Wittgenstein's] quotation, though insightful, is somewhat confusingly put. Better would have been: these machines are Turing's mechanical model of *humans* who calculate. —(Kripke, 2013, p. 96, footnote 12)

8.1 Required Reading:

- **Turing, Alan M. (1936), "On Computable Numbers, with an Application to the *Entscheidungsproblem*", *Proceedings of the London Mathematical Society*, Ser. 2, Vol. 42 (1937): 230–265, http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf**

  - Reprinted, with corrections, in Martin Davis (ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions* (New York: Raven Press, 1965): 116–154.

    1. Concentrate on the informal expository parts; the technical parts are, of course, of interest, but are rather difficult to follow, incorrect in many parts, and can be skimmed.
    2. In particular, concentrate on:
       (a) §§1–6
       (study the simple examples of Turing machines carefully;
       skim the complex ones)
       (b) and §9, part I
       (which elaborates on what it is that a *human* computer does).
    3. §7 describes the universal Turing machine;
       §8 describes the Halting Problem.
       * You can skim these sections (that's 'ski**m**', not 'ski**p**'!)

8.2 Introduction

If there is a single document that could be called the foundational document of CS, it would be Alan Mathison Turing's 1936 article, "On Computable Numbers, with an Application to the *Entscheidungsproblem*", which appeared in the journal *Proceedings of the London Mathematical Society, Series 2*. In this paper, Turing (who was only about 24 years old at the time) accomplished (at least) 5 major goals:

1. He gave what is considered to be the clearest and most convincing mathematical analysis of computation (what is now called, in his honor, a "Turing machine").
2. He proved that there were some functions that were *not* computable, thus showing that computation was not a trivial property. (After all, if all functions were computable—which no doubt would be a very nice feature—then computability would not really be a very interesting or special property. But, because some functions are *not* computable, computability is a property that only some (but not all) functions have, and so it becomes more interesting.)
3. He proved that the Turing-machine analysis of computation was logically equivalent to Church's lambda-calculus analysis of computation.
4. He formulated a "universal" Turing machine, which is a mathematical version of a programmable computer.
5. And (as I suggested in §3.14.6) he wrote the first AI program (see §8.8.2.8.3, below).

Thus, arguably, in this paper, he created the modern discipline of CS.

Because this paper was so important and so influential, it is well worth reading. Fortunately, although parts of it are tough going (and it contains some errors),[1] much of it is very clearly written. It is not so much that the "tough" parts are difficult or hard to understand, but they are full of nitty, gritty details that have to be slogged through. Fortunately, Turing has a subtle sense of humor, too.

In this chapter, I will provide a guide to reading *parts* of Turing's paper slowly and carefully, by actively thinking about it.

Further Reading:
A wonderful guide to reading *all* of it slowly and carefully is (Petzold, 2008). (For a review of Petzold's book, see (Davis, 2008).) Petzold's book has several accompanying websites: http://www.theannotatedturing.com/, http://www.charlespetzold.com/AnnotatedTuring/, and http://www.amazon.com/gp/reader/0470229055/ref=sib_dp_pt#reader-link

Bernhardt 2016, while not focusing on Turing's paper itself, is an excellent guide for the general reader to the mathematics of computation theory.

[1] Some of which Turing himself corrected (Turing, 1938). For a more complete "debugging", see (Davies, 1999).

8.3 Slow and Active Reading

One of the best ways to read is to read slowly and actively. This is especially true when you are reading a technical paper, and even more especially when you are reading mathematics.

Reading slowly and actively means (1) reading each sentence slowly, (2) thinking about it actively, and (3) making sure that you understand it before reading the next sentence.

One way to make sure that you understand it is to ask yourself *why* the author said it, or *why* it might be true. (Recall our discussion in §2.5.1, of the importance of asking "why".) If you don't understand it (after reading it slowly and actively), then you should re-read all of the previous sentences to make sure that you really understood them. Whenever you come to a sentence that you really don't understand, you should ask someone to help you understand it.

(Of course, it could also be the case that you don't understand it because it isn't true, or doesn't follow from what has been said, or is confused in some way—and not because it's somehow your fault that you don't understand it!)

Further Reading:
On the value of slow and active reading in general, see my website (Rapaport, 2018a), as well as (Fletcher, 2007) and (Blessing, 2013).

8.4 Title: "The *Entscheidungsproblem*"

We'll begin our slow and active reading of Turing's paper with the title, in particular its last word: '*Entscheidungsproblem*'. This is a German noun that—as we saw in §6.6—was well known to mathematicians in the 1930s; '*Entscheidung*' means "decision", '*-s*' represents the possessive,[2] and '*problem*' means "problem". So, an *Entscheidungsproblem* is a decision problem, and *the* Decision Problem was the problem of finding an algorithm that would (a) take two things as input: (1) a formal logic L and (2) a proposition φ_L in the language for that logic, and that would (b) output either 'yes', if φ_L was a theorem of that logic, or else 'no', if $\neg\varphi_L$ was a theorem of that logic (that is, if φ_L was not a theorem of L). In other words, the Decision Problem was the problem of finding a general algorithm for deciding whether any given proposition was a theorem.

Wouldn't that be nice? Mathematics could be completely automated: Given any mathematical proposition, one could apply this general algorithm to it, and you would be able to know if it were a theorem or not. Turing was fascinated by this problem, and he solved it. Along the way, he invented CS! He solved the Decision Problem in the negative, by showing that no such algorithm existed—we've already seen how: He showed that there was at least one problem (the Halting Problem) for which there was no such algorithm.

[2] Just as in English, so '*Entscheidungs*' means "decision's".

8.5 Paragraph 1

8.5.1 Paragraph 1, Sentence 1

Let's turn to the first sentence of the first paragraph:

> The "*computable*" numbers may be described briefly as *the real numbers* whose expressions as a decimal are *calculable by finite means.*
> —(Turing, 1936, p. 230, my italics)[3]

8.5.1.1 "Computable"

The word 'computable' occurs in quotes here, because Turing is using it in an informal, intuitive sense. It is the sense that he will make mathematically precise in the rest of the paper.

8.5.1.2 Real Numbers

Real numbers are all of the numbers on the continuous number line. They consist of:

1. the *rational* numbers, which consist of:
   (a) the *integers*, which—in turn—consist of:
      i. the (non-negative) natural numbers (0, 1, 2, ...), and
      ii. the negative natural numbers $(-1, -2, \ldots)$, and
   (b) all other numbers that can be expressed as a ratio of integers (or that *can* be expressed in decimal notation with repeating decimals),

   and
2. the *irrational* numbers (that is, those numbers that *cannot* be expressed as a ratio of integers, such as π, $\sqrt{2}$, etc.)

But the real numbers do not include the "*complex*" numbers, such as $\sqrt{-1}$.

Every real number can be expressed "as a decimal", that is, in decimal notation. For instance:

$1 = 1.0 = 1.00 = 1.000$ (etc.)

$\frac{1}{2} = 0.5 = 0.50 = 0.500 = 0.5000$ (etc.)

$\frac{1}{3} = 0.33333\ldots$

$\frac{1}{7} = 0.142857142857\ldots$

These are all rational numbers and examples of "repeating" decimals. But the reals also include the irrational numbers, which have *non*-repeating decimals:

$\pi = 3.1415926535\ldots$

$\sqrt{2} = 1.41421356237309\ldots$

[3] In the rest of this chapter, citations from (Turing, 1936) will just be cited by section or page number (of the original version).

8.5.1.3 Finitely Calculable

Given a real number, is there an algorithm for computing its decimal representation? If so, then its "decimal [is] calculable by finite means" (because algorithms must be finite, as we saw in §7.4).

Digression and Further Reading:
Decimal notation is also called 'base-10 notation'. It is merely one example (another being binary—or base-2—notation) of what is more generally known as 'radix notation' or 'positional notation'; see http://en.wikipedia.org/wiki/Radix and http://math.comsci.us/radix/radix.html

For another discussion of the computation of real numbers, see (Hartmanis and Stearns, 1967).

(Myhill, 1972) is not directly related to Turing, but "consider[s] the notion of real numbers from a constructive point of view. The point of view requires that any real number can be *calculated*" (p. 748), that is, computed, which is (in part) what Turing's 1936 paper is about.

For the definition (in fact, the construction) of the reals from the rationals in terms of "Dedekind cuts", see the citations in §3.3.3.1, footnote 8.

For a commentary on the "reality" of "real" numbers, see (Knuth, 2001, pp. 174–175).
On their "unreality", see (Chaitin, 2006a) and (Chaitin, 2006b).

8.5.2 Paragraph 1, Last Sentence

Now, if we were really going to do a slow (and active!) reading, we would next move on to sentence 2. But, in the interests of keeping this chapter shorter than a full book (and so as not to repeat everything in (Petzold, 2008)), we'll skip to the last sentence of the paragraph:

> According to my definition, a number is computable if its decimal can be written down *by a machine*. (p. 230, my italics.)

This is probably best understood as an alternative way of expressing the first sentence: To be "calculable by finite means" is to be capable of being "written down by a machine". Perhaps the latter way of putting it extends the notion a bit, because it suggests that if a number is calculable by finite means, then that calculation can be done automatically, that is, by a machine—without human intervention. And that, after all, was the goal of all of those who tried to build calculators or computing machines, as we saw in Chapter 6. So, Turing's goal in this paper is to give a mathematical analysis of what can be accomplished by any such machine (and then to apply the results of this analysis to solving the Decision Problem).

8.6 Paragraph 2

8.6.1 Paragraph 2, Sentence 1

> In §§9, 10 I give some arguments with the intention of showing that the *computable* numbers include all numbers which could *naturally be regarded as computable*. (p. 230, my italics.)

We will look at some of those arguments later, but, right now, let's focus on the phrase 'naturally be regarded as computable'. This refers to the same informal, intuitive, pre-theoretical notion of computation that his quoted use of 'computable' referred to in the first sentence. It is the sense in which Hilbert wondered about which mathematical problems were decidable, the sense in which people used the phrase "effective computation", the sense in which people used the word 'algorithm', and so on. It is the sense in which *people* (mathematicians, in particular) can compute. And one of its crucial features is that it be finite.

The *first* occurrence of 'computable' in this sentence refers to the *formal* notion that Turing will present. Thus, this sentence is an expression of Turing's computability thesis.

8.6.2 Paragraph 2, Last Sentence

Once again, we'll skip to the last sentence of the paragraph:

> The *computable* numbers do not, however, include all *definable* numbers, and an example is given of a definable number which is not computable. (p. 230, my italics)

As we noted in §8.2, above, it is much more interesting if not all functions—or numbers—are computable. Any property that *everything* has is not especially interesting. But if there is a property that *only some* things have (and others lack), then we can begin to categorize those things and thus learn something more about them.

So Turing is promising to show us that computability is an interesting (because not a universal) property. And he's not going to do that by giving us some abstract (or "transcendental") argument; rather, he's actually going to show us a non-computable number (and, presumably, show us *why* it's not computable). We've already seen what this is: It's the (Gödel) number for an algorithm (a Turing machine) for the Halting Problem. So, in this chapter, we'll skip that part of Turing's paper. We'll also skip the rest of the introductory section of the paper, which simply lays out what Turing will cover in the rest of the paper.

8.7 Section 1, Paragraph 1: "Computing Machines"

Let's move on to Turing's Section 1, "Computing Machines". We'll look at the first paragraph and then jump to Turing's Section 9 before returning to this section.

Here is the first paragraph of Section 1:

> We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. No real attempt will be made to justify the definitions given until we reach §9. (p. 231.)

This is why we will jump to that section in a moment. But first let's continue with the present paragraph:

> For the present I shall only say that the justification [of the definitions] lies in the fact that the *human memory is necessarily limited*.
> (p. 231, my interpolation in brackets and my italics.)

Turing's point—following Hilbert—is that we humans do not have infinite means at our disposal. We all eventually die, and we cannot work infinitely fast, so the number of calculations we can make in a single lifetime is finite.

But how big is "finite"? Let's suppose, for the sake of argument, that a typical human (named 'Pat') lives as long as 100 years. And let's suppose that from the time Pat is born until the time Pat dies, Pat does nothing but compute. Obviously, this is highly unrealistic, but I want to estimate the *maximum* number of computations that a typical human could perform. The *actual* number will, of course, be far fewer. How long does a computation performed by a human take? Let's suppose that the simplest possible computation (following our notion of a "basic function" in §7.6.2) is computing the successor of a natural number, and let's suppose that it takes as long as 1 second. In Pat's lifetime, approximately 3,153,600,000 successors can be computed (because that's approximately the number of seconds in 100 years). Are there any problems that would require more than that number of computations? Yes! It has been estimated that the number of possible moves in a chess game is 10^{125}, which is about 10^{116} times as large as the largest number of computations that a human could possibly perform. In other words, we humans are not only finite, we are *very* finite!

But computer scientists and mathematicians tend to ignore such human limitations and pay attention only to the mathematical notion of finiteness. Even the mathematical notion, which is quite a bit larger than the actual human notion (for more on this, see (Knuth, 2001)), is still smaller than infinity, and so the computable numbers, as Turing defines them, include quite a bit.

8.8 Section 9: "The Extent of the Computable Numbers"

8.8.1 Section 9, Paragraphs 1 and 2

I want to skip now to Turing's §9, "The Extent of the Computable Numbers", because it is this section that contains the most fascinating part of Turing's analysis. We'll return to his §1 later. He begins as follows:

> No attempt has yet been made [in Turing's article] to show that the "computable" numbers include all numbers which would *naturally* be regarded as computable. (p. 249, my interpolation and italics.)

Again, Turing is comparing two notions of computability: the technical notion (signified by the first occurrence of the word 'computable'—in "scare quotes") and the informal or "natural" notion. He is going to argue that the first includes the second. Presumably, it is more obvious that the second (the "natural" notion) includes the first (the technical notion), that is, that if a number is technically computable, then it is "naturally" computable. The less obvious inclusion is the one that is more in need of support, that if a number is "naturally" computable, then it is technically computable. But what kind of argument would help convince us of this? Turing says:

> All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. (p. 249.)

Why is this so? Because one of the two notions—the "natural" one—is informal, and so no formal, logical argument can be based on it. This is why the Computability Thesis (that is, Turing's thesis) is a *thesis* and not a *theorem*—it is a hypothesis and not something formally provable. Nonetheless, Turing will give us "appeals to intuition", that is, *in*formal arguments, in fact, three kinds, as he says in the next paragraph:

> The arguments which I shall use are of three kinds.
>
> (*a*) A direct appeal to intuition.
>
> (*b*) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).
>
> (*c*) Giving examples of large classes of numbers which are computable. (p. 249.)

In this chapter, we will only look at (a), his *direct* appeal to intuition.

Let's return to the last sentence of paragraph 1:

> The real question at issue is "What are the possible processes which can be carried out in computing a number?" (p. 249.)

If Turing can answer this question, even informally, then he may be able to come up with a formal notion that captures the informal one. That is his "direct appeal to intuition".

Further Reading:
Robin Gandy, who was Turing's only Ph.D. student, argued "that Turing's analysis of computation by a human being does not apply directly to mechanical devices" (Gandy, 1980). This has become known as "Gandy's Thesis"; commentaries on it include: (Sieg and Byrnes, 1999) (which simplifies and generalizes Gandy's paper), (Israel, 2002), and (Shagrir, 2002).

8.8.2 Section 9, Subsection I

Turing notes about "Type (a)"—the "direct appeal to intuition"—that "this argument is only an elaboration of the ideas of §1" (p. 249). This is why we have made this digression to Turing's §9 from his §1; when we return to his §1, we will see that it summarizes his §9.

8.8.2.1 Section 9, Subsection I, Paragraph 1

The first part of the answer to the question, "What are the possible processes which can be carried out in computing a number?"—that is, the first intuition about "natural" computation—is this:

> Computing is normally done by writing certain symbols on paper. (p. 249.)

So, we need to be able to write symbols on paper. Is this true? What kind of symbols? And what kind of paper?

8.8.2.1.1 Is It True? Is *computing* normally done by writing symbols on paper? We who live in the 21st century might think that this is obviously false: Computers don't *have to* write symbols on paper in order to do their job. They do have to write symbols when we ask the computer to print a document, but they don't when we are watching a YouTube video. But remember that Turing is analyzing the "natural" notion of computing: the kind of computing that *humans* do. And his model includes arithmetic computations. Those typically *are* done by writing symbols on paper (or, perhaps, by imagining that we are writing symbols on paper, as when we do a computation "in our head").

8.8.2.1.2 What About the Paper?

> We may suppose this paper is divided into squares like a child's arithmetic book. (p. 249.)

In other words, we can use graph paper! Presumably, we can put one symbol into each square of the graph paper. So, for example, if we're going to write down the symbols for computing the sum of 43 and 87, we could write it like this:

```
-------
| |1| |
-------
| |4|3|
-------
|+|8|7|
-------
|1|3|0|
-------
```

We write '43' in two squares, then we write '+87' in three squares beneath this, aligning the ones and tens columns. To perform the computation, we compute the sum of 7 and 3, and write it as follows: The ones place of the sum ('0') is written below '7' in the ones column and the tens place of the sum ('1') is "carried" to the square above the tens place of '43'. Then the sum of 1, 4, and 8 is computed and then written as follows: The ones place of that sum, namely, '3' (which is the tens place of the sum of 43 and 87) is written below '8' in the tens column, and the tens place of that sum—namely, '1' (which is the hundreds place of the sum of 43 and 87—namely, '1')—is written in the square to the left of that '3'.

Turing continues:

> In elementary arithmetic the two-dimensional character of the paper is sometimes used. (p. 249.)

—as we have just seen.

> But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. (p. 249.)

In other words, we could have just as well (if not just as easily) written the computation thus:

```
----------------------
|1| |4|3|+|8|7|=|1|3|0|
----------------------
```

Here, we begin by writing the problem '43+87' in five successive squares, followed, perhaps, by an equals-sign. And we can write the answer in the squares following the equals-sign, writing the carried '1' in an empty square somewhere else, clearly separated (here, by a blank square) from the problem. So, the use of two-dimensional graph paper has been avoided (at the cost of some extra bookkeeping). As a consequence, Turing can say:

> I assume then that **the computation is carried out** on one-dimensional paper, *i.e.* **on a tape divided into squares**. (p. 249, my boldface.)

Here is the famous tape of what will become a Turing machine! (Note, though, that Turing has not yet said anything about the *length* of the tape; at this point, it could be finite.) We now have our paper on which we can write our symbols.

It is, perhaps, worth noting that the tape doesn't have to be this simple. As Kleene notes,

> [T]he computer is [not] restricted to taking an ant's eye view of its work, squinting at the symbol on one square at a time. ... [T]he Turing-machine squares can correspond to whole sheets of paper. If we employ sheets ruled into 20 columns and 30 lines, and authorize 99 primary symbols, there are $100^{600} = 10^{1200}$ possible square conditions, and we are at the opposite extreme. The schoolboy [sic] doing arithmetic on $8\frac{1}{2}$ by 12" sheets of ruled paper would never need, and could never utilize, all this variety.
>
> Another representation of a Turing machine tape is as a stack of IBM cards, each card regarded as a single square for the machine. —(Kleene, 1995, p. 26)

Further Reading:

> A 1923 play called *The Adding Machine* lampooned the monotony of assembly-line office work and prefigured fears about machine automation. Its main character, "Mr. Zero," writes down numbers all day long, "upon a square sheet of ruled paper." —Lepore (2018, p. 404)

The play, by Elmer Rice, is online at https://archive.org/details/THEADDINGMACHINE.

8.8.2.1.3 What About the Symbols?

> I shall also suppose that *the number of symbols which may be printed is finite*. (p. 249, my italics.)

This is the first item that Turing has put a limitation on: There are only finitely many symbols.

Actually, Turing is a bit ambiguous here: There might be infinitely many different kinds of symbols, but we're only allowed to *print* a finite number of them. Or there might only be a finite number of different kinds of symbols—with a further vagueness about how many of them we can print: If the tape is finite, then we can only print a finite number of the finite amount of symbols, but, if the tape is infinite, we could print infinitely many of the finite amount of symbols. But it is clear from what he says next that he means that there are only a finite number of different kinds of symbols.

Why finite? Because:

> If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. (p. 249.)

There are two things to consider here: Why would this be the case? And why does it matter? The answer to both of these questions is easy: If the human who is doing the computation has to be able to identify and distinguish among infinitely many symbols, surely some of them may get confused, especially if they look a lot alike! Would they have to look alike? A footnote at this point suggests why the answer is 'yes':

> If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leq x \leq 1, 0 \leq y \leq 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. (p. 249, footnote.)

That is, we may suppose that the square is 1 unit by 1 unit (say, 1 cm by 1 cm). Any symbol has to be printed in this space. Imagine that each symbol consists of very tiny points of ink (akin to pixels, but smaller!). To be able to print infinitely many different kinds of symbols in such a square, some of them are going to differ from others by just a single point of ink, and any two such symbols are going to "differ to an arbitrarily small extent" and, thus, be impossible for a human to distinguish. So, "the number of symbols which may be printed" must be finite in order for the human to be able to easily read them.

Is this really a limitation?

> The effect of this restriction of the number of symbols is not very serious. (p. 249.)

Why not? Because:

> It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols ... (pp. 249–250.)

In other words, the familiar idea of treating a sequence of symbols (a "string" of symbols, as mathematicians sometimes say) as if it were a single symbol allows us to construct as many symbols as we want from a finite number of building blocks. That is, the rules of place-value notation (for Arabic numerals) and of spelling (for words in European languages)—that is, rules that tell us how to "concatenate" our symbols (to string them together)—give us an arbitrarily large number (though still finite!) of symbols.

What about non-European languages? Turing makes a (possibly politically incorrect) joke:

> ... (Chinese, however, attempts to have an enumerable infinity of symbols). (p. 250.)

Chinese writing is pictographic and thus would seem to allow for symbols that run the risk of differing by an arbitrarily small extent, or, at least, that do not have to be constructed from a finite set of elementary symbols. As Turing also notes, using a finite number of basic symbols and rules for constructing complex symbols from them does not necessarily avoid the problem of not being able to identify or differentiate them:

> The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same. (p. 250.)

And probably you can't, either! So doesn't this mean that, even with a finite number of symbols, we're no better off than with infinitely many? Although Turing doesn't say so, we can solve this problem using the techniques he's given us: Don't try to write 15 or 16 occurrences of '9' inside one, tiny square: Write each '9' in a separate square! And then count them to decide which sequence of them contains 15 and which contains 16, which is exactly how *you* "can tell ... whether 9999999999999999 and 999999999999999 are the same."

Incidentally, (Kleene, 1995, p. 19) observes that Turing's emphasis on not allowing "an infinity of symbols" that "differ ... to an arbitrarily small extent" marks the distinction between "*digital* computation rather than *analog* computation".

The other thing that Turing leaves unspecified here is the *minimum* number of elementary symbols we need. The answer, as we saw in §7.5.1, is: two (they could be a blank and '1', or '0' and '1', or any other two symbols). Turing himself will use a few more (just as we did in our addition example above, allowing for the 10 single-digit numerals together with '+' and '=').

8.8.2.2 Section 9, Subsection I, Paragraph 2: States of Mind

So, let's assume that, to compute, we only need a 1-dimensional tape divided into squares and a finite number of symbols (minimally, two). What else?

> **(*)** The behaviour of *the computer* at any moment is determined by the symbols which *he* is observing, and *his* "state of mind" at that moment.
> (p. 250, my label and italics.)

I have always found this to be one of the most astounding and puzzling sentences! 'computer'? 'he'? 'his'? But it is only astounding or puzzling to those of us who live in the late 20th/early 21st century, when computers are machines, not humans! Recall the ad from the 1892 *New York Times* that we saw in §6.2 for a (human) computer. In 1936, when Turing was writing this article, *computers were still humans, not machines.* So, throughout this paper, whenever Turing uses the word 'computer', he means a *human* whose job it is to compute. I strongly recommend replacing (in your mind's ear, so to speak) each occurrence of the word 'computer' in this paper with the word 'clerk'.[4]

So, "the behavior of the *clerk* at any moment is determined by the symbols which he [or she!] is observing". In other words, the clerk decides what to do next by looking at the symbols, and *which* symbols the clerk looks at *partially* determines what the clerk will do. Why do I say 'partially'? Because the clerk also needs to know what to do with them: If the clerk is looking at two numerals, should they be added? Subtracted? Compared? The other information that the clerk needs is his or her "state of mind". What is that? Let's hold off on answering that question till we see what else Turing has to say.

> We may suppose that *there is a bound B to the number of symbols or squares which the computer [the clerk!] can observe at one moment.* If he[!] wishes to observe more, he must use successive observations.
> (p. 250, my interpolations and italics.)

This is the second kind of finiteness: We have a finite number of different kinds of symbols and a finite number of them that can be observed at any given time. This upper bound B can be quite small; in fact, it can equal 1 (and $B = 1$ in most formal, mathematical presentations of Turing machines), but Turing is allowing for B to be large

[4] In order to be able to use a word that sounds like 'computer' without the 21st-century implication that it is something like a Mac or a PC, some writers, such as (Sieg, 1994), use the nonce word 'comput*or*' to mean a human who computes. I prefer to call them 'clerks'.

enough so that the clerk can read a single word without having to spell it out letter by letter, or a single numeral without having to count the number of its digits (presumably, the length of '9999999999999999' exceeds any reasonable *B* for humans). "Successive observations" will require the clerk to be able to move his or her eyes one square at a time to the left or right.

> We will also suppose that *the number of states of mind which need to be taken into account is finite.* (p. 250, my italics.)

Here, we have a third kind of finiteness. But we still don't know exactly what a "state of mind" is. Turing does tell us that:

> If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. (p. 250.)

—just as is the case with the number of symbols. And he also tells us that "the use of more complicated states of mind can be avoided by writing more symbols on the tape" (p. 250), but why that is the case is not at all obvious at this point. (Keep in mind, however, that we have jumped ahead from Turing's §1, so perhaps something that he said between then and now would have clarified this. Nevertheless, let's see what we can figure out.)

Further Reading: For more on the notion of bounds, see (Sieg, 2006, p. 16).

8.8.2.3 Section 9, Subsection I, Paragraph 3: Operations

So, a clerk who is going to compute needs only a (possibly finite) tape divided into squares and a finite number of different kinds of symbols; the clerk can look at only a bounded number of them at a time; and the clerk can be in only a finite number of "states of mind" at a time. Moreover, what the clerk can do (the clerk's "behavior") is determined by the observed symbols and his or her "state of mind".

What kinds of behaviors can the clerk perform?

> Let us imagine the operations performed by the computer [the clerk] to be split up into "*simple operations*" which are so elementary that it is not easy to imagine them further divided. (p. 250, my interpolation and italics.)

These are going to be the basic operations, the ones that all other operations will be constructed from. What could they be? This is an important question, because this is going to be the heart of computation.

> Every such operation consists of some *change of the physical system* consisting of the computer [the clerk] and his[!] tape. (p. 250, my italics and interpolations.)

So, what "changes of the physical system" can the clerk make? The only things that can be changed are the clerk's state of mind (i.e., the clerk can change him- or herself, so to speak) and the tape, which would mean changing a symbol on the tape or changing which symbol is being observed. What else could there be? That's all we have to

manipulate: the clerk, the tape, and the symbols. And all we've been told so far is that the clerk can write a symbol on the tape or observe one that's already written. Turing makes this clear in the next sentence:

> We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer [by the clerk] (possibly with a special order), and the state of mind of the computer [of the clerk].
> (p. 250, my interpolations.)

The "system" is the clerk, the tape, and the symbols. The only things we can know, or need to know, are:

- which symbols are on the tape,
- where they are (their "sequence"),
- which are being observed (and in which order—the clerk might be looking from left to right, from right to left, and so on), and
- what the clerk's (still mysterious) "state of mind" is.

Here is the first "simple operation":

> We may suppose that in a simple operation not more than *one symbol is altered*. Any other changes can be split up into simple changes of this kind.
> (p. 250, my italics.)

Altering a single symbol in a single square is a "simple" operation, that is, a "basic" operation (or "basic program") in the sense of our discussion in Chapter 7. (And alterations of sequences of symbols can be accomplished by altering the single symbols in the sequence.) How do you alter a symbol? You replace it with another one; that is, you write down a (possibly different) symbol. (And perhaps you are allowed to erase a symbol, but that can be thought of as writing a special "blank" symbol, '♭'.)

Further Reading:
However, the ability to erase has a downside: It destroys information, making it difficult, if not impossible, to *reverse* a computation. See, for example, Brian Hayes 2014b, p. 23.

Which symbols can be altered? If the clerk is looking at the symbol in the first square, can the clerk alter the symbol in the 15th square? Yes, but only by *first* observing the 15th square and *then* changing it:

> The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always "observed" squares. (p. 250.)

But wait a minute! If the clerk has to be able to find the 15th square, isn't that a kind of operation?

8.8.2.4 Section 9, Subsection I, Paragraph 4: Operations

Yes:

> Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the computer [by the clerk]. (p. 250, my interpolation.)

And how does the clerk do that? Is "finding the 15th square" a "simple" operation? Maybe. How about "finding the 9999999999999999th square"? No:

> I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square. (p. 250.)

So here we have a fourth kind of boundedness or finiteness: The clerk can only look a certain bounded distance away. How far can the distance be? Some plausible lengths are the length of a typical word or small numeral (so L could equal B). The minimum is, of course, 1 square (taking $L = B = 1$). So, another "simple" operation is looking one square to the left or to the right (and, of course, the ability to repeat that operation, so that the clerk can, eventually, find the 15th or the 9999999999999999th square).

8.8.2.5 Section 9, Subsection I, Paragraph 5: More Operations

What about a different kind of candidate for a "simple" operation: "find a square that contains the special symbol x":

> In connection with "immediate recognisability", it may be thought that there are other kinds of square which are immediately recognisable. In particular, squares marked by special symbols might be taken as immediately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. (pp. 250–252.)

So, Turing allows such an operation as being "simple", because it doesn't violate the finiteness limitations. But he doesn't have to allow them. How would the clerk be able to find the only square that contains the special symbol x (assuming that there is one)? By first observing the current square. If x isn't on that square, then observe the next square to the left. If x isn't on that square, then observe the square to the right of the first one (by observing the square two squares to the right of the current one). And so on, moving back and forth, till a square with x is found. What if the clerk needs to find a sequence of squares marked with a sequence of special symbols?

> If, on the other hand, they [that is, the squares marked by special symbols] are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. (p. 251, my interpolation.)

I won't follow Turing's illustration of how this can be done. Suffice it to say that it is similar to what I just sketched out as a way of avoiding having to include "finding a special square" as a "simple" operation, and Turing admits as much:

> If in spite of this it is still thought that there are other "immediately recognisable" squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. (p. 251.)

In other words, other apparently "simple" operations that can be analyzed into some combination of the simplest operations of writing a symbol and observing are acceptable. It is worth noting that this can be interpreted as a claim that "subroutines" can be thought of as single operations—this is the "procedural abstraction" or "named procedure" operation discussed in §7.5.3.

8.8.2.6 Section 9, Subsection I, Paragraph 6: Summary of Operations

Turing now summarizes his analysis of the minimum that a human computer (what I have been calling a "clerk") needs to be able to do in order to compute:

> The simple operations must therefore include:
>
> (*a*) Changes of the symbol on one of the observed squares.
>
> (*b*) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.
>
> It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:
>
> (*A*) A possible change (*a*) of symbol together with a possible change of state of mind.
>
> (*B*) A possible change (*b*) of observed squares, together with a possible change of state of mind. (p. 251.)

In other words, the two basic operations are (*A*) to write a symbol on the tape (and to change your "state of mind") and (*B*) to look somewhere else on the tape (and to change your "state of mind"). That's it: writing and looking! Well, and "changing your state of mind", which we haven't yet clarified, but will, next.

8.8.2.7 Section 9, Subsection I, Paragraph 7

8.8.2.7.1 Conditions. How does the clerk know which of these two things (writing or looking) to do? Turing's next remark tells us:

> The operation actually performed is determined, as has been suggested on p. 250, by the state of mind of the computer [that is, of the clerk] and the observed symbols. In particular, they determine the state of mind of the computer [that is, of the clerk] *after* the operation is carried out.
> (p. 251, my interpolation and italics.)

The passage on p. 250 that Turing is referring to is the one that I marked '(*)' and called 'astounding', above; it says roughly the same thing as the present passage. So, what Turing is saying here is that the clerk should

- *first* consider his or her state of mind *and* where he or she is currently looking on the paper—that is, consider the current *condition* of the clerk and the paper,
- *then* decide what to do next
  (either write something there or look somewhere else)—that is, perfrm an *action*,

  and,
- *finally*, change his or her state of mind.

Of course, after doing that, the clerk is in a (possibly) new condition—a (possibly) new state of mind and looking at a (possibly) new location on the paper—which means that the clerk is ready to do the next thing.

8.8.2.7.2 States of Mind Clarified. Now, think about a typical computer program, especially an old-fashioned one, such as those written in (early versions of) Basic or Fortran, where each line of the program has a line number and a statement to be executed (a "command"). The computer (and here I mean the machine, not a clerk) starts at the first line number, executes the command, and then (typically) moves to the next line number. In "atypical" cases, the command might be a "jump" or "go to" command, which causes the computer to move to a different line number. At whatever line number the computer has moved to after executing the first command, it executes the command at that new line number. And so on.

But, if you compare this description with Turing's, you will see that *what corresponds to the line number of a program is Turing's notion of a "state of mind"!* And what corresponds to the currently observed symbol? It is the current input to the program! (Or, perhaps slightly more accurately, it is the current state of all "switches" or registers.)

So, let's paraphrase Turing's description of the basic operation that a clerk performs when computing. We'll write the paraphrase in terms of a computer program that the clerk is following:

> The operation performed is determined by the current line number of the program and the current input. The simple operations are: (a) print a symbol and (b) move 1 square left or right on the tape (which is tantamount to accepting new input), followed by changing to a new line number.

We can also say this in a slightly different way:

> If the current line number is N and the current input is I,
> then print or move (or both) and go to line N'.

And a program for such a computer will consist of lines of "code" that look like this:

```
Line N: if input = I
            then
              begin
                print (or move);
                go to Line N'
              end
```

8.8.2.8 Section 9, Subsection I, Paragraph 8

8.8.2.8.1 The Turing Machine. I said above that passage (*) was "astounding"; here is its sequel:

> We may now construct a *machine* to do the work of this *computer.*
> (p. 251, my italics.)

Reading this sentence out of context can make it sound very confusing; after all, isn't a computer a machine? But, as we have seen, a computer (for Turing) is a human clerk who computes. And what Turing is now saying is that the *human* can be replaced by a *machine*, that is, by what we now call a computer (a mechanical device). This sentence marks the end of Turing's analysis of what a human computer does and the beginning of his mathematical construction of a mechanical computer that can do what the human does. His description of it here is very compact; it begins as follows:

> To each state of mind of the computer [of the clerk!] corresponds an "*m*-configuration" of the machine. (p. 251, my interpolation.)

So, an *m*-configuration is something in the machine that corresponds to a line number of a program.[5] But, considering modern computers and programs, programs are separate from the computers that run them, so what could Turing mean when he says that an *m*-configuration belongs to a machine? He means that the machine is "hardwired" (as we would now say) to execute exactly one program (exactly one algorithm). Being hardwired, no separate program needs to be written out; it is already "compiled" into the machine. A "Turing machine" can do one and only one thing; it can compute one and only one function, using an algorithm that is hardwired into it.

Turing continues:

> The *machine* scans *B* squares corresponding to the *B* squares observed by the *computer*. (p. 251, my italics.)

A modern "translation" of this sentence would say: "The *computer* scans *B* squares corresponding to the *B* squares observed by the *clerk*." The clerk is limited to observing a maximum of *B* squares on the tape, as we saw above (in an earlier quote from p. 250). The machine analogue of that is to move, or "scan", *B* squares to the left or right on the tape. In modern mathematical treatments of Turing machines, $B = 1$.

Turing continues:

> In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than *L* squares from one of the other scanned squares. (pp. 251–252.)

In other words, the machine (the Turing machine, or modern hardwired computer) can pay attention to *B* squares at a time, and each line of its program allows it to print a

[5] Why '*m*'? It could stand for 'man', on the grounds that this is a machine analogue of a (hu)man's state of mind; or it could stand for 'mental', on the grounds that it is an analogue of a state of mind. But I think it most likely stands for 'machine', because it is a configuration, or state, of a machine. Of course, Turing might have intended it to be ambiguous among all these options.

new symbol on any of those squares or move to any other square that is no more than L squares away from any of the B squares. Again, modern treatments simplify this: The machine is scanning a single square, and each line of its program allows it to print a new symbol on that square or to move one square to its left or right (or both print and move).

Which "move" should the machine make?

> The move which is done, and the succeeding configuration, [that is, the next *m*-configuration; that is, the next step in the algorithm], are determined by the scanned symbol and the [current] *m*-configuration. (p. 252, my interpolations.)

That is, the move that the machine should make, as well as the *next* *m*-configuation (that is, the next step in the algorithm) are determined by the currently scanned symbol and the *current* *m*-configuration. Or, put in terms of computer programs, the instruction on the current line number together with the current input together determine what to do *now* (print, move, or both) and what to do *next* (which instruction to carry out next).

Digression:
When we think of a machine that prints on a tape, we usually think of the tape as moving through a stationary machine. But in the case of the Turing machine, it is the machine that moves, not the tape! The reason for this is simple: The machine is simulating the actions of a human computer, who writes on different parts of a piece of paper: It is the human who moves, not the paper.

The children's game of Candyland is like a Turing machine: The (randomly shuffled) deck of color cards is like the Turing-machine table, telling us what to do. The path laid out on the board is analogous to the tape. At each point, we can move a marker left or right. (Some squares on the path have other instructions on them, but those could have been encoded in the cards.) The game is completely deterministic, except that it doesn't necessarily "compute" anything of interest because of the random arrangement of the cards. Chess is also completely deterministic, but so "large" that we can't play it deterministically.

(For another candy analogy, see Figure 8.1.)

8.8.2.8.2 Turing's (Computability) Thesis.

> The machines just described do not differ very essentially from computing machines as defined in §2, and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the computer. (p. 252.)

As for the first clause, please recall that we are in the middle of a very long digression in which we have skipped ahead to Turing's §9 from Turing's §1; we have not yet read Turing's §2. When we do, we will see a more detailed version of the machines that Turing has just described for us here in §9.

The next clause is a bit ambiguous. When Turing says "any machine of *this* type", is he referring to the machines of §9 or the machines of §2? It probably doesn't matter, because he has said that the two kinds of machines "do not differ very essentially" from

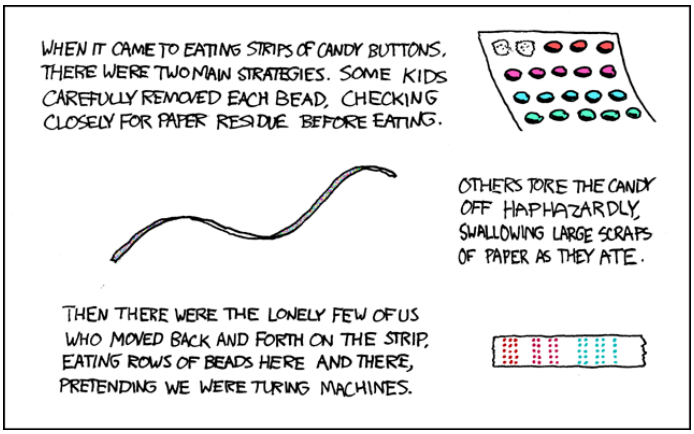


Figure 8.1: https://xkcd.com/205/

each other. But I think that he is, in fact, referring to the machines of §9; "computing machines" are the ones that are "defined in §2".

The last phrase is of more significance: These ("computing") *machines* (of Turing's §2) "compute the same sequence ... computed by the" *clerk*. In other words, whatever a human clerk can do, these machines can also do. What a human clerk can do (i.e., which sequences, or functions, a human clerk can compute) is captured by the *informal* notion of algorithm or computation. "These machines" are a formal counterpart, a formal "explication" of that informal notion. So this last phrase is a statement of Turing's thesis (that is, the Computability Thesis).

What about the other direction? Can a human clerk do everything that one of these machines can do? Or are these machines in some way more powerful than humans? I think the answer should be fairly obvious: Given the way the machines are constructed on the basis of what it is that humans can do, surely a human could follow one of the programs for these machines. So humans can do everything that one of the machines can do, and—by Turing's thesis—these machines can do everything that humans can do (well, everything that is computable in the informal sense). But these are contentious matters, and we will return to them when we consider the controversies surrounding hypercomputation (Chapter 11) and AI (Chapter 19).

8.8.2.8.3 Turing Machines as AI Programs. As we have seen, in order to investigate the *Entscheidungsproblem*,

> ... Turing asked in the historical context in which he found himself *the* pertinent question, namely, what are the possible processes **a human being** can carry out (when computing a number or, equivalently, determining algorithmically the value of a number theoretic function)? —(Sieg, 2000, p. 6; original italics, my boldface)

That is,

> Turing machines appear [in Turing's paper] as a result, a codification, of his analysis of calculations by humans. —(Gandy, 1988, p. 82)

This strategy underlies much of CS, as Alan Perlis observed:

> The intent [of a first computer science course should be] to reveal, through ... examples, how analysis of some *intuitively* performed *human* tasks leads to *mechanical* algorithms accomplishable by a *machine*.
> —(Perlis, 1962, p. 189, my italics)

But not just CS in general. The branch of CS that analyzes how humans perform a task and then designs computer programs to do the same thing is AI; so, in Section 9, **Turing has developed the first AI program!**

One of the founders of AI, John McCarthy, made a similar observation:

> The subject of computation is essentially that of artificial intelligence since the development of computation is in the direction of making machines carry out ever more complex and sophisticated processes, i.e. to behave as intelligently as possible. (McCarthy, 1963, §4.2, p. 38).

This follows from McCarthy's earlier definition (see §3.9.2) of computation as the science of how to get machines to carry out intellectual processes.

Turing was not unaware of this aspect of his work:

> One way of setting about our task of building a 'thinking machine' would be to take a man [sic] as a whole and to try to replace all the parts of him by machinery. —(Turing, 1948, p. 420) as cited in (Proudfoot and Copeland, 2012, p. 2)

But that's almost exactly what Turing's analysis of human computation in his 1936 paper does (at least in part): It takes a human's computational abilities and "replaces" them by (abstract) machinery.

One reason that the Turing machine can be considered as the first AI program is that "the elementary processes underlying human thinking are essentially the same as the computer's elementary information processes" (Simon, 1977, p. 1187), namely, those processes that can be carried out by a Turing machine. And the reason that they are "essentially the same" is simply that that is how the Turing machine is defined, namely, by considering what "elementary processes underl[ie] ... human thinking", as seen in (Turing, 1936, §9).

8.9 Section 1, continued

For now, we have come to the end of our digression, and we now return to Turing's §1, "Computing Machines".

8.9.1 Section 1, Paragraph 2

This is the paragraph in which Turing gives a more detailed presentation of his abstract computing machine, the outcome of his detailed analysis from §9 of *human* computing. He begins as follows:

> We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_R$ which will be called "*m*-configurations". (p. 231.)

Why "may" we do this? Turing will give his justification in his §9, which we have just finished studying. By a "man", Turing of course means a human, not merely a male human. To compute a real number is to compute the output of a real-valued function. And, as we have already seen, an *m*-configuration is a line of a computer program, that is, a step in an algorithm. Here, Turing is saying that each such algorithm has a finite number (namely, R) of steps, each labeled q_i. Put otherwise, (human, or informal) computation can be "compared with" (and, by Turing's thesis, identified with) a finite algorithm.

What else is needed?

> The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". (p. 231.)

There are a couple of things to note here. First, from our study of Turing's §9, we know why this is the case and what, exactly, the tape, squares, and symbols are supposed to be and why they are the way they are.

But, second, why does he put those three words in "scare quotes"? There are two possible answers. I suspect that the real answer is that Turing hasn't, at this point in his paper, explained in detail what they are; that comes later, in his §9.

But there is another possible reason, a mathematical or logical reason: In Turing's formal, mathematical notion of a computing machine, the concepts of "tape", "squares" of a tape, and "symbols" are really *undefined* (or *primitive*) *terms* in exactly the same way that 'point', 'line', and 'plane' are undefined (or primitive) terms in Euclidean plane geometry. As Hilbert famously observed, "One must be able to say at all times—instead of points, lines, and planes—tables, chairs, and beer mugs".[6] So, here, too, one must be able to say at all times—instead of tapes, squares, and symbols—tables, chairs, and beer mugs. (But I'll use place settings instead of chairs; it will make more sense, as you will see.) A Turing machine, we might say, must have a table. Each table must have a sequence of place settings associated with it (so we must be able to talk about the *n*th place setting at a table). And each place setting can have a beer mug on it; there might be different kinds of beer mugs, but they have to be able to be distinguished from each other, so that we don't confuse them. In other words, *it is the logical or mathematical* ***structure*** *of a computing machine that matters, not what it is*

[6] See Hilbert's *Gesammelte Abhandlungen* ("Complete Works"), vol. 3, p. 403, as cited in Coffa 1991, p. 135; see also Stewart Shapiro 2009, p. 176. Elsewhere, Hilbert used a different example:

> ... it is surely obvious that every theory is only a scaffolding or schema of concepts together with their necessary relations to one another, and that the basic elements can be thought of in any way one likes. If in speaking of my points, I think of some system of things, e.g., the system: love, law, chimney-sweep ... and then assume all my axioms as relations between these things, then my propositions, e.g., Pythagoras' theorem, are also valid for these things ... [A]ny theory can always be applied to infinitely many systems of basic elements. (Quoted in Stewart Shapiro 2013, p. 168.)

made of. So, a "tape" doesn't have to be made of paper (it could be a table), a "square" doesn't have to be a regular quadrilateral that is physically part of the "tape" (it could be a place setting at a table), and "symbols" only have to be such that a "square" can "bear" one (for example, a numeral can be written on a square of the tape, or a beer mug can be placed at a place setting belonging to a table).

Further Reading:
An excellent discussion of this kind of abstraction can be found in Cohen and Nagel 1934, Ch. 7, "The Nature of a Logical or Mathematical System".

Turing continues:

> At any moment there is just one square, say the r-th, bearing the symbol $\mathfrak{S}(r)$ which is "in the machine". We may call this square the "scanned square". (p. 231.)

First, '$\mathfrak{S}$' is just the capital letter '*S*' in a font called "German Fraktur", or "black letter". It's a bit hard to read, so I will replace it with '*S*' in what follows (even when quoting Turing).

Note, second, that this seems to be a slight simplification of his §9 analysis, with $B = 1$. Second, being "in the machine" might be another undefined (or primitive) term merely indicating a relationship between the machine and something else. But what else? Turing's punctuation allows for some ambiguity.[7] It might be the *symbol* (whatever it is, '0', '1', or a beer mug) that is in the machine, or it might be the *scanned square*. I think that it is the latter, from remarks that he makes next.

> The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". (p. 231.)

Here, Turing's scare quotes around 'directly aware', together with the hedge 'so to speak', clearly indicate that he is not intending to anthropomorphize his machine. His machines are not really "aware" of anything; only humans can be really "aware" of things. But the machine analogue of human awareness is: being a scanned symbol. There is nothing anthropomorphic about that: Either a square is being scanned (perhaps a light is shining on a particular place setting at the table) or it isn't, and either there is a symbol on the scanned square (there is a beer mug at the lighted place setting), or there isn't.

> However, by altering its *m*-configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. (p. 231.)

What does this mean? Let's try to paraphrase it: "By altering the line number of its program, the computing machine can effectively . . . "—can effectively do what? It can

[7] More precisely, his *lack* of punctuation: If 'which' had been preceded by a comma, then "which is 'in the machin would have been a "non-restrictive relative clause" that refers to the *square*. With no comma, the "which" clause is a "restrictive" relative clause modifying '*symbol* $S(r)$'. For more on relative clauses, see " 'Which' vs. 'that' ", online at http://www.cse.buffalo.edu/~rapaport/howtowrite.html#whichVthat

"remember previously scanned symbols". This is to be contrasted with the *currently* scanned symbol. How does the machine "remember" "by altering a line number"? Well, how would it "remember" what symbol was on, say, the 3rd square if it's now on the 4th square? It would have to move left one square and scan the symbol that's there. To do that, it would have to have an instruction to move left. And to do *that*, it would need to go to that instruction, which is just another way of saying that it would have to "alter its *m*-configuration". (For a different slow-reading analysis of this sentence, see (Dresner, 2003) and (Dresner, 2012).)

> The *possible* behaviour of the machine at any moment is determined by the *m*-configuration q_n and the scanned symbol $S(r)$. (p. 231, my italics.)

It is only a *possible* behavior, because a given line of a program is only executed when control has passed to that line. If it is not being executed at a given moment, then it is only *possible* behavior, not *actual* behavior. The machine's *m*-configuration is the analogue of a line number of a program, and the scanned symbol is the analogue of the external input to the machine.[8]

> This pair $q_n, S(r)$ will be called the "configuration": thus the configuration determines the possible behaviour of the machine. (p. 231.)

Giving a single name ('configuration') to the *combination* of the *m*-configuration *and* the currently scanned symbol reinforces the idea that the *m*-configuration alone is an analogue of a line number and that this combination is the condition (or antecedent) of a condition-action (or a conditional) statement: Line q_n begins, "**if** the currently scanned symbol is $S(r)$, **then** ...", or "**if** the current instruction is the one on line q_n **and if** the currently scanned symbol is $S(r)$, **then** ...".

What follows the '**then**'? That is, what should the machine do if the condition is satisfied?

> In some of the configurations in which the scanned square is blank (*i.e.* bears no symbol) the machine **writes** down a new symbol on the scanned square: in other configurations it **erases** the scanned symbol. The machine may also **change the square** which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations **the *m*-configuration may be changed**.
> (p. 231, my boldface.)

So, we have 5 operations:

1. write a new symbol
2. erase the scanned symbol
3. shift 1 square left

[8] It is also possible, given what the reader knows at this stage of Turing's paper, not yet having read his §9, that an *m*-configuration is the entire internal state of the machine, perhaps encoding what could be called the machine's "prior" or "background" knowledge—in contrast to external information from the outside world, encoded in the scanned symbol. On whether symbols on the tape are analogous to external input, see §8.10.1, below, and Chapters 11 and 17, as well as (Rapaport, 2017a).

4. shift 1 square right

5. change *m*-configuration.

As we saw in §7.5.2, the first 4 of these can be simplified to only 2 operations, each of which is slightly more complex:

1′ write a new symbol (*including* ♭)

2′ shift (which is now an operation that takes an argument: left or right).

There are four things to note:

1. The symbols are left unspecified (which is why we can feel free to add a "blank" symbol), though, as we have seen, they can be limited to just '0' and '1' (and maybe also '♭').

2. Turing has, again, simplified his §9 analysis, letting $L = 1$.

3. "Change *m*-configuration" is essentially a "jump" or "go to" instruction. The whole point of structured programming, as we have seen, is that this can be eliminated—so we really only need the first two of our slightly more complex operations, as long as we require our programs to be structured.

4. There is no "halt" command.
(In §8.10.3, below, we will see why this is not needed.)

Turing next clarifies what symbols are needed. Recall that the kind of computation that Turing is interested in is the computation of the decimal of a real number.

> Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to "assist the memory". It will only be these rough notes which will be liable to erasure. (pp. 231–232.)

So, either we need symbols for the 10 Arabic numerals (if we write the real number in decimal notation) or we only need symbols for the 2 binary numerals (if we write the real number in binary notation). Any other symbols are merely used for bookkeeping, and they (and only they) can be erased afterwards, leaving a "clean" tape with only the answer on it.

There is one more thing to keep in mind: Every real number (in decimal notation)[9] has an infinite sequence of digits to the right of the decimal point, even if it is an integer or (a non-integer) rational number, which are typically written with either no digits, or a finite number of digits, in the decimal expansion (1, 1.0, 2.5, etc.). If the number is an integer, this is an infinite sequence of '0's; for example, $1 = 1.000000000000\ldots$ (which I will abbreviate as $1.\overline{0}$). If the number is rational, this is an infinite sequence of some repeating subsequence; for example:

[9] Similar remarks can be made for binary notation.

$\frac{1}{2} = 0.500000000000\ldots = 0.5\overline{0}$

$\frac{1}{3} = 0.333333333333\ldots = 0.\overline{3}$

$\frac{1}{7} = 0.142857142857\ldots = 0.\overline{142857}$

And if the number is irrational, this is an infinite, *non*-repeating sequence; for example:

$\sqrt{2} = 1.41421356237309\ldots$

$\pi = 3.1415926535\ldots$

What this means is that one of Turing's computing machines should *never halt* when computing (i.e., writing out) the decimal of a real number. It should only halt if it is writing down a *finite* sequence, and it can do this in two ways: It could write down the finite sequence and then halt. Or it could write down the finite sequence and then go into an infinite loop (either rewriting the last digit over and over in the same square, or just looping in a do-nothing operation such as the empty program).

8.9.2 Section 1, Paragraph 3

Finally,

> It is my contention that these operations include all those which are used in the computation of a number. (p. 232.)

This is another statement of Turing's version of the Computability Thesis: To compute, all you have to do is arrange the operations of writing and shifting in a certain way. The way they are arranged—what is now called "the control structure of a computer program"—is controlled by the "configuration" and the change in *m*-configuration (or, in modern structured programming, by Böhm & Jacopini's three control structures (that is, grammar rules) of sequence, selection, and while-repetition). For Turing, it goes unsaid that all computation can be reduced to the computation of a number; this is the insight we discussed in §7.5.1 that all the information about any computable problem can be represented using only '0' and '1'; hence, any information—including pictures and sounds—can be represented as a number. (But it is also important to realize that this kind of universal binary representation of information doesn't have to be thought of as a number, because the two symbols don't have to be '0' and '1'!)

8.10 Section 2: "Definitions"

We are now ready to look at the section in which Turing's "computing machines" are defined, as we read in his §9, subsection I, paragraph 8 (see our §8.8.2.8.2, above).

8.10.1 "Automatic Machines"

Turing begins by giving us a sequence of definitions. The first is the most famous:

> If at each stage the motion of a machine (in the sense of §1) is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a*-machine). (p. 232.)

Clearly, such a machine's "motion" (or behavior) is at least *partly* determined by its configuration (that is, by its *m*-configuration, or line number, together with its currently scanned symbol). Might it be determined by anything else? For all that Turing has said so far, maybe such a machine's human operator could "help" it along by moving the tape for it, or by writing something on the tape. This definition rules that out by limiting our consideration to such machines whose "motion" "is *completely* determined by the configuration". So, a human operator is not allowed to "help" it in any way: No cheating allowed!

Turing may have called such a machine an '*a*-machine'. *We* now call them—in his honor—'Turing machines'. (Alonzo Church seems to have been the first person to use this term, in his review of Turing's paper (Church, 1937).)

What about machines that get outside help?

> For some purposes we might use machines (choice machines or *c*-machines) whose motion is only partially determined by the configuration (hence the use of the word "possible" in §1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. (p. 232.)

First, note that Turing's explanation of the use of 'possible' may be slightly different from mine. But I think that they are consistent explanations. In the previous statements, Turing used 'possible' to *limit* the kind of operations that a Turing machine could perform. Here, he is introducing a kind of machine that has another kind of possible operation: writing, moving, or changing *m*-configuration *not* as the result of an explicit instruction *but* as the result of a "choice ... made by an external operator". Note, by the way, that this external operator *doesn't have to be a human*; it *could* be another Turing machine! Such *c*-machines are closely related to "oracle" machines, which Turing introduced in his doctoral dissertation. We will return to the topic of choice machines and oracle machines in Chapter 11 on hypercomputation and Chapter 17 on the relation of computation to the world (see also (Rapaport, 2017a)).

Further Reading:
For more on Turing's oracle machines, see Feferman 1992, 2006b, and Soare 2016, pp. xxi–xxii and §§3.2 & 17.4.

8.10.2 "Computing Machines"

8.10.2.1 Paragraph 1

Turing gives us some more definitions:

> If an *a*-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. (p. 232.)

The principal definition here is that of 'computing machine', a special case of an *a*- (or Turing) machine that outputs its results as a binary numeral (in accordance with the first insight we discussed, in §7.5.1). Once again, here Turing is simplifying his §9 analysis of human computation, restricting the symbols to '0' and '1'. Well, not quite, because he also allows "symbols of the second kind", used for bookkeeping purposes or intermediate computations. Note, however, that any symbol of the second kind could be replaced—at the computational cost of more processing—by sequences of '0's and '1's.

Turing continues:

> If the machine is supplied with a blank tape and set in motion, starting from the correct initial *m*-configuration, the subsequence of the symbols printed by it which are of the first kind will be called the *sequence computed by the machine.* (p. 232.)

Here, he seems to be allowing for some of the symbols of the *second* kind to remain on the tape, so that only a *sub*sequence of the printed output constitutes the result of the computation. In other words, these secondary symbols need not be erased. One way to think of this is to compare it to the way we write decimal numerals greater than 999, namely, with the punctuation aid of the non-numerical symbols known as a 'comma' and a 'decimal point': 1,234,567.89

In the previous paragraph, I almost wrote, "to remain on the tape *after the computation halts*". But does it halt? It can't—because every real number has an *infinite* decimal part![10] The secondary symbols could still be erased, during the computation; that's not of great significance (obviously, it's easier to not erase them and to just ignore them). The important point to remember is that computations of decimal representations of real numbers never halt. We'll return to this in a moment.

One more small point that simplifies matters:

> The real number whose expression as a binary decimal is obtained by prefacing this sequence by a decimal point is called the *number computed by the machine.* (p. 232.)

What about the part of the expression that is to the *left* of the decimal point? It looks as if the only numbers that Turing is interested in computing are the reals between 0 and 1 (presumably including 0, but excluding 1). Does this matter? Not really; first, all reals can be mapped to this interval, and, second, any other real can be computed

[10] What about real numbers like 2 or 2.1, which seem to either lack a decimal part or have only a finite one? But don't forget that $2 = 2.000\ldots$ and $2.1 = 2.1000\ldots$, so there are infinitely many 0s in their decimal parts.

simply by computing its "non-decimal" part in the same way. Restricting our attention to this subset of the reals simplifies the discussion without loss of generality. (We'll return to this in §8.10.4, below.)

8.10.2.2 Paragraph 2

Two more definitions:

> At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the *m*-configuration will be said to describe the *complete configuration* at that stage. The changes of the machine and tape between successive complete configurations will be called the *moves* of the machine. (p. 232.)

Three points to note: First, at any stage of the motion of the machine, only a *finite* number of symbols will have been printed, so it is perfectly legitimate to speak of "the complete sequence of all symbols on the tape" even though every real number has *infinitely* many numerals after the decimal point. Second, the sequence of all symbols on the tape probably includes all occurrences of '♭' that do not occur after the last non-blank square (that is, that *do* occur *before* the last non-blank square); otherwise, there would be no way to distinguish the sequence $\langle 0, 0, 1, \flat, 0\rangle$ from the sequence $\langle \flat, 0, \flat, 0, \flat, 1, 0\rangle$.

Third, we now have three notions called 'configurations'; let's summarize them for convenience:

1. *m*-configuration = line number, q_n, of a program for a Turing machine.
2. configuration = the pair: $\langle q_n, S(r)\rangle$,
   where $S(r)$ is the symbol on the currently scanned square, r.
3. complete configuration = the triple:
   $\langle r$, the sequence of all symbols on the tape,[11] $q_n\rangle$.

8.10.3 "Circular and Circle-Free Machines"

8.10.3.1 Paragraph 1

We now come to what I have found to be one of the most puzzling sections of Turing's paper. It begins with the following definitions:

> If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called *circular*. Otherwise it is said to be *circle-free*. (p. 233.)

Let's take this slowly: A computing machine is a Turing machine that only prints a binary representation of a real number together with a few symbols of the second kind. If such a machine "*never* writes down more than a finite number of" '0's and '1's, then, trivially, it has *only* written down a *finite* number of such symbols. *That means that it*

[11] As described in the last paragraph.

has halted! And, in that case, Turing wants to call it 'circular'! But, to my ears, at least, 'circular' sounds like 'looping', which, in turn, sounds like it means "*not* halting".

And, if it *does* write down more than a finite number of '0's and '1's, then, trivially, it writes down *infinitely* many of them. *That means that it does not halt!* In that case, Turing wants to call it 'circle-free'! But that sounds like 'loop-free', which, in turn, sounds like it means that it *does* halt.

Further Reading:
Other commentators have made the same observation:

> In Turing's terminology, circularity means that the machine never writes down more than a finite number of symbols (halting behaviour). A non-circular machine is a machine that never halts and keeps printing digits of some computable sequence of numbers. —(De Mol and Primiero, 2015, pp. 197–198, footnote 11)

What's going on? Before looking ahead to see if, or how, Turing clarifies this, here's one guess: The only way that a Turing machine can print a finite number of "figures" (Turing's name for '0' and '1') and still "be circular" (which I am interpreting to mean "loop") is for it to keep repeating printing—that is, to "overprint"—some or all of them, that is, for it to "circle back" and print some of them over and over again. (In this case, no "halt" instruction is needed!)

And the only way that a Turing machine can print infinitely many "figures" and also be "circle-free" is for it to continually print new figures to the right of the previous one that it printed (and, thus, not "circle back" to a previous square, overprinting it with the same symbol that's on it).

Is that what Turing has in mind? Let's see.

8.10.3.2 Paragraph 2

The next paragraph says:

> A machine will be circular if it reaches a configuration from which there is no possible move or if it goes on moving, and possibly printing symbols of the second kind, but cannot print any more symbols of the first kind. The significance of the term "circular" will be explained in §8. (p. 233.)

The first sentence is rather long; let's take it phrase by phrase: "A machine will be circular"—that is, will print out only a finite number of figures—if [Case 1] it reaches a configuration from which there is no possible move . . . ". That is, it will be circular if it reaches a line number q_n and a currently scanned symbol $S(r)$ from which there is no possible move. How could that be? Easy: if there's no line of the program of the form: "Line q_n: **If** currently scanned symbol = $S(r)$ **then** . . . . In that case, the machine stops,[12] because there's no instruction telling it to do anything.[13]

[12] At this point, I cannot resist recommending, once again, that you read E.M. Forster's wonderfully prescient, 1909(!) short story, "The Machine Stops", (Forster, 1909).

[13] Another possibility is that line q_n says: **If** currently scanned symbol = $S(r)$, **then** go to line q_n. In that case, the machine never stops, becuase it forever loops (circles?) back to the same line.

That's even more paradoxical than my interpretation above; here, he is clearly saying that a machine is circular if it halts! Of course, if you are the operator of a Turing machine and you are only looking at the tape (and not at the machinery), would you be able to tell the difference between a machine that was printing the same symbol over and over again on the same square and a machine that was doing nothing?[14] Probably not. So, from an external, behavioral point of view, these would seem to amount to the same thing.

But Turing goes on: A machine will also be circular "... if [Case 2] it goes on moving, and possibly printing [only] symbols of the second kind" but not printing any more "figures". Here, the crucial point is that the machine does not halt but goes on moving. It might or might not print *anything*, but, if it does, it only prints secondary symbols. So we have the following possibilities: a machine that keeps on moving, spewing out square after square of blank tape; or a machine that keeps on moving, occasionally printing a secondary symbol. In either case, it has only printed a *finite* number of *figures*. Because it has, therefore, *not* printed an *infinite* decimal representation of a real number, it has, for all practical purposes, halted—at least in the sense that it has finished its task, though it has not succeeded in computing a real number.

Once again, a machine is circular if it halts (for all practical purposes; it's still working, but just not doing anything significant). This isn't what I had in mind in my interpretation above. But it does seem to be quite clear, no matter how you interpret what Turing says, that he means that *a circular machine is one that does not compute a real number*, either by halting or by continuing on but doing nothing useful (not computing a real number). Machines that *do* compute real numbers are "circle-free", but they must also never halt; they must loop forever, in modern terms, but continually doing useful work (computing digits of the decimal expansion of a real number):

> A machine that computes a real number in this sense was called *circle-free*; one that does not (because it never prints more than a finite number of 0s and 1s) was called *circular*. —(Davis, 1995b, p. 141)

In other words, a "good" Turing machine is a "circle-free" one that does *not* halt and that continually computes a real number. This seems to be contrary to modern terminology and the standard analysis of "algorithms" that we saw in §7.4. And how does this fit in with Turing's claim at the beginning of his paper that "the 'computable' numbers may be described briefly as the real numbers whose expressions as a decimal are calculable *by finite means*" (my italics)? The only way that I can see to make these two claims consistent is to interpret "by finite means" to refer to the number of steps in an algorithm, or the amount of time needed to carry one step out, or the number of operations needed to carry one step out (in case any of the steps are not just basic operations). It cannot mean, as we have just seen, that the entire task can be completed in a finite amount of time or that it would necessarily halt.

Finally, what about the allusion to Turing's §8? That section, which we will not investigate and which is uninformatively titled "Application of the Diagonal Process", is the section in which he proves that the Halting Problem is not computable (more precisely, that a Gödel-like number of a program for the Halting Problem is not a

[14] The machine described in the text and the machine described in the previous footnote have this property.

computable number). And, pretty obviously, his proof is going to be a little bit different from the one that we sketched in §7.7 because of the difference between our modern idea that only Turing machines that halt are "good" and Turing's idea that only Turing machines that are circle-free are "good".

8.10.3.3 Coda: A Possible Explanation of 'Circular'

One possible explanation of the term 'circular' comes from the following observation: Using only *finite* means (a finite number of states, a finite number of symbols, etc.), a Turing-machine can compute *infinitely* many numbers and print *infinitely* many numerals. Machines that could not do that

> would eventually repeat themselves and Turing had attempted precisely to show how a machine with finite specifications would not be constrained to do so. —(Corry, 2017, p. 53, col. 3)

That is, machines that *were* finitely constrained and that would therefore "repeat themselves" would be "circular".

It is interesting to note that, in French, 'circular' would normally be translated as '*circulaire*'. Turing wrote a summary of his his 1936 paper in French. In that document, instead of calling machines that halted without computing a real number '*circulaire*', he called them '*méchant*'—'malicious'! Perhaps he was having second thoughts about the term 'circular' and wanted something more perspicuous.

Further Reading:
For more information on the French summary, see Corry 2017. For more on "circularity", see Petzold 2008, Ch. 10, who notes, by the way, that the concept of "halting" was introduced into the modern literature by Martin Davis (Petzold, 2008, p. 179), "despite the fact that Turing's original machines never halt!" (Petzold, 2008, p. 329). Here is a slightly different observation:

> The halting theorem is often attributed to Turing in his 1936 paper. In fact, Turing did not discuss the halting problem, which was introduced by Martin Davis in about 1952. —(Copeland and Proudfoot, 2010, p. 248, col. 2)

This is clarified in Bernhardt 2016:

> The halting problem is probably the most well-known undecidable decision problem. However, this is not the problem that Turing described in his paper.
>
> As Turing described his machines, they did not have accept states [that is, they did not halt]. They were designed to compute real numbers and so would never stop if computing an irrational number. The notion of a Turing machine was changed [from Turing's original *a*-machines] to include accept states by Stephen Kleene and Martin Davis. Once you had this new formulation of a Turing machine, you could consider the halting problem. Davis [1958] gave the halting problem its name. —Bernhardt (2016, pp. 120–121).

See also Bernhardt 2016, p. 142. For an analysis of these notions in modern terms, see (van Leeuwen and Wiedermann, 2013).

8.10.4 "Computable Sequences and Numbers"

Here are Turing's final definitions from this section. First:

> A *sequence* is said to be computable if it can be computed by a circle-free machine. (p. 233, my italics.)

Although this is presented as a definition of 'computable' (actually, of 'computable sequence'), it can, in fact, be understood as another statement of the Computability Thesis. Being "computable by a circle-free machine" is a very precise, mathematical concept. In this definition, I think that Turing is best understood as suggesting that this precise concept should replace the informal notion of being "computable". Alternatively, Turing is saying here that he will use the word 'computable' in this very precise way.

Next:

> A *number* is computable if it differs by an integer from the number computed by a circle-free machine. (p. 233, my italics.)

Circle-free machines compute (by printing out) a sequence of figures (a sequence of '0's and '1's). Such a sequence can be considered to be a decimal (actually, a binary) representation of a number between 0 and 1 (including 0, but not including 1). Here, Turing is saying that *any* real number can be said to be computable if it has the same decimal part (that is, the same part after the decimal point) of a number representable as a computable sequence of figures. So, for instance, $\pi = 3.1415926535\ldots$ differs by the integer 3 from the number $0.1415926535\ldots$, which is computable by a circle-free Turing machine; hence, π is also computable.

8.11 Section 3: "Examples of Computing Machines"

We are now ready to look at some "real" Turing machines, more precisely, "computing machines", which, recall, are "automatic" *a*-machines that print only figures ('0', '1') and maybe symbols of the second kind. Hence, they compute real numbers. Turing gives us two examples, which we will look at in detail.

8.11.1 Section 3, Example I

8.11.1.1 Section 3, Example I, Paragraph 1

> A machine can be constructed to compute the sequence 010101.... (p. 233.)

Actually, as we will see, it prints

$$0\text{b}1\text{b}0\text{b}1\text{b}0\text{b}1\text{b}\ldots$$

What real number is this? First, note that it is a *rational* number of the form $0.\overline{01}$. Treated as being written in binary notation, it $= \frac{1}{3}$; treated as being written in decimal notation, it $= \frac{1}{99}$.

> The machine is to have the four *m*-configurations "𝔟", "𝔠", "𝔣", "𝔢" and is capable of printing "0" and "1". (p. 233.)

The four line numbers are (in more legible italic font): b, c, f, e.

> The behaviour of the machine is described in the following table in which "*R*" means "the machine moves so that it scans the square immediately on the right of the one it was scanning previously". Similarly for "*L*". "*E*" means "the scanned symbol is erased" and "*P*" stands for "prints". (p. 233.)

This is clear enough. It is, however, interesting to note that it is the Turing machine that moves, not the tape! But, of course, when you do a calculation with pencil and paper, your hand moves; the paper doesn't! Of course, a pencil is really only an *output* device (it prints and erases). To turn it into a full-fledged computer (or, at least, a physical Turing machine), you need to add eyes (for input), hands (for moving left and right), and a mind (for "states of mind"). (See Figure 8.2 and the epigraph to §8.14.)

Figure 8.2: http://rhymeswithorange.com/comics/november-19-2009/, ©2009 Hilary B. Price

Before going on with this paragraph, let's look at the "table".[15] In later writings by others, such tables are sometimes called 'machine tables'; *they are computer programs for Turing machines*, written in a "Turing-machine programming language" for which Turing is now giving us the syntax and semantics.[16]

However, it is important to keep in mind that the Turing machine does not "consult" this table to decide what to do. We humans would consult it in order to simulate the Turing machine's behavior. But the Turing machine itself simply behaves *in accordance with* that table, not by *following* it. The table should be thought of as a mathematical-English description of the way that the Turing machine is "hardwired" to behave. (We'll revisit this idea in §§10.4.1 and 12.4.4.1.2.2.)

Here's the table, written a bit more legibly than in Turing's paper:

[15] Not to be confused with our table of place settings and beer mugs!

[16] That is, the grammar and meaning; see Ch. 22.

Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
b	None	$P0, R$	c
c	None	R	e
e	None	$P1, R$	f
f	None	R	b

This program consists of 4 lines. It is important to note that it is a *set* of lines, not a *sequence*: The order in which the lines are written down in the table (or "program") is irrelevant; there will never be any ambiguity as to which line is to be executed. Perhaps a better way of saying this is: There will never be any ambiguity as to which line is *causing the Turing machine to move*.

Each line consists of two principal parts: a "configuration" and a "behavior". Each configuration, as you may recall, consists of two parts: an *m*-configuration (or line number) and a symbol (namely, the currently scanned symbol). Each behavior consists also of two parts: an "operation" (one or more of E, L, R, or P) and a "final *m*-configuration" (that is, the next line number to be executed).

> This table (and all succeeding tables of the same kind) is to be understood to mean that *for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the m-configuration described in the last column.* (p. 233, my italics.)

That is, each line of the program should be understood as follows: "Under the conditions described by the configuration, do the operation and then go to the instruction at the final *m*-configuration". Or, to use Turing's other terminology: "If your current state of mind is the one listed in the current *m*-configuration, and if the symbol on the current square being scanned is the one in the *symbol* column, then do the *operation* and change your state of mind to the one in the *final m-configuration* column."

A modern way of thinking about this is to consider it to be a "production system". Production systems are an architecture introduced by Emil Post in his analysis of computation and used by many researchers in AI. A production system consists of a set of "condition-action" pairs; if the condition is satisfied, then the action is carried out. That's exactly what we have in this Turing-machine table: The configuration is the condition, and the behavior is the action.

Further Reading:
(Sieg, 2000, p. 7) notes that even Turing considered Turing machines as production systems. Post's writings on computations that have become known as "production systems" are (Post, 1941) and (Post, 1943).

For more on production systems in AI, see (Winston, 1977, pp. 357–366), (Agre, 1992, pp. 294–295), and http://www.cse.buffalo.edu/~rapaport/663/F03/prodsys.eg.html

A further qualification:

> When the second column [that is, the symbol column] is left blank, it is understood that the behaviour of the third and fourth columns applies for any symbol and for no symbol. (p. 233, my interpolation.)

That is the situation we have in this first example, where 'None' is the entry in each row of the symbol column. So the only condition determining the behavior of this Turing machine is its current "state of mind", that is, its current line number.

Finally, we need to know what the initial situation that this "production system" is intended to apply to:

> The machine starts in the *m*-configuration *b* with a blank tape. (p. 233.)

Perhaps '*b*' stands for "begin", with subsequent "states of mind" (in alphabetical as well as sequential order) being *c*, *e*, and *f* ('*f*' for "final"? What happened to '*d*'?).

Let's trace this program. We start with a blank tape, which I will show as follows:

$$\flat\flat\flat\flat\flat\flat\flat\flat\flat\flat\flat\ldots$$

We are in state *b*.

Looking at the table, we see that if we are in state *b*, then (because any symbol that might be on the tape is irrelevant), we should do the sequence of operations *P*0, *R*. Turing hasn't told us what '*P*0' means, but, because '*P*' means "print", it's pretty obvious that this means "print 0 on the currently scanned square".

Note, too, that he hasn't told us which square is currently being scanned! It probably doesn't matter,[17] but it's worth thinking about some of the options. One possibility is that we are scanning the first, or leftmost, square; this is the most likely option and the one that I will assume in what follows. But another possibility is that we are scanning some other square somewhere in the "middle" of the tape. That probably doesn't matter, because Turing only told us that it would compute the sequence '010101…'; he didn't say *where* it would be computed! There is one further possbiility, not very different from the previous one: The tape might not have a "first" square—it might be infinite in both directions! And now we need to consider something that Turing hasn't mentioned: How long is the tape? For all that he has told us, it could be infinitely long. But many mathematicians and philosophers (not to mention engineers!) are not overly fond of actual infinities. So, we can say, instead, that, at any moment, the tape only has a finite number of squares, *but* there is no limit to the number of extra squares that we are allowed to add on at one (or maybe both) ends. (As my former teacher and colleague John Case used to put it, if we run out of squares, we can always go to an office-supply store, buy some extra squares, and staple them onto our tape!)

So, let's now show our initial tape as follows, where the currently scanned square is underlined:

$$\underline{\flat}\ \flat\ \flat\ \flat\ \flat\ \flat\ \flat\ \flat\ \flat\ \flat\ \ldots$$

[17] It probably doesn't matter, because all squares on the tape are blank. If the tape is infinite (or endless) in both directions, then each square is indistinguishable from any other square, at least until something is printed on one square.

Performing the two operations on line *b* converts our initial tape to this one:

$$0\ \underline{\text{þ}}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \ldots$$

and puts us in state *c*. That is, we next execute the instruction on line *c*.

Looking at line *c*, we see that, no matter what symbol is on the current square (it is, in fact, blank), we should simply move right one more square and change our mind to *e*. So now our tape will look like this:

$$0\ \text{þ}\ \underline{\text{þ}}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \ldots$$

Because we are now in state *e*, we look at line *e* of the program, which tells us that, no matter what, if anything, is on the current square, print '1' there, move right again, and go into state f. So, our tape becomes:

$$0\ \text{þ}\ 1\ \underline{\text{þ}}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \ldots$$

Now we are in state f, so, looking at line f, we see that we merely move right once again, yielding:

$$0\ \text{þ}\ 1\ \text{þ}\ \underline{\text{þ}}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \text{þ}\ \ldots$$

And we go back into state *b*. But that starts this cycle all over again; we are indeed in an infinite loop! One more cycle through this turns our tape into:

$$0\ \text{þ}\ 1\ \text{þ}\ 0\ \text{þ}\ 1\ \underline{\text{þ}}\ \text{þ}\ \text{þ}\ \text{þ}\ \ldots$$

Clearly, repeated cycles through this infinitely looping program will yield a tape consisting entirely of the infinite sequence 010101… with blank squares separating each square with a symbol printed on it:

$$0\ \text{þ}\ 1\ \text{þ}\ 0\ \text{þ}\ 1\ \text{þ}\ 0\ \text{þ}\ 1\ \text{þ}\ \ldots$$

8.11.1.2 Section 3, Example I, Paragraph 2

Can this program be written differently?

> If (contrary to the description in §1) we allow the letters L, R to appear more than once in the operations column we can simplify the table considerably. (p. 234.)

In "the description in §1" (p. 231), Turing allowed the machine to "change the square which is being scanned, but only by shifting it *one* place to right or left" (my italics). Now, he is allowing the machine to move *more* than one place to the right or left; this is accomplished by allowing a *sequence* of moves. Here is the modified program:

m-config.	*symbol*	*operations*	*final m-config.*
	None	$P0$	b
b	0	$R, R, P1$	b
	1	$R, R, P0$	b

Note that there is only one *m*-configuration, that is, only one line number; another way to think about this is that this program has only one instruction. Turing would say

that this machine never changes its state of mind. But that one instruction is, of course, more complex than the previous ones. This one is what would now be called a 'case' statement: In case there is no current symbol, print 0; in case the current symbol = 0, move right 2 squares and print 1; and in case the current symbol = 1, move right 2 squares and print 0.

Exercise for the Reader:
I urge you to try to follow this version of the program, both for practice in reading such programs and to convince yourself that it has the same behavior as the first one.

Another interesting exercise is to write a program for a Turing machine that will print the sequence 010101… *without* intervening blank squares.

So, our machine has "compute[d] the sequence 010101…". Or has it? It has certainly written that sequence down. Is that the same thing as "computing" it?

And here is another question: Earlier, I said that 010101… was the binary representation of $\frac{1}{3}$ and the decimal representation of $\frac{1}{99}$. Have we just computed $\frac{1}{3}$ in base 2? Or $\frac{1}{99}$ in base 10?

Further Reading:
For an interesting discussion of this, see (Rescorla, 2013). In Chapter 10, we'll return to some of the issues discussed in Rescorla's paper.

Even if you are inclined to answer 'yes' to the question whether writing is the same as copying, you might be more inclined to answer 'no' to the question whether we have computed $\frac{1}{3}$ in base 2 or $\frac{1}{99}$ in base 10. Although Turing may have a convincing reason (in his §9) to say that computing consists of nothing more than writing down symbols, surely there has to be more to it than that; surely, just writing down symbols is only *part* of computing. The other parts have to do with *which* symbols get written down, *in what order*, and *for what reason*. If I asked you to compute the decimal representation of $\frac{1}{99}$, how would you know that you were supposed to write down 010101…? Surely, *that* is the heart of computation. Or is it?

At this point, however, we should give Turing the benefit of the doubt. After all, *he* did not say that we were going to compute $\frac{1}{99}$, only that we were going to "compute" 010101…, and, after all, "computing" that sequence really just *is* writing it down; it's a trivial, or basic, or elementary, or primitive computation (choose your favorite adjective). Moreover, arguably, Turing only showed us this trivial example so that we could clearly see the *format* of his Turing-machine programs before getting a more complex example.

Before turning to such a more complex program, let's consider the syntax (specifically, the grammatical structure) of these programs a bit more. (For more on what 'syntax' means, see Chapter 22 and (Rapaport, 2017b).) Each line of the program has the following, general form:

$$q_B \; S \; O \; q_E$$

where:

1. q_B is an initial (or **B**eginning) *m*-configuration (a line number)
2. S is the symbol on the currently scanned square (possibly a blank)
3. O is an operation (or a sequence of operations) to be performed (where the operations are Px, E, L, R, and where x is any legally allowed symbol)[18]
4. q_E is a final (or **E**nding) *m*-configuration.

And the semantics (that is, the meaning or interpretation) of this program line is:

> **if** the Turing machine is in *m*-configuration q_B, **and**
> **if either** the current input = S **or** no input is specified,
> **then**
> **begin**
> do the sequence of operations O;
>
> {where each operation is *either*:
>
> - *P*rint x on the current square,
> (where printing x overwrites whatever is currently printed on the square), *or*
> - *E*rase the symbol that is on the current square,
> (where erasing results in a blank square, even if the square is already blank), *or*
> - move *L*eft one square, *or*
> - move *R*ight one square}
>
> go to *m*-configuration q_E
> **end**

8.11.2 Section 3, Example II

8.11.2.1 Section 3, Example II, Paragraph 1

We now come to "a slightly more difficult example":

> As a slightly more difficult example we can construct a machine to compute the sequence 00101101110111011111.... (p. 234.)

First, note that the sequence to be computed consists of the subsequences

$$0,\ 1,\ 11,\ 111,\ 1111,\ 11111,\ \ldots$$

That is, it is a sequence beginning with '0', followed by the numbers 1, 2, 3, 4, 5, ... written in base 1 (that is, as "tally strokes")—with each term separated by a '0'.

But this seems very disappointing! It seems that this "more difficult" computation is still just writing down some symbols without "computing" anything. Perhaps. But note that what is being written down (or "computed") here are the natural numbers.

[18] In our first program, the only symbols were '0' and '1'; we will see others in subsequent examples.

This program will begin counting, starting with 0, then the successor of 0, the successor of that, and so on. But, as we saw in §7.6.2, the successor function is one of the basic recursive functions, that is, one of the basic computable functions.

Being able to (merely!) write down the *successor* of any number, being able to (merely!) write down the *predecessor* of any non-0 number, and being able to *find* a given term in a sequence of numbers are the only basic recursive (or computable) functions. Turing's "slightly more difficult example" will show us how to compute the first of these. Devising a Turing machine program for computing the predecessor of the natural number n should simply require us to take a numeral represented as a sequence of n occurrences of '1' and erase the last one. Devising a Turing machine program for computing the jth term in a sequence of k symbols should simply require us to move a certain number of squares in some direction to find the term (or, say, the first square of a sequence of squares that represents the term, if the term is complex enough to have to be represented by a sequence of squares).

And any other other recursive function can be constructed from these basic functions by generalized composition (sequencing), conditional definition (selection), and while-recursion (repetition), which are just "control structures" for how to find a path (so to speak) through a Turing-machine program—that is, ways to organize the sequence of m-configurations that the Turing machine should go through.

So, it looks as if **computation really is nothing more than writing things down, moving around (on a tape), and doing so in an order that will produce a desired result!** As historian Michael Mahoney suggested, the shortest description of Turing's accomplishment might be that Turing

> showed that any computation can be described in terms of a machine shifting among a finite number of states in response to a sequence of symbols read and written one at a time on a potentially infinite tape. —(Mahoney, 2011, p. 79)

We'll return to this idea in §9.6.

Let's now look at this "slightly more difficult" program:

> The machine is to be capable of five m-configurations, viz., "o", "q", "p", "f", "b" and of printing "ə", "x", "0", "1".
> (p. 234, substituting italics for German Fraktur letters)

The first two printable symbols are going to be used only for bookkeeping purposes.[19] So, once again, Turing is really restricting himself to binary notation for the important information.

Continuing:

> The first three *symbols* on the tape will be "əə0"; the other *figures* follow on alternate squares. (p. 234, my italics.)

It may sound as if Turing is saying that the tape comes with some pre-printed information. But, when we see the program, we will see that, in fact, the first instruction has us print 'əə0' on the first three squares before beginning the "real" computation. Had

[19] The inverted 'e' is called a 'schwa'; it is used in phonetics to represent the sound "uh", as in 'but'. Turing uses it merely as a bookkeeping symbol with no meaning.

the tape come with pre-printed information, perhaps it could have been considered as "innate" knowledge,[20] though a less cognitive description could simply have been that the manufacturer of the tape had simplified our life knowing that the first thing that the program does to a completely blank tape is to print 'əə0' on the first three squares before beginning the 'real' computation. Because that only has to be done once, it might have been simpler to consider it as pre-printed on the tape.

Note that Turing calls these 'symbols' in the first clause, and then talks about 'figures' in the second clause. Figures, you may recall from §8.10.2, are the numerals '0' and '1'. So, Turing seems to be saying that all subsequent occurrences of '0' and '1' will occur on "alternate squares". What happens on the other squares? He tells us:

> On the intermediate squares we never print anything but "*x*". These letters serve to "keep the place" for us and are erased when we have finished with them. We also arrange that in the sequence of figures on alternate squares there shall be no blanks. (p. 234.)

So, it sounds as if the final tape will begin with 'əə0'; during the computation, subsequent squares will have '0' or '1' interspersed with '*x*'; and at the end of the computatation, those subsequent squares will only have '0' or '1', and no blanks. Of course, at the end, we could go back and erase the initial occurrences of 'ə', so that there would only be "figures" and no other symbols.

Here is the program:

Configuraton		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
b		Pə$,R,P$ə$,R,P0,R,R,P0,L,L$	*o*
o	1	*R*, *Px*, *L*, *L*, *L*	*o*
	0		*q*
q	Any (0 or 1)	R,R	*q*
	None	$P1,L$	*p*
p	*x*	E,R	*q*
	ə	R	*f*
	None	L,L	*p*
f	Any	R,R	*f*
	None	$P0,L,L$	*o*

[20] That is, knowledge that it was "born" with (or, to use another metaphor, knowledge that is "hardwired"). For more on innate knowledge, see (Samet and Zaitchik, 2017).

I think it will be helpful to restate this program in a more readable format:

```
b  begin
       print 'əə0' on the first 3 squares;
       P0 on the 5th square;
       move left to the 3rd square (which has '0' on it);
       go to line o
   end

o  if current symbol = 1
       then
          begin
             move right;
             Px;
             move left 3 squares;
             go to line o {that is, stay at o}
          end
       else if current symbol = 0
             then go to line q

q  if current symbol = 0 or current symbol = 1
       then
          begin
             move right 2 squares;
             go to line q
          end
       else if current square is blank
             then
                begin
                   P1;
                   move left;
                   go to line p
                end
```

[continued on next page]

```
p  if current symbol = x
       then
         begin
           erase the x;
           move right;
           go to line q
         end
       else if current symbol = ə
            then
              begin
                move right;
                go to line f
              end
       else if current square is blank
            then
              begin
                move left 2 squares;
                go to line p
              end

f  if current square is not blank
       then
         begin
           move right 2 squares;
           go to line f
         end
       else
         begin
           P0;
           move left 2 squares;
           go to line o
         end
```

Note that no line of the program ends with the machine changing its state of mind to *m*-configuration *b*. So that line of the program, which is the one that initializes the tape with 'əə0' on the first 3 squares, is only executed once. Note also that, whenever an instruction ends with a command to stay in the same *m*-configuration (that is, to go to that very same line), we are in a loop. A structured version of the program would use a **while...do** control structure, instead.

There are some odd things to consider in lines o, q, p: What happens if the machine is in state *o* but the current symbol is not a "figure"? What happens in state *q* if the current symbol is 'ə' or '*x*'? And what happens in state *p* if the current symbol *is* a "figure"? Turing doesn't specify what should happen in these cases. One possibility is that he has already determined that none of these cases *could* occur. Still, modern software engineering practice would recommend that an error message be printed out in those cases. In general, in a computer program, when a situation occurs for which the

program does not specify what should happen, anything is legally allowed to happen, and there is no way to predict what will happen; this is sometimes expressed in the slogan, "garbage in, garbage out".

8.11.2.2 Section 3, Example II, Paragraph 2

Turing goes on "to illustrate the working of this machine" with "a table ... of the first few complete configurations" (p. 234.) Recall that a "complete configuration" consists of information about which square is currently being scanned, the sequence of all symbols on the tape, and the line number of the instruction currently being executed. Rather than use Turing's format, I will continue to use the format that I used for Example I, adding the line number at the beginning, using underscoring to indicate the currently scanned square, and assuming that any squares not shown are blank; any blank square that is between two non-blank squares (if there are any) will be indicated by our symbol for a blank that has been made visible: ♭. You are urged to compare my trace of this program with Turing's.

So, we begin with a blank tape. What is the machine's initial state of mind, its initial *m*-configuration? Turing has forgotten to tell us! But it is fairly obvious that *b* is the initial *m*-configuration, and, presumably, we are scanning the leftmost square (or, if the tape is infinite in both directions, then we are scanning any arbitrary square), and, of course, all squares are blank:

$$b : \underline{♭}, ♭, \ldots$$

The initial instruction tells us to print ə, move right, print another ə, move right again, print 0, move right 2 more squares, print another 0, move 2 squares back to the left, and go into state *o*. After doing this sequence of primitive operations, our complete configuration looks like this:

$$o : ə, ə, \underline{0}, ♭, 0, ♭, \ldots$$

Digression on Notation:
To help you in reading Turing's paper, my notation for the initial situation should be compared with his. Here is his:

$$\begin{array}{c} : \\ b \end{array}$$

He has an invisible blank, followed by a colon, with the *m*-configuration '*b*' underneath the blank, marking the currently scanned square.

Instead, I have '*b*:' preceding a sequence of (visible) blanks, the first one of which is marked as being the scanned square.

Turing then shows the second configuration:

$$\begin{array}{cccccc} ə & ə & 0 & & 0 & : \\ & & o & & & \end{array}$$

Turing has two occurrences of 'ə' followed by two '0's that are separated by an (invisible) blank, with the *m*-configuration '*o*' underneath the currently scanned square (which contains the first '0'), followed by a colon to mark the end of this complete configuration.

Instead, I have '*o*:' preceding a sequence consisting of the two occurrences of 'ə', followed by a '0' that is marked as being the scanned square, followed by a (visible) blank, followed by the second '0'.

We are now in *m*-configuration *o*, and the currently scanned square contains '0', so the second case (that is, the bottom row) of this second instruction tells us merely to go into state *q*. The "operations" column is left empty, so there is no operation to perform. It is worth noting that, although there does not always have to be an operation to perform, *there does always have to be a final state to go into, that is, a next instruction to perform.* So, the tape looks exactly as it did before, except that the machine is now in state *q*:

$$q : ə, ə, \underline{0}, ♭, 0, ♭, \ldots$$

Because the machine is now in state *q* and still scanning a '0', the first case (that is, the top row) of this third instruction tells us to move two squares to the right but to stay in state *q*. So the tape now looks like this:

$$q : ə, ə, 0, ♭, \underline{0}, ♭, \ldots$$

Because the machine is still in state *q* and still scanning a '0' (although the currently scanned *square* is different), we perform the same (third) instruction, moving two more squares to the right and staying in state *q*:

$$q : ə, ə, 0, ♭, 0, ♭, \underline{♭}, \ldots$$

The machine is still in state *q*, but now there is no scanned symbol, so the second case (bottom line) of the third instruction is executed, resulting in a '1' being printed on the current square, and the machine moves left, going into state *p*.

Whenever the machine is in state *p* and scanning a blank (as it is now), the third case (last line) of the fourth instruction is executed, so the machine moves two squares to the left and stays in state *p*:

$$p : ə, ə, 0, \underline{♭}, 0, ♭, 1, \ldots$$

Now the machine is in state p scanning a blank, so the same instruction is executed: It moves two more squares to the left and continues in state p:

$$p : ə, \underline{ə}, 0, ♭, 0, ♭, 1, \ldots$$

But now it is the second case (middle line) of the fourth instruction that is executed, so the machine moves right and goes into state f:

$$f : ə, ə, \underline{0}, ♭, 0, ♭, 1, \ldots$$

When in state f scanning any symbol (but not a blank), the machine moves two squares to the right, staying in f:

$$f : ə, ə, 0, ♭, \underline{0}, ♭, 1, \ldots$$

Again, it moves two squares to the right, staying in f:

$$f : ə, ə, 0, ♭, 0, ♭, \underline{1}, \ldots$$

And again:

$$f : ə, ə, 0, ♭, 0, ♭, 1, ♭, \underline{♭}, \ldots$$

But now it executes the second case of the last instruction, printing '0', moving two squares to the left, and returning to state o:

$$o : ə, ə, 0, ♭, 0, ♭, \underline{1}, ♭, 0, \ldots$$

Now, for the first time, the machine executes the first case of the second instruction, moving right, printing 'x', moving three squares to the left, but staying in o:

$$o : ə, ə, 0, ♭, \underline{0}, ♭, 1, x, 0, \ldots$$

At this point, you will be forgiven if you have gotten lost in the "woods", having paid attention only to the individual "trees" and not seeing the bigger picture.[21] Recall that we are trying to count: to produce the sequence 0, 1, 11, 111, ... with '0's between each term:

$$0\ 0\ 1\ 0\ 11\ 0\ 111\ 0 \ldots$$

We started with a blank tape:

$$♭♭♭ \ldots$$

and we now have a tape that looks like this:

$$əə0♭0♭1x0♭ \ldots$$

Clearly, we are going to have to continue tracing the program before we can see the pattern that we are expecting; Turing, however, ends his tracing at this point. But we shall continue; however, I will only show the complete configurations without spelling

[21] My apologies for the mixed metaphor.

out the instructions (doing that is left to the reader). Here goes, continuing from where we left off:

o :	ə	ə	0	♭	$\underline{0}$	♭	1	x	0	…					
q :	ə	ə	0	♭	$\underline{0}$	♭	1	x	0	…					
q :	ə	ə	0	♭	0	♭	$\underline{1}$	x	0	…					
q :	ə	ə	0	♭	0	♭	1	x	$\underline{0}$	…					
q :	ə	ə	0	♭	0	♭	1	x	0	♭	$\underline{♭}$	…			
p :	ə	ə	0	♭	0	♭	1	x	0	$\underline{♭}$	1	…			
p :	ə	ə	0	♭	0	♭	1	$\underline{x}$	0	♭	1	…			
q :	ə	ə	0	♭	0	♭	1	♭	$\underline{0}$	♭	1	…			
q :	ə	ə	0	♭	0	♭	1	♭	0	♭	$\underline{1}$	…			
q :	ə	ə	0	♭	0	♭	1	♭	0	♭	1	♭	$\underline{♭}$	♭…	
p :	ə	ə	0	♭	0	♭	1	♭	0	♭	1	$\underline{♭}$	1	♭…	

Hopefully, now you can see the desired pattern beginning to emerge. The occurrences of 'x' get erased, and what's left is the desired sequence, but with blank squares between each term *and* with two leading occurrences of 'ə'. You can see from the program that there is no instruction that will erase those 'ə's; the only instructions that pay any attention to a 'ə' are (1) the second case of m-configuration p, which only tells the machine to move right and to go into state f, and (2) the first case of m-configuration f, which, when scanning any symbol, simply moves two squares to the right (but, in fact, that configuration will never occur!).

In the third paragraph, Turing makes some remarks about various notation conventions that he has adopted, but we will ignore these, because we are almost finished with our slow reading. I do want to point out some other highlights, however.

8.12 Section 4: "Abbreviated Tables"

In this section, Turing introduces some concepts that are central to programming and software engineering.

> There are certain types of process used by nearly all machines, and these, in some machines, are used in many connections. These processes include copying down sequences of symbols, comparing sequences, erasing all symbols of a given form, etc. (p. 235.)

In other words, certain sequences of instructions occur repeatedly in different programs and can be thought of as being single "processess": copying, comparing, erasing, etc.

Turing continues:

> Where such processes are concerned we can abbreviate the tables for the m-configurations considerably by the use of "skeleton tables". (p. 235.)

The idea is that skeleton tables are descriptions of more complex sequences of instructions that are given a single name. This is the idea behind "subroutines" (or "named procedures") and "macros" in modern computer programming. (Recall our discussion

of this in §7.5.3.) If you have a sequence of instructions that accomplishes what might better be thought of as a single task (for example, copying a sequence of symbols), and if you have to repeat this sequence many times throughout the program, it is more convenient (for the human writer or reader of the program!) to write this sequence down only once, give it a name, and then refer to it by that name whenever it is needed.[22] There is one small complication: Each time that this named abbreviation is needed, it might require that parts of it refer to squares or symbols on the tape that will vary depending on the current configuration, so the one occurrence of this named sequence in the program might need to have variables in it:

> In skeleton tables there appear capital German letters and small Greek letters. These are of the nature of "variables". By replacing each capital German letter throughout by an *m*-configuration and each small Greek letter by a symbol, we obtain the table for an *m*-configuration. (pp. 235–236.)

Of course, whether one uses capital German letters, small Greek letters, or something more legible or easier to type is an unimportant, implementation detail. The important point is this:

> The skeleton tables are to be regarded as nothing but abbreviations: they are not essential. (p. 236.)

8.13 Section 5: "Enumeration of Computable Sequences"

Another highlight of Turing's paper that is worth pointing out occurs in his §5: a way to convert every program for a Turing machine into a number. Let me be a bit more precise about this before seeing how Turing does it.

First, it is important to note that, for Turing, there really is no difference between one of his *a*-machines (that is, a Turing machine) and the *program for* it. Turing machines are "hardwired" to perform exactly one task, as specified in the program (the "table", or "machine table") for it. So, converting a program to a number is the same as converting a Turing machine to a number.

Second, "converting to a number"—that is, assigning a number to an object—really means that you are *counting*. So, in this section, Turing shows that you can count Turing machines by assigning a number to each one.

Third, if you can count Turing machines, then you can only have a countable number of them. But there are *un*countably many real numbers, so there will be some real numbers that are not computable!

Here is how Turing counts Turing machines. First (using the lower-case Greek letter "gamma", γ):

> A computable sequence γ is determined by a description of a machine which computes γ. Thus the sequence 001011011101111... is determined by the table on

[22] "As Alfred North Whitehead wrote, 'Civilisation advances by extending the number of important operations which we can perform without thinking about them.' " (Brian Hayes 2014b, p. 22).

> p. 234, and, in fact, any computable sequence is capable of being described in terms of such a table. (p. 239.)

"A description of a machine" is one of the tables such as those we have been looking at; that is, it is a computer program for a Turing machine.

But, as we have also seen, it is possible to write these tables in various ways. So, before we can count them, we need to make sure that we don't count any twice because we have confused two different ways of writing the same table with being two different tables. Consequently:

> It will be useful to put these tables into a kind of standard form. (p. 239.)

The first step in doing this is to be consistent about the number of separate operations that can appear in the "operations" column of one of these tables. Note that in the two programs we have looked at, we have seen examples in which there were as few as 0 operations and as many as 10 (not to mention the variations possible with skeleton tables). So:

> In the first place let us suppose that the table is given in the same form as the first table, for example, I on p. 233. [See our §8.11.1, above.] That is to say, that the entry in the operations column is always of one of the forms $E : E,R : E,L : Pa : Pa,R : Pa,L : R : L$: or no entry at all. The table can always be put into this form by introducing more *m*-configurations. (p. 239, my interpolation.)

In other words, the operation in the operations column will be exactly one of:

erase
erase and then move right
erase and then move left
print symbol a
print a and then move right
print a and then move left

> (where 'a' is a variable ranging over all the possible symbols in a given program)

move right
move left
do nothing

"Introducing more *m*-configurations" merely means that a single instruction such as:

$$b \;\; 0 \;\; P1,R,P0,L \;\; f$$

can be replaced by two instructions:

$$\begin{array}{cccc} b & 0 & P1,R & f_1 \\ f_1 & & P0,L & f \end{array}$$

where 'f_1' is a new *m*-configuration not appearing in the original program. Put otherwise, a *single* instruction consisting of a *sequence* of operations can be replaced by a

sequence of instructions each consisting of a *single* operation. (For convenience, presumably, Turing allows *pairs* of operations, where the first member of the pair is either E or P and the second is either R or L. So a single instruction consisting of a sequence of (pairs of) operations can be replaced by a sequence of instructions each consisting of a single operation or a single such pair.)

Numbering begins as follows:

> Now let us give numbers to the m-configurations, calling them $q_1, \ldots, q_R$ as in §1. The initial m-configuration is always to be called q_1. (p. 239.)

So, each m-configuration's number is written as a subscript on the letter 'q'.

The numbering continues:

> We also give numbers to the symbols $S_1, \ldots, S_m$ and, in particular, blank = $S_0, 0 = S_1, 1 = S_2$. (pp. 239–240.)

So, each symbol's number is written as a susbscript on the letter 'S'.

Note, finally, that Turing singles out three symbols for special treatment, namely, '0', '1', and what I have been writing as ♭. (So, Turing is finally making the blank visible.)

At this point, we have the beginnings of our "standard forms", sometimes called 'normal' forms (which Turing labels N_1, N_2, N_3):

> The lines of the table are now [one] of [the following three] form[s]

m-config.	*Symbol*	*Operations*	*Final m-config.*	
q_i	S_j	PS_k, L	q_m	(N_1)
q_i	S_j	PS_k, R	q_m	(N_2)
q_i	S_j	PS_k	q_m	(N_3)

> (p. 240, my interpolations.)

So, we have three "normal forms":

N_1 m-configuration q_i = **if** currently scanned symbol is S_j,
then
begin
print symbol S_k;
move left;
go to q_m
end

N_2 m-configuration q_i = **if** currently scanned symbol is S_j,
then
begin
print symbol S_k;
move right;
go to q_m
end

N_3 m-configuration q_i = **if** currently scanned symbol is S_j,
then
begin
print symbol S_k;
go to q_m
end

As Turing notes in the following passage (which I will not quote but merely summarize), erasing (E) is now going to be interpreted as printing a blank (PS_0), and a line in which the currently scanned symbol is S_j and the operation is merely to move right or left is now going to be interpreted as overprinting the very same symbol (PS_j) and then moving. So, all instructions require printing something: either a visible symbol or a blank symbol, and then either moving or not moving. As Turing notes:

> In this way we reduce each line of the table to a line of one of the forms $(N_1), (N_2), (N_3)$. (p. 240.)

Turing simplifies even further, eliminating the 'print' command and retaining only the symbol to be printed. After all, if all commands involve printing something, you don't need to write down 'P'; you only need to write down what you're printing. So each instruction can be simplified to a 5-tuple consisting of the initial m-configuration, the currently scanned symbol (and there will always be one, even if the "symbol" is blank, because the blank has been replaced by 'S_0'), the symbol to be printed (again, there will always be one, even if it's the blank), and the final m-configuration:

> From each line of form (N_1) let us form an expression $q_iS_jS_kLq_m$; from each line of form (N_2) we form an expression $q_iS_jS_kRq_m$; and from each line of form (N_3) we form an expression $q_iS_jS_kNq_m$. (p. 240.)

Presumably, N means something like "no move". A slightly more general interpretation is that, not only do we always print something (even if it's a blank), but we also always move somewhere, except that sometimes we "move" to our current location. This standardization is consistent with our earlier observation (in §7.5.2, above) that the only two verbs that are needed are 'print(symbol)' and 'move(location)'.

Next:

> Let us write down all expressions so formed from the table for the machine and separate them by semi-colons. In this way we obtain a complete description of the machine. (p. 240.)

Turing's point here is that the set of instructions can be replaced by a single string of 5-tuples separated by semi-colons. There are two observations to make. First, because the machine table is a *set* of instructions, there could (in principle) be several different strings (that is, descriptions) for each such set, because strings are *sequences* of symbols. Second, Turing has here introduced the now-standard notion of using a semi-colon to separate lines of a program; however, this is not quite the same thing as the convention of using a semi-colon to signal *sequencing*, because the instructions of a Turing-machine program are not an ordered sequence of instructions (even if, whenever they are written down, they have to be written down in some order).

So, Turing has developed a standard encoding of the lines of a program: an *m*-configuration encoded as q_i (forget about b, f, etc.), a pair of symbols encoded as S_j, S_k (the first being the scanned input, the second being the printed output; again, forget about things like '0', '1', 'x', etc.), a symbol (either L, R, or N) encoding the location to be moved to, and another *m*-configuration encoded as q_m. Next, he gives an encoding of these standardized codes:

> In this description we shall replace q_i by the letter "D" followed by the letter "A" repeated i times, and S_j by "D" followed by "C" repeated j times. (p. 240.)

Before seeing *why* he does this, let's make sure we understand *what* he is doing. The only allowable *m*-configuration symbols in an instruction are: $q_1, \ldots, q_l$, for some l that is the number of the final instruction. What really matters is that each instruction can be assumed to begin and end with an *m*-configuration symbol, and the only thing that really matters is which one it is, which can be determined by the subscript on q. In this new encoding, "D" simply marks the beginning of an item in the 5-tuple, and the i occurrences of letter 'A' encode the subscript. Similarly, the only allowable symbols are: $S_1, \ldots, S_n$, for some n that is the number of the last symbol in the alphabet of symbols. What really matters is that, in each instruction, the second and third items in the 5-tuple can be assumed to be symbols (including a visible blank!), and the only thing that really matters is which ones they are, which can be determined by the subscript on S. In our new encoding, "D" again marks the beginning the next item in the 5-tuple, and the j occurrences of 'C' encode the subscript.

Turing then explains that:

> This new description of the machine may be called the *standard description* (S.D). It is made up entirely from the letters "A", "C", "D", "L", "R", "N", and from ";". (p. 240.)

So, for example, this 2-line program:

$$q_3 S_1 S_4 R q_5$$

$$q_5 S_4 S_0 L q_5$$

will be encoded by an S.D consisting of this 38-character string:

$$DAAADCDCCCCRDAAAAA; DAAAAADCCCCDLDAAAAA$$

The next step in numbering consists in replacing these symbols by numerals:

> If finally we replace "*A*" by "1", "*C*" by "2", "*D*" by "3", "*L*" by "4", "*R*" by "5", "*N*" by "6", ";" by "7" we shall have a description of the machine in the form of an arabic numeral. The integer represented by this numeral may be called a *description number* (D.N) of the machine.
> (p. 240; Turing did not capitalize 'arabic'.)

Just as Gödel numbering is one way to create a number corresponding to a string, "Turing numbering" is another. The D.N of the machine in our previous example is this numeral:

311132322225311111731111132222343111111

which, written in the usual notation with commas, is:

31,113,232,222,531,111,173,111,113,222,234,311,111

or, in words, 31 undecillion, 113 decillion, 232 nonillion, 222 octillion, 531 septillion, 111 sextillion, 173 quintillion, 111 quadrillion, 113 trillion, 222 billion, 234 million, 311 thousand, one hundred eleven. That is the "Turing number" of our 2-line program!

Turing observes that:

> The D.N determine the S.D and the structure of the machine uniquely. The machine whose D.N is n may be described as $\mathcal{M}(n)$. (pp. 240–242.)

Clearly, given a D.N, it is trivial to decode it back into an S.D in only one way. Equally clearly (and almost as trivially), the S.D can be decoded back into a program in only one way. Hence, "the structure of the machine" encoded by the D.N is "determine[d] ... uniquely" by the D.N. However, because of the possibility of writing a program for a machine in different ways (permuting the order of the instructions), two different D.Ns might correspond to the same machine, so there will, in general be distinct numbers n, m (that is, $n \neq m$) such that $\mathcal{M}(n) = \mathcal{M}(m)$. That is, "the" Turing machine whose number = n might be the same machine as the one whose number = m; a given Turing machine might have two different numbers. Alternatively, we could consider that we have here two different machines that have exactly the same input-output behavior and that execute exactly the same algorithm. Even in that latter case, where we have more machines than in the former case, the machines are enumerable; that is, we can count them.

Can we also count the sequences that they compute? Yes; Turing explains why (with Turing's explanation in italics and my comments interpolated in brackets):

> *To each computable sequence* [that is, to each sequence that is printed to the tape of a Turing machine] *there corresponds at least one description number* [we have just seen why there might be more than one], *while to no description number does there correspond more than one computable sequence* [that is, each machine prints out exactly one sequence; there is no way a given machine could print out two different sequences, because the behavior of each machine is completely determined by its program, and no program allows for any arbitrary, free, or random "choices"

> that could vary what gets printed on the tape]. *The computable sequences and numbers* [remember: every sequence corresponds to a unique number][23] *are therefore enumerable* [that is, countable]. (p. 241, italics and interpolations added.)

Next, on p. 241, Turing shows how to compute the D.N of program I (the one that printed the sequence $\overline{01}$). And he gives a D.N without telling the reader what program corresponds to it.

Finally, he alludes to the Halting Problem:

> A number which is a description number of a circle-free machine will be called a *satisfactory* number. In §8 it is shown that there can be no general process for determining whether a given number is satisfactory or not. (p. 241.)

A "satisfactory" number is the number of a circle-free Turing machine, that is, a Turing machine that never halts and that does compute the infinite decimal representation of a real number. That is, a "satisfactory" number is the number of a Turing machine for a *computable* number. So, in Turing's §8, he is going to show that there is "no general process"—that is, no Turing machine that can decide (by computing)—"whether a given number is satisfactory", that is, whether a given number is the number of a *circle-free* Turing machine. It is easy to determine if a given number is the number of a Turing machine: Just decode it, and see if the result is a syntactically correct, Turing-machine program. But, even if it is a syntactically correct, Turing-machine program, there will be no way to decide (that is, to compute) whether it halts or not. (Remember: For Turing, *halting* is bad, *not* halting is *good*; in modern presentations of computing theory, halting is good, not halting is (generally considered to be)[24] *bad*.)

8.14 Section 6: "The Universal Computing Machine"

> A man provided with paper, pencil, and rubber [eraser], and subject to strict discipline, is in effect a universal machine. —Turing (1948, p. 416)

> In fact we have been universal computers ever since the age we could follow instructions. —Bernhardt (2016, p. 12; see also p. 94)

Although Turing's §6 is at least one of, if not *the* most important section of Turing's paper, we will only look at it briefly in this chapter. As before, you are encouraged to consult (Petzold, 2008) for aid in reading it in detail.

Turing begins with this claim:

> It is possible to invent a single machine which can be used to compute any computable sequence. (p. 241.)

So, instead of needing as many Turing machines as there are computable numbers, we only need one. Recall that our first "great insight" was that all information can

[23] Although, because of a curiosity of decimal representation, some numbers correspond to more than one sequence. The classic example is that $1 = 1.\overline{0} = 0.\overline{9}$.

[24] But see Chapter 11!

be represented using only '0' and '1' (§7.5.1, above). That means that all information that we would want to compute with—not only numbers, but language, images, sounds, etc.—can be represented by a sequence of '0's and '1's, that is, as a computable number (in binary notation). So, Turing's claim is that there is a *single* machine that can be used to compute *anything* that is computable.

Most of you own one. Indeed, most of you own several, some of which are small enough to be carried in your pocket! They are made by Apple, Dell, et al.; they come in the form of laptop computers, smartphones, etc. They are general-purpose, programmable computers.

> If this [single] machine $\mathcal{U}$ is supplied with a tape on the beginning of which is written the S.D of some computing machine $\mathcal{M}$, then $\mathcal{U}$ will compute the same sequence as $\mathcal{M}$. (pp. 241–242, my interpolation.)

Your laptop or smartphone is one of these $\mathcal{U}$s. A program or "app" that you download to it is an S.D (written in a different programming language than Turing's) of a Turing machine that does only what that program or "app" does. The computer or smartphone that runs that program or "app", however, can also run other programs, in fact, many of them. That's what makes it "universal":

> But to do all the things a smartphone can do without buying one, ... [a] consumer would need to buy the following:
>
> A cellphone ....
> A mobile e-mail reader ....
> A music player ....
> A point-and-shoot camera ....
> A camcorder ....
> A GPS unit ....
> A portable DVD player ....
> A voice recorder ....
> A watch ....
> A calculator ....
>
> Total cost: $1,999
>
> *In a smartphone, all those devices are reduced to software.*
> —(Grobart, 2011, my italics)

A Turing machine is to a universal Turing machine as a music box is to a player piano: A music box (or Turing machine) can only play (or execute) the tune (or program) that is hardwired into it. Player pianos (or universal Turing machines) can play (or execute) any tune (or program) that is encoded on its piano-roll (or tape). Here's a related question: "Why is a player piano *not* a computer?" (Kanat-Alexander, 2008). Alternatively, when is a universal Turing machine a player piano? The "instructions" on the piano roll cause certain keys to be played; you can think of each key as a Turing-machine tape cell, with "play" or "don't play" analogous to "print-one" or "print-zero". One difference is that a player piano would be a *parallel* machine, because you can play chords.

Further Reading: For discussion of the music-box analogy, see (Sloman, 2002).

How does Turing's universal computer work? Pretty much the same way that a modern computer works: A program (an "app") is stored somewhere in the computer's memory; similarly, the S.D of a Turing machine is written at the beginning of the universal machine's tape. The operating system of the computer fetches (that is, reads) an instruction and executes it (that is, "simulates its behavior" (Dewdney, 1989, p. 315)), then repeats this "fetch-execute" cycle until there is no next instruction; similarly, the single program for the universal machine fetches the first instruction on its tape, executes it, then repeats this cycle until there is no next instruction on its tape. The details of how it does that are fascinating, but beyond our present scope.

However, here is one way to think about this: Suppose that we have *two* tapes. Tape 1 will be the one we have been discussing so far, containing input (the symbols being scanned) and output (the symbols being printed). Tape 2 will contain the computer's program, with each square representing a "state of mind". The computer can be thought of as starting in a square on Tape 2, executing the instruction in that square (by reading from, and writing to, a square on Tape 1 and then moving to a(nother) square on Tape 1), and then moving to a(nother) square on Tape 2, and repeating this "fetch-execute" loop. In reality, Turing machines only have one tape, and the instructions are not written anywhere; rather, they are "hardwired" into the Turing machine. Any written verion of them is (merely) a description of the Turing machine's behavior (or of its "wiring diagram"). But, if we encode Tape 2 on a portion of Tape 1, then we have a "stored-program"—or universal—computer.

Further Reading:
Dewdney 1989, Chs. 1, 28, 48; Petzold 2008, and Bernhardt 2016, Ch. 6 cover this in detail. Another presentation, using the notion of a register machine, can be found in Dennett 2013a, pp. 126–128. Incidentally, it can be proved that any two-tape Turing machine is equivalent to a one-tape Turing machine; see (Dewdney, 1989, Ch. 28).

Universal Turing machines running software *S* often "evolve" into Turing machines that only execute *S*:

> [S]oftware innovations lead ... the way and hardware redesigns follow ..., once the software versions have been proven to work. If you compare today's computer chips with their ancestors of fifty years ago, you will see many innovations that were first designed as software systems, as *simulations* of new computers running on existing hardware computers. Once their virtues were proven and their defects eliminated or minimized, they could serve as specifications for making new processing chips, much faster versions of the simulations. ... [B]ehavioral competences were first explored in the form of programs running on general-purpose computers [that is, on universal Turing machines] ... and then ... incorporated into "dedicated" hardware [that is, Turing machines].
> —(Dennett, 2017, p. 256)

Further Reading:
Lammens 1990 is a Common Lisp implementation of Turing's universal program as specified in Davis and Weyuker 1983. Cooper 2012 "examine[s] challenges to ... [the] continued primacy [of universal Turing machines] as a model for computation in daily practice and in the wider universe." Davis 2012, p. 147 notes that "Turing's universal machine showed that the distinctness of these three catgories [machine, program, and data] is an illusion."

8.15 The Rest of Turing's Paper

Sections 1–5 of Turing's paper cover the nature of computation, defining it precisely, and stating what is now called "Turing's (computability) thesis". Sections 6 and 7 of Turing's paper cover the universal machine. Section 8 covers the Halting Problem.

We have already examined Section 9 in detail; that was the section in which Turing analyzed how humans compute and then designed a computer program that would do the same thing.

Section 10 shows how it can be that many numbers that one might *think* are *not* computable are, in fact, computable. Section 11 proves that Hilbert's *Entscheidungsproblem* "can have no solution" (p. 259). And the Appendix proves that Turing's notion of computation is logically equivalent to Church's.

Except for modern developments and some engineering-oriented aspects of CS, one could create an undergraduate degree program in CS based solely on this one paper that Turing wrote in 1936!

Further Reading:
Turing's last paper before his death—"Solvable and Unsolvable Problems" (Turing, 1954)—is a fascinating version of Turing 1936 aimed at a(n intelligent) general audience. In this article, he shows that many, if not most, puzzles can be put into a "kind of 'normal form' or 'standard form' " (p. 587) called a "substitution type of puzzle" (p. 588), which turns out to be essentially the notion of a formal system or a Turing machine. He states a kind of "Turing's Thesis" for such puzzles, noting that such a ...

> ... statement is moreover one which one does not attempt to prove. Propaganda is more appropriate to it than proof, for its status is something between a theorem and a definition. In so far as we know a priori what is a puzzle and what is not, the statement is a theorem. In so far as we do not know what puzzles are, the statement is a definition which tells us something about what they are. (p. 588).

In this article, he also proves a version of the Halting Problem, in the sense that he proves "that there cannot be any systematic procedure for determining whether a puzzle be solvable or not" (p. 590).

8.16 Further Sources of Information

1. **Turing's Writings:**

   (a) (Copeland, 2004b) is an anthology of Turing's papers (critically reviewed in (Hodges, 2006)).

   (b) Jack Copeland's "AlanTuring.net Reference Articles on Turing", http://www.alanturing.net/turing_archive/pages/Reference%20Articles/referencearticlesindex.html is a website that contains a variety of interesting papers on various aspects of Turing's work, most written by Copeland, a well-respected contemporary philosopher, including essays on the Church-Turing Thesis: http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html and Turing machines: http://www.alanturing.net/turing_archive/pages/Reference%20Articles/What%20is%20a%20Turing%20Machine.html

   (c) (Cooper and van Leeuwen, 2013) is a large, "coffee table"-sized anthology of Turing's writings with commentaries by, among many others, Rodney Brooks, Gregory Chaitin, B. Jack Copeland, Martin Davis, Daniel Dennett, Luciano Floridi, Lance Fortnow, Douglas Hofstadter, Wilfried Sieg, Aaron Sloman, Robert I. Soare, and Stephen Wolfram (reviewed in (Avigad, 2014)).

2. **Biographical Information:**

   (a) (Hodges, 2012a) is the standard biography. See also Hodges's "Alan Turing: The Enigma" website, http://www.turing.org.uk/index.html. Reviews of the first edition of Hodges's biography can be found in (Hofstadter, 1983), (Bernstein, 1986), and (Toulmin, 1984) (see also (Stern and Daston, 1984) with a reply by Toulmin).

   (b) Another biography is (Leavitt, 2005), reviewed in (Cooper, 2006).

   (c) (Fitzsimmons, 2013) discusses the British pardon of Turing for his "crime" of being homosexual.

   (d) (Smith, 2014b) reviews four Turing biographies.

3. **Dramatizations:**

   (a) *Breaking the Code* (Whitemore, 1966) is a play, later made into a superb film (http://en.wikipedia.org/wiki/Breaking_the_Code); the film may be online at http://www.youtube.com/watch?v=S23yie-779k. An excerpt of the play was published as (Whitemore, 1988).

   (b) *Enigma* (2001), http://en.wikipedia.org/wiki/Enigma_(2001_film) is of interest if only because this fictionalized version omits Turing!

   (c) *Codebreaker: The Story of Alan Turing* (2011), http://en.wikipedia.org/wiki/Codebreaker_(film): Half dramatization, half documentary, with interviews of people who knew Turing (including some who are fictionalized in *The Imitation Game*).

   (d) The most recent Hollywood treatment, rather controversial because of the many liberties it took with the facts, is *The Imitation Game* (2014), https://en.wikipedia.org/wiki/The_Imitation_Game. There are several websites that offer background and critiques of the film. One is "The Imitation Game: The Philosophical Legacy of Alan Turing", *The Critique* (31 December 2014), http://www.thecritique.com/news/the-imitation-game-the-philosophical-legacy-of-alan-turing/ Others include: (Anderson, 2014), (von Tunzelmann, 2014), (Woit, 2014), and (Caryl, 2015).

4. **Turing's Legacy:**

   (a) (Muggleton, 1994) "show[s] that there is a direct evolution in Turing's ideas from his earlier investigations of computability to his later interests in machine intelligence and machine learning."

   (b) (Copeland and Proudfoot, 1996) observes that "Turing was probably the first person to consider building computing machines out of simple, neuron-like elements connected together into networks in a largely random manner. ... By the application of what he described as 'appropriate interference, mimicking education' ... [such a] machine can be trained to perform any task that a Turing machine can carry out ..." (p. 361).

   (c) (Copeland, 1997) is an essay on hypercomputation (or "non-classical" computation),
   but the introductory section (pp. 690–698) contains an enlightening discussion of the scope and limitations of Turing's accomplishments.

   (d) (Davis, 2004), which is also on hypercomputation, contains a good discussion of Turing's role in the history of computation.

   (e) (American Mathematical Society, 2006) is a special issue of the *Notices of the AMS* that includes articles by Andrew Hodges, Solomon Feferman, S. Barry Cooper, and Martin Davis, among others.

   (f) (Soare, 2013b) compares Turing and Church to Michelangelo and Donatello, respectively (illustrations at http://www.people.cs.uchicago.edu/~soare/Art/).

   (g) (Daylight, 2013) argues from historical evidence that the reasons that Turing is considered "the father of computer science" have nothing to do with his involvement in computer design or construction.

   (h) (Daylight, 2014) "assess[es] the accuracy of popular descriptions of Alan Turing's influences and legacy" (p. 36).

   (i) (Haigh, 2014) discusses the role of the Turing machine in the history of computers, minimizing its historical importance to the actual development of computing machines.

   (j) (Bullynck et al., 2015) wonders "why ... computer science ma[de] a hero out of Turing".

   (k) Other evaluations include (Robinson, 1994), (Sieg, 1994, §3), (Kleene, 1995), (Leiber, 2006), (Michie, 2008), (Copeland, 2012), (Copeland, 2013), (Vardi, 2013), and (Vardi, 2017).

5. **Pedagogy:**

   (a) On Turing machines and pedagogy, see Schagrin et al. 1985, "Turing Machines", Appendix B, pp. 327–339, http://www.cse.buffalo.edu/~rapaport/Papers/schagrinetal85-TuringMachines.pdf; Tymoczko and Goodhart 1986; Suber 1997c,d; Michaelson 2012; Lodder 2014.

   (b) There are other Turing-machine exercises at http://www.math.nmsu.edu/hist_projects/ and http://www.math.nmsu.edu/hist_projects/DMRG.turing.pdf

6. **Implementations:**

   There are several Turing-machine simulators and implementations, some quite curious:

   (a) (Curtis, 1965) contains a program written for the IBM 1620 designed for education purposes.

   (b) (Weizenbaum, 1976, Ch. 2, "Where the Power of the Computer Comes From") contains a masterful presentation of a Turing Machine implemented with pebbles and toilet paper! (This is also an excellent book on the role of computers in society, by the creator of the "Eliza" AI program.)

   (c) Stewart 1994 and Brian Hayes 2007b are implementations of Turing machines using subway trains and railroad trains, respectively.

   (d) (Rendell, 2000), (Rendell, 2001), and (Rendell, 2010) are implementations of Turing machines in John Conway's Game of Life (on which, see https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life).

   (e) (Smith, 2014c) and (Smith, 2014a) are implementations on a business card! (For more information on the business-card Turing machines, see (Casselman, 2014).)

Chapter 9

What Is a Computer?
A Philosophical Perspective

Version of 24 June 2019,[1] DRAFT © 2004–2019 William J. Rapaport

> What is computation? *In virtue of what is something a computer? Why do we say a slide rule is a computer but an egg beater is not?* These are . . . the philosophical questions of computer science, inasmuch as they query foundational issues that are typicaly glossed over as researchers get on with their projects.
> —(Churchland and Sejnowski, 1992, p. 61, italics added)

> . . . everyone who taps at a keyboard, opening a spreadsheet or a word-processing program, is working on an incarnation of a Turing machine . . .
> —*Time* magazine, 29 March 1999, cited in (Davis, 2006a, p. 125)

Figure 9.1: http://familycircus.com/comics/march-6-2012/, ©2012 Bil Keane Inc.

[1] A version of an earlier draft of this chapter appears as Rapaport 2018c.

9.1 Readings:

1. Required:
   (a) Searle, John R. (1990), "Is the Brain a Digital Computer?", *Proceedings and Addresses of the American Philosophical Association* 64(3) (November): 21–37.
      i. online version without footnotes at:
      http://users.ecs.soton.ac.uk/harnad/Papers/Py104/searle.comp.html
      ii. A slightly revised version is often cited instead of the 1990 article:
      Searle, John (1992), *The Rediscovery of the Mind*
      (Cambridge, MA: MIT Press), Ch. 9.
   (b) Hayes, Patrick J. (1997), "What Is a Computer? An Electronic Discussion",
   The Monist 80(3).
      • Original emails at: http://www.philo.at/mii/wic.dir9601/maillist.html
   (c) Lloyd, Seth; & Ng, Y. Jack (2004), "Black Hole Computers", *Scientific American* 291(5) (November): 52–61,
   https://www.scientificamerican.com/article/black-hole-computers-2007-04/
      • Argues that the universe is a computer.
   (d) Weinberg, Steven (2002), "Is the Universe a Computer?", *New York Review of Books* 49(16) (24 October),
   http://www.nybooks.com/articles/2002/10/24/is-the-universe-a-computer/
      i. This is a review of Wolfram 2002 (see "Strongly Recommended", below).
      ii. There is a follow-up letter:
      Shepard, Harvey; & Weinberg, Steven (2003, 16 January), "Is the Universe a Computer?", *New York Review of Books* 50(1), http://www.nybooks.com/articles/2003/01/16/is-the-universe-a-computer/
2. Very Strongly Recommended:
   (a) Samuel, Arthur L. (1953), "Computing Bit by Bit, or Digital Computers Made Easy", *Proceedings of the IRE* [Institute of Radio Engineers] 41(10) (October): 1223–1230.
      • An early introduction to computers, aimed at (radio) engineers who might not be familiar with them; written by an IBM researcher who later became famous for his work on computer checkers-players.
   (b) Shagrir, Oron (2006), "Why We View the Brain as a Computer", *Synthese* 153(3) (December): 393–416, http://moon.cc.huji.ac.il/oron-shagrir/papers/Why_we_view_the_brain_as_a_computer.pdf
      • §1, "The Problem of Physical Computation: What Does Distinguish Computers from Other Physical Systems?", contains a good survey of various theories of what a computer is.
   (c) Piccinini, Gualtiero (2006), "Computational Explanation in Neuroscience", *Synthese* 153(3) (December): 343–353,
   http://www.umsl.edu/∼piccininig/Computational%20Explanation%20in%20Neuroscience.pdf
      • §§1–4 are a good summary of issues related to the nature of computationalism, observer-dependence (as opposed to what Searle calls "intrinsic" computationalism—see §9.5.4, below), and universal realizability (or "pancomputationalism").

(d) Piccinini, Gualtiero (2007), "Computational Modelling vs. Computational Explanation: Is Everything a Turing Machine, and Does It Matter to the Philosophy of Mind?", *Australasian Journal of Philosophy* 85(1): 93–115, http://www.umsl.edu/~piccininig/Is_Everything_a_TM.pdf

3. Strongly Recommended:

(a) Rapaport, William J. (1998), "How Minds Can Be Computational Systems", *Journal of Experimental and Theoretical Artificial Intelligence* 10: 403–419.

i. §1 is relevant to Searle 1990; it presents my understanding of what it means to say that cognition is computable (as opposed to "computational").

ii. §§2–3 criticize two other theories of computationalism; you can just skim these for now.

(b) Wolfram, Stephen (2002), "Introduction to *A New Kind of Science*", http://www.stephenwolfram.com/publications/recent/nksmain/

(c) Piccinini, Gualtiero (2010), "The Mind as Neural Software? Understanding Functionalism, Computationalism, and Computational Functionalism", *Philosophy and Phenomenological Research* 81(2) (September); 269–311, http://www.umsl.edu/~piccininig/Computational_Functionalism.htm

(d) Powell, Corey S. (2006), "Welcome to the Machine", *New York Times Book Review* (2 April): 19, http://www.nytimes.com/2006/04/02/books/review/02powell.html

- A review of Lloyd 2006 (see "Further Sources of Information", below)[2]

(e) Schmidhuber, Jürgen (2006), "The Computational Universe", *American Scientist* 94 (July-August): 364ff, http://www.americanscientist.org/bookshelf/pub/the-computational-universe

- Another review of Lloyd 2006 (see "Further Sources of Information", below)

(f) Hayes, Brian (2007), "Trains of Thought", *American Scientist* 95(2) (March-April): 108–113.

- Can a railroad layout be a computer?

(g) Rapaport, William J. (2007), "Searle on Brains as Computers", *American Philosophical Association Newsletter on Philosophy and Computers* 6(2) (Spring): 4–9, http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/v06n2Computers.pdf

- A direct reply to Searle 1990.

[2]Why do I recommend a review of a book, but not the book? Because the review offers an overview and summary that is worth reading independently of the book being reviewed.

9.2 Introduction

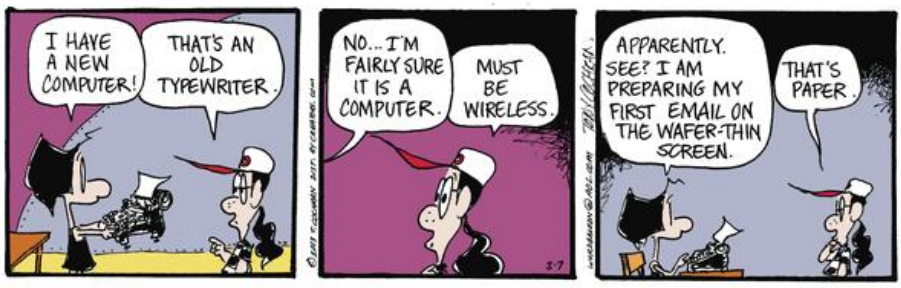


Figure 9.2: http://www.gocomics.com/agnes/2013/3/7, ©2013 T. Cochran

We began our study of the philosophy of computer science by asking what computer science is. We then asked what it studies: Does it study *computers*? Or *computing*? (Or maybe something else, such as information?) In Chapters 7 and 8, we began our investigation into what *computing* is. And in Chapter 6, we *began* our investigation into what a *computer* is, from a historical perspective.

In the present chapter, armed with the results of these investigations, we *return* to that question: If computer science is the study of computers, what *is* a computer? Of course, as we saw in §6.3, the earliest computers were humans! (To the extent that computer science is the study of computers, does that mean that computer science is, at least in part, a study of what humans are?) Note, however, that one of the questions that we will be looking at is whether the brain is a computer, so perhaps the issue of humans as computers has only been reformulated. In any case, when the question is asked today, it is generally assumed to refer to computing *machines*, and that is primarily the way that we will understand it in this chapter.

According to Arthur L. Samuel, in a 1953 article introducing computers to engineers who might never have encountered them before,

> a computer . . . can be looked at from two different angles, which Professor Hartree has called the "anatomical" and the "physiological," that is, "of what is it made?" and "how does it tick?" —Samuel (1953, p. 1223), citing Hartree 1949, p. 56

Samuel then goes on to describe the anatomy in terms of things like magnetic cores and vacuum tubes. Clearly, the anatomy has changed since then, so defining 'computer' "anatomically" in such terms doesn't seem to be the right way to go: It's too changeable. What's needed is a "physiological"—or functional—definition. At the very least, we might say that a computer is a physical machine (where, perhaps, it doesn't matter what it is made of) that is designed (that is, engineered) to compute (that is, to do computations) and, perhaps, that interacts with the world. (In Chapters 11 and 17, we'll see why I added "perhaps" to this interaction clause.)

But does a computer *have* to be a "machine"? Does it *have* to be "engineered"? If the brain is a computer, then it would seem that computers could be *biological* entities

(which, arguably, are not machines) that *evolved* (which, arguably, means that they were not engineered).[3] So, we should also ask whether the brain is a computer. But is it even correct to limit a computer to a *physical* device? Aren't Turing machines computers? Should we distinguish a "real" computer from a mathematical abstraction such as a Turing machine? But, arguably, my iMac—which is surely a computer if anything is—*isn't* a Turing machine; rather, it can be *modeled* by a (universal) Turing machine. And, to the extent that Turing machines don't interact with the world, so much the worse for them as a model of what a computer is.[4]

But what about a *virtual* computer implemented in some software, such as a program that "simulates" a computer of a certain type (perhaps even a Turing machine) but that runs on a (physical) computer of a very different type? For example, for an introductory course that I once taught, I wrote a very simple Pascal program that added two integers. This program was compiled (that is, implemented) using the "P88 Assembly Language Simulator"—a virtual machine whose programming language was "P88 Assembly Language", a very simple assembly language designed for instructional purposes (Biermann, 1990). That assembly language was written (that is, implemented) in another virtual machine whose programming language was a dialect of Pascal called MacPascal, which was, in turn, implemented in MacOS assembly language, which was implemented in the machine language that was implemented on a physical Mac II computer. (See Rapaport (2005b) for more details; we'll come back to this in §14.3.) Note that, ultimately, there is a physical substrate in these cases.

Question for the Reader:
When two integers are input to my original Pascal program, and their sum is output, "where" does the actual addition take place? Is it my Pascal program that adds the two integers? Or is it "really" the Mac II computer that adds them? Or is it one (or all?) of the intermediate implementations? (We'll return to these questions of what it is that a computer does in §§12.4.4.1.2.2 and 13.7.)

Further Reading:
For more on virtual machines, see Popek and Goldberg 1974, Pylyshyn 1992, and—especially—Pollock 2008.

If the purpose of a computer is to compute, what kind of computations do they perform? Are they restricted to mathematical computations? Even if that's so, is that a restriction? The binary-representation insight (§7.5.1) suggests that any (computable) information can be represented as a binary numeral; hence, any computation on such information could be considered to be a mathematical computation.

And what about the difference between a "hardwired" Turing machine that can only compute one thing and a "programmable" universal Turing machine that can compute anything that is computable? And is a "programmable" computer the same as a "stored-program" computer? Or what about the difference between a real, physical computer that can only compute whatever is *practically* computable (that is, subject to reasonable

[3] At least, not engineered by *humans*. (Dennett, 2017) would say that they *were* engineered—by Mother Nature, via the natural-selection algorithm.

[4] Thanks to my colleague Stuart C. Shapiro for many of these points.

space and time constraints) and an abstract, universal Turing machine that is not thus constrained?

And what about egg beaters, or rocks? Surely, they are *not* computers. Or are they? In short, what is a computer?

9.3 Informal Definitions

9.3.1 Reference-Book Definitions

If you ask a random person what a computer is, they might try to describe their laptop. If you look up 'computer' in a reference book,[5] you will find things like this (from the *Encyclopedia of Computer Science*):

> A *digital computer* is a machine that will accept data and information presented to it in a discrete form, carry out arithmetic and logical operations on this data, and then supply the required results in an acceptable form.
> —Morris and Reilly (2000, p. 539)

Or this (from the *Oxford English Dictionary*):

> **computer,** *n*.
>
> 1. A person who makes calculations or computations; a calculator, a reckoner; *spec[ifically,]* a person employed to make calculations in an observatory, in surveying, etc. Now chiefly *hist[orical].* [earliest citation dated 1613]
> 2. A device or machine for performing or facilitating calculation. [earliest citation dated 1869]
> 3. An electronic device (or system of devices) which is used to store, manipulate, and communicate information, perform complex calculations, or control or regulate other devices or machines, and is capable of receiving information (data) and of processing it in accordance with variable procedural instructions (programs or software); *esp[ecially]* a small, self-contained one for individual use in the home or workplace, used esp. for handling text, images, music, and video, accessing and using the Internet, communicating with other people (e.g. by means of email), and playing games. [earliest citation dated 1945]
>
> —*OED*, http://www.oed.com/view/Entry/37975 (my bracketed interpolations)

We'll come back to these in §9.3.5.

9.3.2 Von Neumann's Definition

In his "First Draft Report on the EDVAC", which—along with Turing's 1936 paper—may be taken as one of the founding documents of computer science, John von Neumann gives the following definition:

[5] But recall our caution in §5.2 about dictionary (or encyclopedia) definitions!

> An *automatic computing system* is a (usually highly composite) device, which can carry out instructions to perform calculations of a considerable order of complexity .... The instructions ... must be given to the device in absolutely exhaustive detail. They include all numerical information which is required to solve the problem under consideration ... . All these procedures require the use of some code to express ... the problem ..., as well as the necessary numerical material .... [T]he device ... must be able to carry them out completely and without any need for further intelligent human intervention. At the end of the required operations the device must record the results again in one of the forms referred to above.
> —von Neumann (1945, §1.0, p. 1)

Other comments (in this section of von Neumann 1945, as well as later, in §5.0 (pp. 6ff)) indicate that the code should be binary, hence that the computer is a "digital" device (§1.0, p. 1). This definition hews closely to being a physical implementation of a Turing machine, with clear allusions to the required algorithmic nature of the instructions, and with a requirement that there be both input and output (recall our discussion of the necessity of this—or lack thereof—in §7.4).

9.3.3 Samuel's Definition

Samuel's "physiological"—or functional—definition of a computer is this:

> an information or data processing device which accepts data in one form and delivers it in an altered form. —Samuel (1953, p. 1223)

This seems to be a very high-level description—perhaps *too* high a level: It omits any mention of computation or of algorithms. It does mention that the "delivered" data must have been "processed" from the "accepted" data by the "device"; so it's not just a *function* that relates the two forms of data—it's more of a function *machine*. But there's no specification of the *kind* of processing that it does.

Partly because of this, and on purpose, it also doesn't distinguish between analog and digital computers. Samuel resolves this by adding the modifier 'digital', commenting that "Any operation which can be reduced to arithmetic or to simple logic can be handled by such a machine. There does not seem to be any theoretical limit to the types of problems which can be handled in this way" (Samuel, 1953, p. 1224)—a nod, perhaps, to our binary-representation insight (Ch. 7, §7.5.1). Still, this doesn't limit the processing to *algorithmic* processing. It does, however, allow the *brain* to be considered as a computer: "when the human operator performs a reasonably complicated numerical calculation he [sic][6] is forcing his brain to act as a digital computer" (Samuel, 1953, p. 1224).[7]

A bit later (p. 1225), he does say that the processing must be governed by rules; this gets closer to the notion of an algorithm, though he (so far) puts no constraints on the rules. It is only after he discusses the control unit of the computer and its programming

[6] The use of the male gender here is balanced by Samuel's earlier statement that computers have "advantages in terms of the reductions in clerical manpower *and woman power*" (Samuel, 1953, p. 1223, my italics).

[7] See the quote from Chalmers 2011 at the end of §9.8.1, below.

(pp. 1226ff) that he talks about the kinds of control structures (loops, etc.) that are involved with algorithms. So, perhaps we could put all of this together and say that, for Samuel, a (digital) computer is a physical device that algorithmically processes digital data.

Further on, he adds the need for input and output devices (p. 1226). Are these really needed? Are they part of the abstract, mathematical model of a computer, namely, a Turing machine? Your first reaction might be to say that the tape serves as both input and output device. But the tape is an integral part of the Turing machine; it is really more like the set of internal switches of a physical computer, whereas physical computers normally have input and output devices (think of keyboards and monitors) as separate, additional components: Think of a computer like the Mac Mini, which is sold without a keyboard and a monitor. This is related to the necessity (or lack thereof!) of inputs and outputs that we discussed in §7.4.3.3. A computer with no input-output devices can only do batch processing of pre-stored data (if that—the Mac Mini can't do anything if there's no way to tell it to start doing something). Computers that interact with the external world require input-output devices, and that raises the question of their relationship to Turing machines (a discussion that we will begin in Chapter 10). Briefly, interacting computers that halt or that have only computable input are simulable by Turing machines; interacting computers with *non*-computable input are equivalent to Turing's oracle machines, which we will look at in §11.4.4.

9.3.4 Davis's Characterization

Computer scientist Martin Davis (Davis, 2000, pp. 366–367) suggests (but does not explicitly endorse) the idea that a computer is simply any device that carries out an algorithm. Of course, this depends on what 'carries out' means: Surely it has to include as part of its meaning that the internal mechanism of the device must operate in accordance with—must behave exactly like—one of the logically equivalent mathematical models of computation. Surely, any computer does that. But is anything that does that a computer? Can a computer be defined (merely) as a set of registers with contents or switches with settings? If they are binary switches, each is either on or else off; computation changes the contents (the settings). Do some of the register contents or switch settings have to be interpreted as data, some as program, and the rest as irrelevant (and some as output?). Who (or what) does the interpreting?

9.3.5 Discussion

One common thread in such definitions (ignoring the ones that are only of historical interest) is that computers are:

1. devices or machines . . .
2. . . . that take input (data, information),
3. process it (manipulate it; or operate, calculate, or compute with it) . . .
4. . . . in accordance with instructions (a program),

5. and then output a result (presumably, more data or information, but also including control of another device).

There are some other features that are usually associated with "computers": The kind that we are interested in must be, or typically are:

automatic

There is no human intervention (beyond, perhaps, writing the program). Of course, the holy grail of programming is to have self-programmed computers, possibly to include having the "desire" or "intention" to program themselves (as in science fiction). Humans might also supply the input or read the output, but that hardly qualifies as "intervention". (We will explore "intervention"—in the guise of "interactive" computing—in §11.4.3.)

general purpose

A computer must be capable of *any* processing that is "algorithmic", by means of a suitable program. This is the heart of Turing's universal machine. Recall that a Turing machine "runs" only *one* program. The universal Turing machine is also a Turing machine, so it, too, also runs only one program, namely, the fetch-execute cycle that enables the *simulation* of *another* (that is, *any* other) single-program Turing machine.

physically efficient

Many lists of computer features such as this one say that computers are *electronic*. But that is a matter of "anatomy". Modern computers are, as a matter of fact, electronic, but there is work on quantum computers (see the citations in §3.5.4), optical computers (https://en.wikipedia.org/wiki/Optical_computing), DNA computers (see the citations in §3.5.4), etc. So, being electronic is not essential. The crucial ("physiological") property is, rather, to be constructed in such a way as to allow for high processing speeds or other kinds of physical efficiencies. Turing (1950, §4, p. 439) noted this point:[8]

> Importance is often attached tothe fact that modern digital computers are electrical .... Since Babbage's machine was not electrical, and since all digital computers are in a sense equivalent, we see that this use of electricity cannot be of theoretical importance. Of course electricity usually comes in where fast signalling is concerned, so that it is not surprising that we find it in [digital computers] .... The feature of using electricity is thus seen to be only a very superficial similarity.

[8] In an earlier paper, he had said something similar: "From the point of view of the mathematician the property of being digital should be of greater interest than that of being electronic. That it is electronic is certainly important because these machines owe their high speed to this, and without the speed it is doubtful if financial support for their construction would be forthcoming. But this is virtually all that there is to be said on that subject. That the machine is digital however has more subtle significance. ... With digital machines however it is almost literally true that they are able to tackle any computing problem." (Turing, 1947, p. 378)

digital

Computers should process information expressed in discrete, symbolic form (typically, alpha-numeric form, but perhaps also including graphical form). The contrast is typically with being "analog", where information is represented by means of continuous physical quantities.

algorithmic

What about the "calculations", the "arithmetic and logical operations"? Presumably, these need to be algorithmic, though neither the *OED* nor the *Encyclopedia of Computer Science* definitions say so. And it would seem that the authors of those definitions have in mind calculations or operations such as addition, subtraction, etc.; maybe solving differential equations; Boolean operations involving conjunction, disjunction, etc; and so on. These require the data to be numeric (for math) or propositional (or truth-functional—for Boolean and logical operations), at least in some "ultimate" sense: That is, any *other* data (pictorial, etc.) must be encoded as numeric or propositional, or else would need to allow for other kinds of operations.

There are clear cases of things that *are* computers, both *digital* and *analog*. For example, Macs, PCs, etc. are clear cases of digital computers. And slide rules and certain machines at various universities are clear cases of analog computers. (However, these may be mostly of historical interest, don't seem to be programmable—that is, universal, in Turing's sense—and seem to be outside the historical development explored in Chapter 6.)

Further Reading:
On analog computers, see the citations in §6.5.2. On the university analog computers just mentioned, see Hedger 1998's report on the research of Jonathan W. Mills at Indiana University, and "Harvard Robotics Laboratory: Analog Computation", http://hrl.harvard.edu/analog/.

And there seem to be clear cases of things that are *not* computers: I would guess that most people would not consider rocks, walls, ice cubes, egg beaters, or solid blocks of plastic to be computers (note that I said 'most' people!). And there are even clear cases of devices for which it might be said that it is not clear whether, or in what sense, they are computers, such as Atanasoff and Berry's ABC. (Recall the patent lawsuit discussed in §6.5.4.)

So: What is a computer? What is the relation of a computer to a Turing machine and to a universal Turing machine? Is the (human) brain a computer? Is your smartphone a computer? *Could* a rock or a wall be considered to be a computer? Might *anything* be a computer? Might *everything*—such as the universe itself—be a computer? Or are some of these just badly formed questions?

Further Reading:
Chalmers 2011, "What about computers?", pp. 335–336 (originally written in 1993) suggests that a computer is a device that can implement multiple computations by being programmable. (The 2011 version of this essay was accompanied by commentaries, including Egan 2012; Rescorla 2012b; Scheutz 2012; and Shagrir 2012a; and a reply by Chalmers (Chalmers, 2012).)

Shagrir 1999 is a short, but wide-ranging, paper on the nature of computers, hypercomputation, analog computation, and computation as not being purely syntactic.

Harnish 2002 is a textbook survey of numerous definitions.

Kanat-Alexander 2008 defines a computer as "Any piece of matter which can carry out symbolic instructions and compare data in assistance of a human goal."

See also the website Anderson 2006 and the video Chirimuuta et al. 2014.

9.4 Computers, Turing Machines, and Universal Turing Machines

> All modern general-purpose digital computers are physical embodiments of the same logical abstraction[:] Turing's universal machine.
> —Robinson (1994, pp. 4–5)

9.4.1 Computers as Turing Machines

Let's try our hand at a more formal definition of 'computer'. An obvious candidate for such a definition is this:

> **(DC0) A computer is *any physical device that computes.***

Because a Turing machine is a mathematical model of what it means to compute, we can make this a bit more precise:

(DC1) A computer is *an implementation of a Turing machine.*

A Turing machine, as we have seen, is an abstract, mathematical structure. We will explore the meaning of 'implementation' in Chapter 14. For now, it suffices to say that an implementation of an abstract object is (usually) a physical object that satisfies the definition of the abstract one. (The hedge-word 'usually' is there in order to allow for the possibility of non-physical—or "virtual"—software implementations of a Turing machine.) So, a physical object that satisfies the definition of a Turing machine would be an "implementation" of one. Of course, no physical object can satisfy that definition if part of the definition requires it to be "perfect" in the following sense:

> A Turing machine is like an actual digital computing machine, except that (1) it is error free (i.e., it always does what its table says it should do), and (2) by its access to an unlimited tape it is unhampered by any bound on the quantity of its storage of information or "memory". —Kleene (1995, p. 27)

The type-(2) limitation of a "real" (physical) Turing machine is not a very serious one, given (a) the option of always buying another square and (b) the fact that no computation could require an actual infinity of squares (else it would not be a finite computation). The more significant type-(2) limitation is that some computations might require more squares than there could be in the universe (as is the case with NP computations such as playing perfect chess).

The type-(1) limitation of "real" Turing machines—being error free—does not obviate the need for program verification (see Chapter 16). Even an "ideal" Turing machine could be poorly programmed.

So let's modify our definition to take care of this:

(DC2) A computer is a "*physically plausible*" implementation of a Turing machine

where 'physically plausible' is intended to allow for those physical limitations. Turing himself said something similar: "Machines such as the ACE [Turing's "automatic computing engine"; recall §6.5.4] may be regarded as practical versions of this same type of machine" (that is, one of his *a*-machines; (Turing, 1947, p. 379; my interpolated brackets). And, more generally:

> If we take the properties of the universal [Turing] machine in combination with the fact that the machine processes and rule of thumb processes [that is, algorithmic processes] are synonymous we may say that the universal machine is one which, when supplied with the appropriate instructions, can be made to do any rule of thumb process. This feature is paralleled in digital computing machines such as the ACE, They are in fact practical versions of the universal machine. —Turing (1947, p. 383)

Let's now consider two questions:

- Is a Turing machine a computer?
- Is a Mac (or a PC, or any other real computer) a physically plausible implementation of a Turing machine?

The first question we can dismiss fairly quickly: Turing machines are not physical objects, so they can't be computers. A Turing machine is, of course, a mathematical model of a computer. (But a virtual, software implementation of a Turing machine *is*, arguably, a computer.)

The second question is trickier. Strictly speaking, the answer is 'no', because Macs (and PCs, etc.) don't behave the way that Turing machines do. They actually behave more like another mathematical model of computation: a register machine.

Further Reading:
For reasons that will become clear in a moment, it won't be necessary for us to go into the details of what a register machine is. But you can read about them in (Shepherdson and Sturgis, 1963) and at https://en.wikipedia.org/wiki/Register_machine. They are descended from the machines of Wang 1957 that we discussed in §7.5.2. We'll talk a bit more about them in §§11.4.4 and 14.4.3.

Register machines, however, are logically equivalent to Turing machines; they are just another mathematical model of computation. Moreover, other logically equivalent models of computation are even further removed from Turing machines or register machines: How might a computer based on recursive functions work? Or one based on the lambda calculus? (Think of Lisp Machines.) This suggests a further refinement to our definition:

(DC3) A computer is a physically plausible implementation of *anything logically equivalent to* a Turing machine.

There is another problem, however: Computers, in any informal sense of the term (think laptop or even mainframe computer) are programmable. Turing machines are not!

But *universal* Turing machines are! The ability to store a program on a universal Turing machine's tape makes it programmable; that is, the universal Turing machine can be changed from simulating the behavior of one Turing machine to simulating the behavior of a different one. A computer in the modern sense of the term really means a *programmable* computer, so here is a slightly better definition:

(DC4) A (programmable) computer is a physically plausible implementation of anything logically equivalent to a *universal* Turing machine.

But a program need not be stored physically in the computer: It could "control" the computer via a wireless connection from a different location. The ability to store a program *in* the computer *along with* the data allows for the program to change *itself*. Moreover, a hardwired, non-universal computer could be programmed by re-wiring it. (This assumes that the wires are manipulable. We'll return to this point in §12.4.) That's how early mainframe computers (like ENIAC) were programmed. So, this raises another question: What exactly is a "stored-program" computer, and does it differ from a "programmable" computer?

Further Reading:
Allen Newell points out that physical systems are limited in speed, space, and reliability (Newell, 1980, p. 148). Hence, a physical device that implements a Turing machine will inevitably be more limited than that abstract, ideal "machine". So, some functions that are theoretically computable on a Turing machine might *not* be physically computable on a real computer. In Chapter 10, we will be looking closely at the Church-Turing Computability Thesis. One question to think about is whether there is a difference between an abstract Computability Thesis that applies to (abstract) Turing machines, on the one hand, and a physical Computability Thesis that applies to (physical) computers, on the other hand. If Newell is right, then a physical version of the Computability Thesis is going to differ in plausibility from an abstract one. The abstract version can be made slightly more realistic, perhaps, by redefining, or placing limits on, some of the terms:

> [R]ather than talk about memory being actually unbounded, we will talk about it being *open*, which is to say available up to some point, which then bounds the performance, both qualitatively and quantitatively. *Limited*, in opposition to open, will imply that the limit is not only finite, but small enough to force concern. Correspondingly, *universal* can be taken to require only sufficiently open memory, not unbounded memory. —Newell (1980, p. 161)

9.4.2 Stored Program vs. Programmable

> [A] Turing machine ... was certainly thought of as being programmed in a 'hard-wired' way. ... It is reasonable to view the universal Turing machine as being programmed by the description of the machine it simulates; since this description is written on the memory tape of the universal machine, the latter is an abstract stored program computer. —Carpenter and Doran (1977, p. 270)

In my experience, the phrase 'stored program' refers to the idea that a computer's program can be stored in the computer itself (for example, on a Turing machine's tape) and be changed, either by storing a different program or by modifying the program itself (perhaps while it is being executed, and perhaps being (self-)modified by the program itself). However, when I asked a colleague who first came up with the notion of "stored program" (fully expecting him to say either Turing or von Neumann), he replied—quite reasonably—"Jacquard".[9]

On this understanding, the phrase 'stored-program computer' becomes key to understanding the difference between software and hardware (or programmed vs. hard-wired computer)—see Chapter 12 for more on this—and it becomes a way of viewing the nature of the universal Turing machine.

Here is von Neumann on the concept:

> If the device [the "very high speed automatic digital computing sytem" (§1.0, p. 1)] is to be *elastic*, that is as nearly as possible *all purpose*, then a distinction must be made between the specific instructions given for and defining a particular problem, and the general control organs which see to it that these instructions—no matter what they are—are carried out. The former must be **stored** in some way... the latter are represented by definite operating parts of the device. By the *central control* we mean this latter function only ....
> —von Neumann (1945, §2.3, p. 2; italics in original, my boldface).

The "specific instructions" seems clearly to refer to a specific Turing machine's fetch-execute program as encoded on the tape of a universal Turing machine. The "central control" seems clearly to refer to the universal Turing machine's program. So, if this is what is meant by "stored program", then it pretty clearly refers to the way that a universal Turing machine works.

Randell (1994, p. 12)—also discussing the controversy over what stored programming is—makes a statement that suggests that the main difference between programmed vs. hardwired computers might lie in the fact that "a program held on some read-only medium, such as switches, punched cards, or tape ... was quite separate from the (writable) storage device used to hold the information that was being manipulated by the machine". "[S]toring the program within the computer, in a memory that could be read at electronic speeds during program execution" had certain "advantages" first noted by "the ENIAC/EDVAC team" (p. 13). Randell, however, thinks that the analogy with universal Turing machines is more central (p. 13), and that "EDVAC does not qualify as a stored-program computer" because the "representations [of data and

[9] Stuart C. Shapiro, personal communication, 7 November 2013.

instructions] were quite distinct, and no means were provided for converting data items into instructions" or vice versa (p. 13).

Vardi (2013) defines 'stored-program' in terms of "uniform handling of programs and data", which he says can be "traced back to Gödel's arithmetization of provability". (But Copeland (2013) objects to this; Vardi replies in Vardi 2017.) The commonality between both of these ideas is that of representing two different things in *the same notation*: Programs and data can both be represented by '0's and '1's; logic and arithmetic can both be represented by numbers (or numerals; indeed, by '0's and '1's!). And it is worth noting that the brain represents everything by neuron firings. There is a second aspect of this commonality: Storing both data and program (represented in the same notation) *in the same place*: Programs and data can be stored in different sections of the same Turing-machine tape; arithmetical operations can be applied to both numbers and logical propositions; and all neuron firings are in the brain. If we reserve 'stored program' to refer to Vardi's commonality, it certainly seems to describe the principal feature of a universal Turing machine (even if Turing shouldn't be credited with the invention of the commonality). Clearly, a stored-program computer is programmable. Are all programmable computers stored-program computers?

Further Reading:
Haigh 2013 argues that the phrase 'stored-program computer' is ambiguous between several different readings and often conflated with the notion of a universal Turing machine, hence that it would be better to refrain from using it. See also Haigh and Priestley 2016.

According to Daylight 2013, p. XX, note 3, the phrase 'stored program' was not commonly used in the early 1950s. Furthermore, according to Daylight (2013, §3, p. VII), storing data and instructions together "was based on practical concerns, not theoretical reasoning' of the sort that might have been inspired by Turing's notion of a universal Turing machine.

Philosophical Digression:
There is a sense in which a stored-program computer does two things: It executes a "hardwired" fetch-execute cycle, and it executes whatever software program is stored on its tape. Which is it "really" doing? Newell (1980, p. 148) suggests that it is the former:

> A machine is defined to be a system that has a specific determined behavior as a function of its input. By definition, therefore, it is not possible for a single machine to obtain even *two* different behaviors, much less any behavior. The solution adopted is to decompose the input into two parts (or aspects): one part (the *instruction*) being takento determine which input-output function is to be exhibited by the second part (the *input-proper*) along with the output.

One way to interpret this passage is to take the "decomposition" to refer, on the one hand, to the hardwired, fetch-execute program of the univeral Turing machine, and, on the other hand, to the software, stored program of the "virtual" machine. The universal Turing machine thus ("consciously") executes the software program indirectly by directly ("unconsciously") executing its hardwired program. It *simulates* a Turing machine that has that software program hardwired as its machine table (Newell, 1980, p. 150).

In the next three sections, we will look at three recent attempts in the philosophical

literature to define 'computer'. In §9.8, we will briefly consider two non-standard, alleged examples of computers: brains and the universe itself.

9.5 John Searle: Anything Is a Computer

9.5.1 Searle's Argument

John Searle's presidential address to the American Philosophical Association, "Is the Brain a Digital Computer?" (Searle, 1990), covers a lot of ground and makes a lot of points about the nature of computers, the nature of the brain, the nature of cognition, and the relationships among them. In this section, we are going to focus on what Searle says about the nature of computers, with only a few side glances at the other issues.

Further Reading:
Searle 1990 was reprinted with a few changes as Chapter 9 of Searle 1992. For more detailed critiques and other relevant commentary, see Piccinini 2006b, 2007b, 2010a; and Rapaport 2007.

Here is Searle's argument relevant to our main question about what a computer is:

1. Computers are described in terms of 0s and 1s.
   (See Searle 1990, p. 26; Searle 1992, pp. 207–208.)

   > Taken literally, he is saying that computers are described in terms of certain *numbers*. Instead, he might have said that computers are described in terms of '0's and '1's. In other words, he might have said that computers are described in terms of certain *numerals*. Keep this distinction (which we discussed in §6.8.1) in mind as we discuss Searle's argument.

2. Therefore, being a computer is a syntactic property.
   (See Searle 1990, p. 26; Searle 1992, pp. 207.)

   > Syntax is the study of the properties of, and relations among, symbols or uninterpreted marks on paper (or on some other medium); a rough synonym is 'symbol manipulation' (see Chapter 22). In line with the distinction between numbers and numerals, note that only numerals are symbols.

3. Therefore, being a computer is not an "*intrinsic*" *property* of physical objects.
   (See Searle 1990, pp. 27–28; Searle 1992, p. 210.)

4. Therefore, *we can ascribe* the property of being a computer *to* any object.
   (See Searle 1990, p. 26; Searle 1992, p. 208.)

5. Therefore, everything is a computer.
   (See Searle 1990, p. 26; Searle 1992, p. 208.)

Of course, this doesn't quite answer our question, "What is a computer?". Rather, the interpretation and truth value of these theses will depend on what Searle thinks a computer is. Let's look at exactly what Searle says about these claims.

9.5.2 Computers Are Described in Terms of 0s and 1s

After briefly describing Turing machines as devices that can perform the actions of printing '0' or '1' on a tape and of moving left or right on the tape, depending on conditions specified in its program, Searle says this:

> If you open up your home computer you are most unlikely to find any 0's and 1's or even a tape. But this does not really matter for the definition. To find out if an object is really a digital computer, it turns out that we do not actually have to look for 0's and 1's, etc.; rather we just have to look for something that **we** could *treat as* or *count as* or *could be used to* function as 0's and 1's.
> —Searle (1990, p. 25, my boldface, Searle's italics); Searle 1992, p. 206

So, according to Searle, a computer is a physical object that can be *described* as a Turing machine. Recall from §8.9.1 that anything that satisfies the definition of a Turing machine *is* a Turing machine, whether it has a paper tape divided into squares with the symbols '0' or '1' printed on them or whether it is a table and placemats with beer mugs on them. All we need is to be able to "treat" some part of the physical object as *playing the role of* the Turing machine's '0's and '1's. So far, so good.

Or is it? Is your home computer *really* a Turing machine? Or is it a device whose behavior is "merely" *logically equivalent* to that of a Turing machine? That is, is it a device that can compute all and only the functions that a Turing machine can compute, even if it does so differently from the way that a Turing machine does? Recall that there are lots of different mathematical models of computation: Turing machines and recursive functions are two of them that we have looked at. Suppose someone builds a computer that operates in terms of recursive functions instead of in terms of a Turing machine. That is, it can compute successors, predecessors, and projection functions, and it can combine these using generalized composition, conditional definition, and while-recursion, instead of printing '0's and '1's, moving left and right, and combining these using "go to" instructions (changing from one *m*-configuration to another). These two computers (the Turing-machine computer and the recursive-function computer), as well as your home computer (with a "von Neumann" architecture, whose method of computation uses the primitive machine-language instructions and control structures of, say, an Intel chip), are all logically equivalent to a Turing machine, in the sense of having the same input-output behavior, but their internal behaviors are radically different. To use a terminology from an earlier chapter, we can ask: Are recursive-function computers, Turing machines, Macs, and PCs not only *extensionally* equivalent but also *intensionally* equivalent? Can we really describe the recursive-function computer and your home computer in terms of a Turing machine's '0's and '1's? Or are we limited to showing that anything that the recursive-function computer and your home computer can compute can also be computed by a Turing machine (and vice versa)—but not necessarily *in the same way*?

Here is an analogy to help you see the issue: Consider translating between French and English. To say 'It is snowing' in French—that is, to convey in French the same information that 'It is snowing' conveys in English—you say: *Il neige*. The 'il' means "it", and the 'neige' means "is snowing". This is very much like (perhaps it is exactly like) describing the recursive-function machine's behavior (analogous to the French

sentence) using '0's and '1's (analogous to the English sentence).

But here is a different example: In English, if someone says: 'Thank you', you might reply, 'You're welcome', but, in French, if someone says *Merci*, you might reply: *Je vous en prie*. Does '*merci*' "mean" (the same as) 'thank you'? Does '*Je vous en prie*' "mean" (the same as) 'You're welcome'? Have we translated the English into French, in the way that we might "translate" a recursive-function algorithm into a Turing machine's '0's and '1's? Not really: Although '*merci*' is *used* in much the same way in French that 'thank you' is *used* in English, there is no part of '*merci*' that *means* (the same as) 'thank' or 'you'; and the literal translation of '*je vous en prie*' is something like 'I pray that of you'. So there is a way of communicating the same information in both French and English, but the phrases used are not literally inter-translatable.

So, something might be a computer without being "described in terms of '0's and '1's", depending on exactly what you mean by 'described in terms of'. Perhaps Searle should have said something like this: Computers are described in terms of the primitive elements of the mathematical model of computation that they implement. But let's grant him the benefit of the doubt and continue looking at his argument.

9.5.3 Being a Computer Is a Syntactic Property

Let us suppose, for the sake of the argument, that computers are described in terms of '0's and '1's. Such a description is *syntactic*. This term (which pertains to symbols, words, grammar, etc.) is usually contrasted with 'semantic' (which pertains to meaning), and Searle emphasizes that contrast early in his essay when he says that "syntax is not the same as, nor is it by itself sufficient for, semantics" (Searle, 1990, p. 21). But now Searle uses the term 'syntactic' as a contrast to being *physical*. Just as there are many ways to be computable (Turing machines, recursive functions, lambda-calculus, etc.)—all of which are equivalent—so there are many ways to be a carburetor. "A carburetor ... is a device that blends air and fuel for an internal combustion engine" (http://en.wikipedia.org/wiki/Carburetor), *but it doesn't matter what it is made of*, as long as it can perform that blending "function" (purpose). "[C]arburetors can be made of brass or steel" (Searle, 1990, p. 26); they are "multiply realizable"—that is, you can "realize" (or make) one in "multiple" (or different) ways. They "are defined in terms of the production of certain *physical* effects" (Searle, 1990, p. 26).

> But the class of computers is defined **syntactically** in terms of the *assignment* of 0's and 1's.
> —Searle (1990, p. 26, Searle's italics, my boldface); Searle 1992, p. 207

In other words, if something is defined in terms of symbols, like '0's and '1's, then it is defined in terms of syntax, not in terms of what it is physically made of.

Hence, being a computer is a syntactic property, not a physical property. It is a property that something has in virtue of ... of what? There are two possibilities, given what Searle has said. First, perhaps being a computer is a property that something has in virtue of *what it **does**, its function or purpose*. Second, perhaps being a computer is a property that something has in virtue of *what someone **says** that it does, how it*

is described. But what something *actually* does may be different from what someone *says* that it does.

So, does Searle think that something is a computer in virtue of its *function* or in virtue of its *syntax*? Recall our thought experiment from §3.9.5 (inspired by (Weizenbaum, 1976, Ch. 5)): Suppose you find a black box with a keyboard and a screen in the desert and that, by experimenting with it, you determine that it displays on its screen the greatest common divisor (GCD) of two numbers that you type into it. It certainly seems to *function* as a computer (as a Turing machine for computing GCDs). And you can probably describe it in terms of '0's and '1's, so you can also *say* that it is a computer. It seems that *if something functions as a computer, then you can describe it in terms of '0's and '1's.*

What about the converse? If you can *describe* something in terms of '0's and '1's, does it *function* as a computer? Suppose that the black box's behavior is inscrutable: The symbols on the keys are unrecognizable, and the symbols displayed on the screen don't seem to be related in any obvious way to the input symbols. But suppose that someone manages to invent an interpretation of the symbols in terms of which the box's behavior can be described as computing GCDs. Is "computing GCDs" really what it does? Might it not have been created by some extraterrestrials solely for the purpose of entertaining their young with displays of pretty pictures (meaningless symbols), and that it is only by the most convoluted (and maybe not always successful) interpretation that it can be described as computing GCDs?

You might think that the box's *function* is more important for determining what it is. Searle thinks that our ability to *describe* it syntactically is more important! After all, whether or not the box was *intended* by its creators to compute GCDs or to entertain toddlers, if it can be accurately *described* as computing GCDs, then, in fact, it computes GCDs (as well as, perhaps, entertaining toddlers with pretty pictures).

An alternative view of this is given by the logician and philosopher Nicolas D. Goodman:

> Suppose that a student is successfully doing an exercise in a recursive function theory course which consists in implementing a certain Turing machine program. There is then no reductionism involved in saying that he is carrying out a Turing machine program. He intends to be carrying out a Turing machine program. ... Now suppose that, unbeknownst to the student, the Turing machine program he is carrying out is an implementation of the Euclidean algorithm. His instructor, looking at the pages of more or less meaningless computations handed in by the student, can tell from them that the greatest common divisor of 24 and 56 is 8. The student, not knowing the purpose of the machine instructions he is carrying out, cannot draw the same conclusion from his own work. I suggest that the instructor, but not the student, should be described as carrying out the Euclidean algorithm. (This is a version ... of Searle's Chinese room argument ...)[10]
> —Goodman (1987, p. 484)

Again, let's grant this point to Searle. He then goes on to warn us:

> But this has two consequences which might be disastrous:

[10] We will discuss the Chinese Room Argument in Ch. 19.

> 1. The same principle that implies multiple realizability would seem to imply universal realizability. If computation is defined in terms of the assignment of syntax then everything would be a digital computer, because any object whatever could have syntactical ascriptions made to it. You could describe anything in terms of 0's and 1's.
> 2. Worse yet, syntax is not intrinsic to physics. The ascription of syntactical properties is always relative to an agent or observer who treats certain physical phenomena as syntactical.
> —Searle (1990, p. 26); Searle 1992, pp. 207–208

Let's take these in reverse order.

9.5.4 Being a Computer Is Not an Intrinsic Property of Physical Objects

According to Searle, being a computer is not an intrinsic property of physical objects, because being a computer is a syntactic property, and "syntax is not intrinsic to physics". What does that quoted thesis mean, and why does Searle think that it is true?

What is an "intrinsic" property? Searle doesn't tell us, though he gives some examples:

> [G]reen leaves intrinsically perform photosynthesis[;] . . . hearts intrinsically pump blood. It is not a matter of us arbitrarily or "conventionally" assigning the word "pump" to hearts or "photosynthesis" to leaves. There is an actual fact of the matter. —Searle (1990, p. 26); Searle 1992, p. 208

So, perhaps "intrinsic" properties are properties that something "really" has as opposed to merely being *said* to have, much the way our black box in the previous section may or may not "really" compute GCDs but can be *said* to compute them. But what does it mean to "really" have a property?

Here are some possible meanings for 'intrinsic':

1. An object might have a property P "intrinsically" if it has P "essentially" rather than "accidentally". An *accidental* property is a property that something has that is such that, if the object lacked that property, then it would still be the *same* object. So, it is merely an accidental property of me that I was wearing a tan shirt on the day that I wrote this sentence. If I lacked that property, I would still be the same person. An *essential* property is a property that something has such that, if the object lacked that property, then it would be a *different* object. So, it is an essential property of me that I am a human being. If I lacked that property, I wouldn't even be a person at all. (This is the plot of Franz Kafka's story *The Metamorphosis*, in which the protagonist awakes one day to find that he is no longer a human, but a beetle.) The exact nature of the essential-accidental distinction, and its truth or falsity, are matters of great dispute in philosophy. Here, I am merely suggesting that perhaps this is what Searle means by 'intrinsic': Perhaps he is saying that being a computer is not an essential property of an object, but only an accidental property.

2. An object might have *P* "intrinsically" if it has *P* as a kind of "first-order" property, not as a kind of "second-order" property. A *second-order* property is a property that something has in virtue of having some other (or *first-order*) property. A simple example is a "relational property", such as the property of being an aunt or the property of being an uncle: Someone is an aunt or uncle only if someone else (such as a sibling) has a child. Another example is an object's color. An apple has the property of being red, not "intrinsically", but in virtue of reflecting light with wavelength approximately 650 nm. We perceive such reflected light in a certain way, which we call 'red'. But, conceivably, someone with a different neural make-up (say, red-green color-blindness) might perceive it either as a shade of gray that is indistinguishable from green or, in a science-fiction kind of case called an "inverted spectrum", as green. The point is that it is the measurable wavelength of reflected light that might be an "intrinsic" property belonging to the apple, whereas the perceived color is a property belonging to the perceiver and only "secondarily" to the apple itself. So, perhaps Searle is saying that being a computer is only a second-order property of an object.

3. Another way that an object might have *P* "intrinsically" (perhaps this is closer to what Searle has in mind) is if *P* is a "natural kind". (Recall our discussion of this in §3.3.3.1.) This is another controversial notion in philosophy. The idea, roughly, is that a natural kind is a property that something has as a part of "nature" and not as a result of what an observer thinks.[11] So, being a bear is a natural kind; there would be bears even if there were no people to see them or to call them 'bears'. This does not mean that it is easy to define such natural kinds. Is a bear "A heavily-built, thick-furred plantigrade quadruped, of the genus *Ursus*; belonging to the *Carnivora*, but having teeth partly adapted to a vegetable diet" (http://www.oed.com/view/Entry/16537)—that is, a heavily-built, thick-furred mammal that walks on the soles of its four feet, eats meat, but can also eat plants? What if a bear was born with only three feet (or loses one in an accident), or never eats meat? Is an animal a bear whether or not it satisfies such definitions, and whether or not there were any humans to try to give such definitions? If so, then being a bear is a natural kind and, perhaps, an "intrinsic" property. As we saw in §3.3.3, Plato once said that it would be nice if we could "carve nature into its joints", that is, find the real, perhaps "intrinsic", natural properties of things (Phaedrus 265d–e). But perhaps the best we can do is to "carve joints *into* nature", that is, to "overlay" categories onto nature so that we can get some kind of understanding of it and control over it, even if those categories don't really exist in nature. Is being a computer a natural kind? Well, it's certainly not like being a bear! There probably aren't any computers in nature (unless the brain is a computer; and see §9.8.2 on whether nature itself is a computer), but there may also not be any prime numbers in nature, yet mathematical objects are something thought to exist independently of human thought. If they

[11] You'll note that several of these concepts are closely related; they may even be indistinguishable. For instance, perhaps "essential" properties are "natural kinds" or perhaps second-order properties are not natural kinds. Investigating these relationships is one of the tasks of metaphysics. It is beyond the scope of the philosophy of computer science.

do, then, because a Turing machine is a mathematical object, it might exist independently of human thought and, hence, being a computer might be able to be considered to be a "natural" mathematical kind. So, perhaps Searle is saying that being a computer is not a natural kind in one of these senses.

Further Reading:
For more discussion on what 'intrinsic' means, see (Lewis, 1983), (Langton and Lewis, 1998), (Skow, 2007), (Bader, 2013) (a highly technical essay, but it contains useful references to the literature on "intrinsic properties"), (Marshall, 2016), and (Weatherson and Marshall, 2018).

(Rescorla, 2014a, p. 180) uses slightly different terms for similar ideas:

> *Inherited meanings* arise when a system's semantic properties are assigned to it by external observers, either through explicit stipulation or through tacit convention. Nothing about the system helps generate its own semantics. *Indigenous meanings* arise when a system helps generate its own semantics (perhaps with ample help from its evolutionary, design, or causal history, along with other factors). The system helps confer content upon itself, through its internal operations or its interactions with the external world. Its semantics does not simply result from external assignment.

In fact, a computer is probably not a natural kind for a different reason: It is an artifact, something created by humans. Again, the nature of artifacts is controversial (as we saw in §3.5.1): Clearly, chairs, tables, skyscrapers, atomic bombs, and pencils are artifacts; you don't find them in nature, and if humans had never evolved, there probably wouldn't be any of these artifacts. But what about bird's nests, beehives, beaver dams, and other such things constructed by non-human animals? What about socially "constructed" objects like money? One of the crucial features of artifacts is that what they *are* is relative to what a person *says* they are. You won't find a table occurring naturally in a forest, but if you find a tree stump, you might use it as a table. So, something might be a computer, Searle might say, only if a human uses it that way or can describe it as one. In fact, Searle says this explicitly:

> [W]e might discover in nature objects which had the same sort of shape as chairs and which could therefore be used as chairs; but we could not discover objects in nature which were functioning as chairs, except relative to some agents who regarded them or used them as chairs. —Searle (1990, p. 28); Searle 1992, p. 211

Why does Searle think that syntax is not "intrinsic" to physics? Because " 'syntax' is not the name of a physical feature, like mass or gravity. ... [S]yntax is essentially an observer relative notion" (Searle 1990, p. 27; Searle 1992, p. 209). I think that what Searle is saying here is that we can analyze physical objects in different ways, no one of which is "privileged" or "more correct"; that is, we can carve nature into different joints, in different ways. On some such carvings, we may count an object as a computer; on others, we wouldn't. By contrast, an object has mass independently of how it is described: *Having* mass is *not* relative to an observer. *How* its mass is measured *is* relative to an observer.

But couldn't being a computer be something like that? There may be lots of different ways to measure mass, but an object always has a certain quantity of mass, no matter whether you measure it in grams or in some other units. In the same way, there may be lots of different ways to measure length, but an object always has a certain length, whether you measure it in centimeters or in inches. Similarly, an object (natural or artifactual) will have a certain structure, whether you describe it as a computer or as something else. If that structure satisfies the definition of a Turing machine, then it *is* a Turing machine, no matter how anyone *describes* it.

Searle anticipates this reply:

> [S]omeone might claim that the notions of "syntax" and "symbols" are just a manner of speaking and that what we are really interested in is the existence of systems with discrete physical phenomena and state transitions between them. On this view we don't really need 0's and 1's; they are just a convenient shorthand.
> —Searle (1990, p. 27); Searle 1992, p. 210

Compare this to my example above: Someone might claim that specific units of measurement are just a manner of speaking and that what we are really interested in is the actual length of an object; on this view, we don't really need centimeters or inches; they are just a convenient shorthand.

Searle replies:

> But, I believe, this move is no help. **A physical state of a system is a computational state only relative to the assignment to that state of some computational role, function, or interpretation.** The same problem arises without 0's and 1's because notions such as computation, algorithm and program do not name intrinsic physical features of systems. Computational states are not *discovered within* the physics, they are *assigned* to the physics.
> —Searle (1990, p. 27, my boldface, Searle's italics.); Searle 1992, p. 210

But this just repeats his earlier claim; it gives no new reason to believe it. He continues to insist that being a computer is more like "inches" than like length.

Further Reading:
For more detailed objections to Searle from the nature of measurement, see (Dresner, 2010) and (Matthews and Dresner, 2017).

So, we must ask again: Why does Searle think that syntax is not intrinsic to physics? Perhaps, if a property is intrinsic to some object, then that object can only have the property in one way. For instance, color is presumably not intrinsic to an object, because an object might have different colors depending on the conditions under which it is perceived. But the physical structure of an object that causes it to reflect a certain wavelength of light is always the same; that physical structure is intrinsic. On this view, here is a reason why syntax might not be intrinsic: The syntax of an object is, roughly, its abstract structure.[12] But an object might be able to be understood in terms of several different abstract structures (and this might be the case whether or

[12] See Ch. 14, §14.3; Ch. 19, §19.6.3.3; and Ch. 22 for further discussion of this point.

not human observers assign those structures to the object). If an object has no unique syntactic structure, then syntax is not intrinsic to it. But if an object has (or can be assigned) a syntax of a certain kind, then it does have that syntax even if it also has another one. And if, under one of those syntaxes, the object is a computer, then it *is* a computer.

But that leads to Searle's next point.

9.5.5 We Can Ascribe the Property of Being a Computer to Any Object

There is some slippage in the move from "syntax is not intrinsic to physics" to "we can ascribe the property of being a computer to any object". Even if syntax is not intrinsic to the physical structure of an object (perhaps because a given object might have several different syntactic structures), why must it be the case that *any* object *can* be ascribed the syntax of being a computer?

One reason might be this: Every object has (or can be ascribed) *every* syntax. That seems to be a very strong claim. To refute it, however, all we would need to do is to find an object O and a syntax S such that O lacks (or cannot be ascribed) S. One possible place to look would be for an O whose "size" in some sense is smaller than the "size" of some S. I will leave this as an exercise for the reader: If you can find such O and S, then I think you can block Searle's argument at this point.

Here is another reason why *any* object might be able to be ascribed the syntax of being a computer: There might be something special about the syntax of being a computer—that is, about the formal structure of Turing machines—that does allow it to be ascribed to (or found in) any object. This may be a bit more plausible than the previous reason. After all, Turing machines are fairly simple. Again, to refute it, we would need to find an object O such that O lacks (or cannot be ascribed) the syntax of a Turing machine. Again, I will leave this as an exercise for the reader, but we will return to it later (when we look at the nature of "implementation" in Chapter 14). Searle thinks that we cannot find such an object.

9.5.6 Everything Is a Computer

> Unlike computers, ordinary rocks are not sold in computer stores and are usually not taken to perform computations. Why? What do computers have that rocks lack, such that computers compute and rocks don't? (If indeed they don't?) ... A good account of computing mechanisms should entail that paradigmatic examples of computing mechanisms, such as digital computers, calculators, both universal and non-universal Turing machines, and finite state automata, compute. ... A good account of computing mechanisms should entail that all paradigmatic examples of non-computing mechanisms and systems, such as planetary systems, hurricanes, and digestive systems, don't perform computations. —Piccinini (2015, pp. 7, 12)

We can ascribe the property of being a computer to any object if and only if everything is a computer.

> Thus for example the wall behind my back is right now implementing the Wordstar program, because there is some pattern of molecule movements which is isomorphic with the formal structure of Wordstar.
> —Searle (1990, p. 27); Searle 1992, pp. 208–209

Searle does not offer a detailed argument for how this might be the case, but other philosophers have done so, and, in Chapter 14, we will explore how they think it can be done. Let's assume, for the moment, that it can be done.

In that case, things are not good, because this trivializes the notion of being a computer. If everything has some property P, then P isn't a very interesting property; P doesn't help us categorize the world, so it doesn't help us understand the world:

> [A]n objection to Turing's analysis... is that although Turing's account may be necessary it is not sufficient. If it is taken to be sufficient then too many entities turn out to be computers. The objection carries an embarrassing implication for computational theories of mind: such theories are devoid of empirical content. If virtually anything meets the requirements for being a computational system then wherein lies the explanatory force of the claim that the brain is such a system?
> —Copeland (1996, §1, p. 335).

So, x is a computer iff x is a (physical) model of a Turing machine. To say that this "account" is "necessary" means that, if x is a computer, then it is a model of a Turing machine. That seems innocuous. To say that it is a "sufficient" account is to say that, if x is a model of a Turing machine, then it is a computer. This is allegedly problematic, because, allegedly, anything can be gerrymandered to make it a model of a Turing machine; hence, anything is a computer (including, for uninteresting reasons, the brain).

How might we respond to this situation? One way is to bite the bullet and accept that, under some description, any object (even the wall behind me) can be considered to be a computer. And not just some specific computer, such as a Turing machine that executes the Wordstar program:

> [I]f the wall is implementing Wordstar then if it is a big enough wall it is implementing any program, including any program implemented in the brain.
> —Searle (1990, p. 27); Searle 1992, p. 209

If a big enough wall implements any program, then it implements the universal Turing machine!

But perhaps this is OK. After all, there is a difference between an "intended" interpretation of something and what I will call a "gerrymandered" interpretation. For instance, the intended interpretation of Peano's axioms for the natural numbers is the sequence $\langle 0, 1, 2, 3, \ldots \rangle$. There are also many other "natural" interpretations, such as $\langle$I, II, III,...$\rangle$, or $\langle \varnothing, \{\varnothing\}, \{\{\varnothing\}\}, \ldots \rangle$, or $\langle \varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \ldots \rangle$, and so on. As Chris Swoyer (1991, p. 504, note 26) notes, "According to structuralism, any countably infinite (recursive) set can be arranged to form an ω-sequence that can play the role of the natural numbers. It is the structure common to all such sequences, rather than the particular objects which any happens to contain, that is important for arithmetic." But extremely contorted ones, such as a(n infinite) sequence of all numeral names in French

arranged alphabetically, are hardly "good" examples. Admittedly, they *are* examples of natural numbers, but not very useful ones. (For further discussion, see Benacerraf 1965.)

A better reply to Searle, however, is to say that he's wrong: Some things are *not* computers. Despite what he said in the last passage quoted above, the wall behind me is *not* a universal Turing machine; I really cannot use it to post to my Facebook account or to write a letter, much less to add $2+2$. It is an empirical question whether something actually behaves as a computer. And the same goes for other syntactic structures. Consider the formal definition of a mathematical group:

> A *group* is$_{def}$ a set of objects (for example, integers) that is closed under an associative binary operation (for example, addition), that has an identity element (for example, 0), and is such that every element of the set has an inverse (for example, in the case of integer n, its inverse is $-n$).

Not every set is a group. Similarly, there is no reason to believe that everything is a Turing machine.

> In order for the system to be used to compute the addition function these causal relations have to hold *at a certain level of grain*, a level that is determined by the discriminative abilities of the user. That is why ... no money is to be made trying to sell a rock as a calculator. Even if (*per mirabile*) there happens to be a set of state-types at the quantum-mechanical level whose causal relations do mirror the formal structure of the addition function, microphysical changes at the quantum level are *not* discriminable by human users, hence human users could not use such a system to add. (God, in a playful mood, could use the rock to add.)
> —Egan (2012, p. 46)

Chalmers (2012, pp. 215–216) makes much the same point:

> On my account, a pool table will certainly implement various a-computations [that is, computations as abstract objects] and perform various c-computations [that is, concrete computational processes]. It will probably not implement interesting computations such as algorithms for vector addition, but it will at least implement a few multi-state automata and the like. These computations will not be of much explanatory use in understanding the activity of playing pool, in part because so much of interest in pool are not organizationally invariant and therefore involve more than computational structure.

In other words, even if Searle's wall implements Wordstar, *we* wouldn't be able to use it as such.

9.5.7 Other Views in the Vicinity of Searle's

> We count something as a computer because, and only when, its inputs and outputs can usefully and systematically be interpreted as representing the ordered pairs of some function that interests us. ... This means that delimiting the class of computers is not a sheerly empirical matter, and hence that "computer" is not a

> natural kind. ... Similarly, we suggest, there is no intrinsic property necessary and sufficient for all computers, just the interest-relative property that someone sees value in interpreting a system's states as representing states of some other system, and the properties of the system support such an interpretation. ... [I]n this very wide sense, even a sieve or a threshing machine [or an eggbeater!?] could be considered a computer.... —Churchland and Sejnowski (1992, pp. 65–66)

This is essentially Searle's point, only with a positive spin put on it. Note that their definition in the first sentence has an objective component (the inputs and outputs must be computationally related; note, too, that no specification is placed on whether the mechanism by which the inputs are transformed into the outputs is a computational one) as well as a subjective component (if the function computed by the alleged computer is of no human interest, then it is not a computer!).

Thus, this is a bit different from Searle: Where Searle says that the wall behind me is (or can be interpreted as) a word processor, Churchland & Sejnowski say that the wall behind me is computing something, but we don't care what it is, so we don't bother considering it to be a computer.

Presumably, the wall behind me doesn't have to *be* a computer in order for its (molecular or subatomic) behavior to be *describable* computationally. Or is Searle making a stronger claim, namely, that, not only is its behavior *describable* computationally, but it *is* a computation? Dana Ballard (1997, p. 11) has an interesting variation on that stronger claim:

> Something as ordinary as a table might be thought of as running an algorithm that adjusts its atoms continually, governed by an energy function. Whatever its variables are, just denote them collectively by x. Then you can think of the table as solving the problem of adjusting its atoms so as to minimize energy, that is, $\min_x E(x)$. Is this computation?

Note that this is different from Searle's claim that the table (or a wall) might be computing a word processor. It seems closer to the idea that the solar system might be computing Kepler's law (see §9.8.2, below).

Another claim in the vicinity of Searle's and Ballard's concerns DNA computing:

> Computer. The word conjures up images of keyboards and monitors. ... But must it be this way? The computer that you are using to read these words [that is, your brain!—comment by WJR] bears little resemblance to a PC. Perhaps our view of computation is too limited. *What if computers were ubiquitous and could be found in many forms? Could a liquid computer exist in which interacting molecules perform computations?* The answer is yes. This is the story of the DNA computer. —Adleman (1998, p. 54, my italics)

Of course, Adelman is not making the Searlean claim that everything is a computer and that, therefore, the interacting molecules of (any) liquid peform computations. Nor is he making the Ballardian claim that DNA computes in the way that a table computes. (Others have, however, made such a claim, on the grounds that strands of DNA are similar to Turing-machine tapes with a four symbols instead of two and with the processes

of DNA transcription and recombination as being computable processes (Shapiro and Benenson, 2006).) Rather, Adleman's claim is that one can use DNA "to solve mathematical problems". However, contrary to what the editors of *Scientific American* wrote in their subtitle to Adleman's article, it is unlikely that that "is redefining what is meant by 'computation' ". After all, the advent of transistors did not change Turing's mathematical characterization of computing any more than the use of vacuum tubes did. At most, DNA computers might change what the lay public means by 'computer'. But (as we saw in §6.3) that has already happened, with the meaning changing from "humans" to "computing machines".

Let's take stock of where we are. Presumably, computers are things that compute. Computing is the process that Turing machines give a precise description for. That is, computing is what Turing machines do. And, what Turing machines do is to move around in a discrete fashion and print discrete marks on discrete sections of the space in which they move around. So, a computer is a *device*—presumably, a *physical* device—that does that. Searle agrees that computing is what Turing machines do, and he seems to agree that computers are devices that compute. He also believes that everything is a computer; more precisely, he believes that everything can be *described as* a computer (because that's what it means to *be* a computer). And we've also seen reason to think that he might be wrong about that last point.

In the next two sections, we look at two other views about what a computer is.

9.6 Patrick Hayes: Computers as Magic Paper

Let's keep straight about three intertwined issues that we have been looking at:

1. What is a computer?
2. Is the brain a computer?
3. Is everything a computer?

Our principal concern is with the first question. Once we have an answer to that, we can try to answer the others. As we've just seen, Searle thinks that a computer is anything that is (or can be described as) a Turing machine, that everything is (or can be described as) a computer, and, therefore, that the brain is a computer, but only trivially so, and not in any interesting sense.

AI researcher Patrick J. Hayes (1997) gives a different definition, in fact, two of them. Here's the first:

Definition H1

> By "computer" I mean *a machine which performs computations, or which computes*.
> —Hayes (1997, p. 390, my italics)

A full understanding of this requires a definition of 'computation'; this will be clarified in his second definition. But there are a few points to note about this first one.

He prefaces it by saying:

> First, I take it as simply obvious both that computers exist and that not everything is a computer, so that, contra Searle, the concept of "computer" is not vacuous. —Hayes (1997, p. 390)

So, there are (1) machines that compute (that is, there are things that *are* machines-that-compute), and there are (2) things that are *not* machines-that-compute. Note that (2) can be true in two ways: There might be (2a) *machines* that *don't* compute, or there might be (2b) *things* that *do* compute but that *aren't machines*. Searle disputes the first possibility, because he thinks that everything (including, therefore, any machine) computes. But contrary to what Hayes says, Searle would probably *agree* with the second possibility, because, after all, he thinks that everything (including, therefore, anything that is not a machine) computes! Searle's example of the wall that implements (or that can be interpreted as implementing) Wordstar would be such a non-machine that computes. So, for Hayes's notion to contradict Searle, it must be that Hayes believes that there are machines that do not compute. Perhaps that wall is one of them, or perhaps a dishwasher is a machine that doesn't compute anything.[13]

Further Reading:
For a detailed study of what it means for a *machine* to compute, see Piccinini 2015. See also Bacon 2010.

Are Hayes's two "obvious" points to be understood as criteria of adequacy for any definition—criteria that Hayes thinks need no argument (that is, as something like "axioms")? Or are they intended to be more like "theorems" that follow from his first definition? If it's the former, then there is no interesting debate between Searle and Hayes; one simply denies what the other argues for. If it's the latter, then Hayes needs to provide arguments or examples to support his position.

A second thing to note about Hayes's definition is that he says that a computer "*performs* computations", not "*can* perform computations". Strictly speaking, your laptop when it is turned off is not a computer by this definition, because it is not performing any computation. And, as Hayes observes,

> On this understanding, a Turing machine is not a computer, but a mathematical abstraction of a certain kind of computer. —Hayes (1997, p. 390)

What about Searle's wall that implements Wordstar? There are two ways to think about how the wall might implement Wordstar. First, it might do so *statically*, simply in virtue of there being a way to map every part of the Wordstar program to some aspect of the molecular or subatomic structure of the wall. In that case, Hayes could well argue that the wall is *not* a Wordstar computer, because it is not computing (even if it *might be able to*). But the wall might implement Wordstar *dynamically*; in fact, that is why Searle thinks that the wall implements Wordstar . . .

> . . . because there is some pattern of molecule *movements* which is isomorphic with the formal structure of Wordstar.
> —Searle (1990, p. 27, my italics); Searle 1992, pp. 208–209

[13] A dishwasher might, however, be described by a (non-computable?) function that takes dirty dishes as input and that returns clean ones as output. Aaronson 2012 considers (semi-humorously) a "toaster-enhanced Turing machine".

But a pattern of *movements* suggests that Searle thinks that the wall *is computing*, so it *is* a computer!

Hayes's second definition is a bit more precise, and it is, presumably, his "official" one:

Definition H2

> [F]ocus on the memory. A computer's memory contains patterns ... which are stable but labile [that is, changeable], and it has the rather special property that changes to the patterns are under the control of other patterns: that is, some of them describe changes to be made to others; and when they do, the memory changes those patterns in the way described by the first ones. ... A computer is *a machine which is so constructed that patterns can be put in it, and when they are, the changes they describe will in fact occur to them.* If it were paper, it would be "magic paper" on which writing might spontaneously change, or new writing appear.
>
> —Hayes (1997, p. 393, my italics and bracketed interpolation)

There is a subtle difference between Hayes's two definitions, which highlights an ambiguity in Searle's presentation. Recall the distinction between a Turing machine and a universal Turing machine: Both Turing machines and universal Turing machines are hardwired and compute only a single function. The Turing machine computes whichever function is encoded in its machine table; it cannot compute anything else. But the one function, hardwired into its machine table, that a universal Turing machine computes is the fetch-execute function that takes as input a program and its data, and that outputs the result of executing that program on that data. In that way, a universal Turing machine (besides computing the fetch-execute cycle) can (in a different way) compute any computable function as long as a Turing machine program for that function is encoded and stored on the universal Turing machine's tape. The universal Turing machine is programmable in the sense that the input program can be varied, not that its hardwired program can be.

Definition H1 seems to include physical Turing machines (but, as he noted, not abstract ones), because, after all, they compute (at least, when they are turned on and running). Definition H2 seems to *exclude* them, because the second definition requires patterns that describe changes to other patterns. That first kind of pattern is a stored program; the second kind is the data that the program operates on. So, *Definition H2 is for a universal Turing machine.*

Here is the ambiguity in Searle's presentation: Is Searle's wall a Turing machine or a universal Turing machine? On Searle's view, Wordstar is a Turing machine, so the wall must be a Turing machine, too. So, the wall is not a computer on Definition H2. Could a wall (or a rock, or some other suitably large or complex physical object other than something like a PC or a Mac) be a *universal* Turing machine? My guess is that Searle would say "yes", but it is hard to see how one would actually go about programming it.

The "magic paper" aspect of Definition H2 focuses, as Hayes notes, on the memory, that is, on the tape. It is as if you were looking at a universal Turing machine, but all you saw was the tape, not the read-write head or its states (*m*-configurations) or its mechanism. If you watch the universal Turing machine compute, you would see the patterns (the '0's and '1's) on the tape "magically" change. (This would be something

like looking at an animation of the successive states of the Turing-machine tape in §8.11.2.2, p. 345.)

A slightly different version of the "magic paper" idea is Alan Kay's third "computing whammy" (see §7.5.5):

> Matter can hold and interpret and act on descriptions that describe anything that matter can do. —Guzdial and Kay (2010)

The idea of a computer as magic paper or magic matter may seem a bit fantastic. But there are more down-to-earth ways of thinking about this. Philosopher Richmond Thomason has said that

> ... all that a program can do between receiving an input and producing an output is to change variable assignments ...—Thomason (2003, p. 328).

A similar point is made by Turing Award winner Leslie Lamport:

> [A]n execution of an algorithm is a sequence of states, where a state is an assignment of values to variables. —Lamport (2011, p. 6)

If programs tell a computer how to change the assignments of values to variables, then a computer is a (physical) device that changes the contents of register cells (the register cells that are the physical implementations of the variables in the program). This is really just another version of Turing's machines, if you consider the tape squares to be the register cells.

Similarly, Stuart C. Shapiro points out that

> a computer is a device consisting of a vast number of connected switches. ... [T]he switch settings both determine the operation of the device and can be changed by the operation of the device. —Shapiro (2001, p. 3)

What is a "switch"? Here is a nice description from Samuel's 1953 article:

> To bring the discussion down to earth let us consider the ordinary electric light switch in your home. This is by definition a switch. It enables one to direct electric current to a lighting fixture at will. Usually there is a detent mechanism [see below] which enables the switch to remember what it is supposed to be doing so that once you turn the lights on they will remain on. It therefore has a memory. It is also a binary, or perhaps we should say a bistable device. By way of contrast, the ordinary telegrapher's key is a switch without memory since the key will remain down only as long as it is depressed by the operator's hand. But the light switch and the telegraph key are binary devices, that is, they have but two operating states. —(Samuel, 1953, p. 1225, my bracketed interpolation)

Further Reading:
A "detent" is "a catch in a machine that prevents motion until released" (https://www.google.com/search?q=detent). For more on computers as switch-setting devices, see the discussions in Stewart 1994 and Brian Hayes 2007b of how train switches can implement computations. Both of these are also examples of Turing machines implemented in very different media than silicon (namely, trains)!

So, a switch is a physical implementation of a Turing-machine's tape cell, which can also be "in two states" (that is, have one of two symbols printed on it) and also has a "memory" (that is, once a symbol is printed on a cell, it remains there until it is changed). Hayes's magic-paper patterns are just Shapiro's switch-settings or Thomason's and Lamport's variable assignments.

Does this definition satisfy Hayes's two criteria? Surely, such machines exist. I am writing this book on one of them right now. And surely not everything is such a machine: At least on the face of it, the stapler on my desk is not such "magic paper". Searle, I would imagine, would say that we might see it as such magic paper if we looked at it closely enough and in just the right way. And so the difference between Searle and Hayes seems to be in how one is supposed to look at candidates for being a computer: Do we look at them as we normally do? In that case, not everything is a computer. Or do we squint our eyes and look at them closely in a certain way? In that case, perhaps we could see that everything could be considered to be a computer. Isn't that a rather odd way of thinking about things?

What about the brain? Is it a computer in the sense of "magic paper" (or magic matter)? If Hayes's "patterns" are understood as patterns of neuron firings, then, because surely some patterns of neuron firings cause changes in other such patterns, I think Hayes would consider the brain to be a computer.

9.7 Gualtiero Piccinini: Computers as Digital String Manipulators

In a series of three papers, the philosopher Gualtiero Piccinini has offered an analysis of what a computer is that is more precise than Hayes's and less universal than Searle's (Piccinini, 2007b,d, 2008) (see also Piccinini 2015). It is more precise than Hayes's, because it talks about *how* the magic paper performs its tricks. And it is less universal than Searle's, because Piccinini doesn't think that everything is a computer.

Unfortunately, there are two slightly different definitions to be found in Piccinini's papers:

Definition P1

> The mathematical theory of how to generate output strings from input strings in accordance with general rules that apply to all input strings and depend on the inputs (and sometimes internal states) for their application is called computability theory. Within computability theory, the activity of manipulating strings of digits in this way is called computation. *Any system that performs this kind of activity is a computing system properly so called.* —Piccinini (2007b, p. 108, my italics)

Definition P2

> *[A]ny system whose correct mechanistic explanation ascribes to it the function of generating output strings from input strings (and possibly internal states), in accordance with a general rule that applies to all strings and depends on the input strings (and possibly internal states) for its application, is a computing mechanism.* The mechanism's ability

> to perform computations is explained mechanistically in terms of its components, their functions, and their organization. —Piccinini (2007d, p. 516, my italics)

These are almost the same, but there is a subtle difference between them.

9.7.1 Definition P1

Let's begin with Definition P1. It implies that a computer is any "system" (presumably, a physical device, because only something physical can actively "perform" an action) that manipulates strings of digits, that is, that "generate[s] output strings from input strings in accordance with general rules that apply to all input strings and [that] depend on the inputs (and sometimes internal states) for their application". What kind of "general rule"? Piccinini (2008, p. 37) uses the term 'algorithm' instead of 'general rule'. This is consistent with the view that a computer is a Turing machine, and explicates Hayes's "magic trick" as being an algorithm.

The crucial point, according to Piccinini, is that the inputs and outputs must be strings of digits. This is the significant difference between (digital) computers and "analog" computers: The former manipulate strings of digits; the latter manipulate "real variables".

Piccinini explicates the difference between digits and real variables as follows:

> A *digit* is a particular [that is, a particular object or component of a device] or a discrete state of a particular, discrete in the sense that it belongs to one (and only one) of a finite number of types. ... A *string* of digits is a concatenation of digits, namely, a structure that is individuated by the types of digits that compose it, their number, and their ordering (i.e., which digit token is first, which is its successor, and so on). —Piccinini (2007b, p. 107, my bracketed interpolation)[14]

Piccinini (2007d, p. 510) observes that a digit is analogous to a letter of an alphabet, so they are like Turing's symbols that can be printed on a Turing machine's tape.

On the other hand,

> real variables are physical magnitudes that (i) *vary* over time, (ii) (are assumed to) take a *continuous range of values* within certain bounds, and (iii) (are assumed to) vary *continuously* over time. Examples of real variables include the rate of rotation of a mechanical shaft and the voltage level in an electrical wire.
> —Piccinini (2008, p. 48)

So far, so good. Neither Searle nor Hayes should be upset with this characterization.

Further Reading:
For a different approach to the computation of real numbers, see Blum 2004, which is a relatively informal and historical presentation of a more technical paper (Blum et al., 1989, described briefly in Traub 2011).

[14] In the other two papers in his trilogy, Piccinini gives slightly different characterizations of what a digit is, but these need not concern us here; see Piccinini 2007d, p. 510; Piccinini 2008, p. 34.

9.7.2 Definition P2

But Piccinini's second definition adds a curious phrase. This definition implies that a computer is any system "whose correct mechanistic explanation ascribes to it the function of" manipulating digit strings according to algorithms. What is the import of that extra phrase?

It certainly sounds as if this is a weaker definition. In fact, it sounds a bit Searlean, because it sounds as if it is not the case that a computer *is* an algorithmic, digit-string manipulator, but rather that it is anything that can be so *described* by some kind of "mechanistic explanation". And that sounds as if being a computer is something "external" and not "intrinsic".

So let's consider what Piccinini has in mind here. He says:

> Roughly, a mechanistic explanation involves a partition of a mechanism into parts, an assignment of functions and organization to those parts, and a statement that a mechanism's capacities are due to the way the parts and their functions are organized. —Piccinini (2007d, p. 502)

As we will explain in more detail in Chapter 22, syntax in its most general sense is the study of the properties of a collection of objects and the relations among them. If a "mechanism" is considered as a collection of its parts, then Piccinini's notion of a mechanistic explanation sounds a lot like a description of the mechanism's syntax. But syntax, you will recall, is what Searle says is not intrinsic to a system (or a mechanism).

So how is Piccinini going to avoid a Searlean "slippery slope" and deny that everything is a computer? One way he tries to do this is by suggesting that even if a system can be analyzed syntactically in different ways, only one of those ways will help us understand the system's behavior:

> Mechanistic descriptions are sometimes said to be perspectival, in the sense that the same component or activity may be seen as part of different mechanisms depending on which phenomenon is being explained .... For instance, the heart may be said to be for pumping blood as part of an explanation of blood circulation, or it may be said to be for generating rhythmic noises as part of an explanation of physicians who diagnose patients by listening to their hearts. This kind of perspectivalism does not trivialize mechanistic descriptions. Once we fix the phenomenon to be explained, the question of what explains the phenomenon has an objective answer. This applies to computations as well as other capacities of mechanisms. A heart makes the same noises regardless of whether a physician is interested in hearing it or anyone is interested in explaining medical diagnosis.
> —Piccinini (2007d, p. 516)

Let's try to apply this to Searle's "Wordstar wall": From one perspective, the wall is just a wall; from another, according to Searle, it can be taken as an implementation of Wordstar. Compare this to Piccinini's claim that, from one perspective, a heart is a pump, and, from another, it is a noisemaker. If you're a doctor interested in hearing the heart's noises, you'll consider the heart as a noisemaker. If you're a doctor interested in making a medical diagnosis, you'll consider it as a pump. Similarly, if you're a house painter, say, you'll consider the wall as a flat surface to be colored, but if you're Searle,

you'll try to consider it as a computer program. (Although I don't think you'll be very successful in using it to write a term paper!)

9.8 What Else Might Be a Computer?

So, what is a computer? It would seem that almost all proposed definitions agree on at least the following:

- Computers are physical devices.
- They interact with other physical devices in the world.
- They algorithmically manipulate (physical) symbols (strings of digits), converting some into others.
- They are physical implementations of (universal) Turing machines in the sense that their input-output behavior is logically equivalent to that of a (universal) Turing machine (even though the details of their processing might not be). A slight modification of this might be necessary to avoid the possibility that a physical device might be considered to be a computer even if it doesn't compute: We probably want to rule out "real magic", for instance.

Does such a definition include too much? Let's assume for a moment that something like Piccinini's reply to Searle carries the day, so that it makes sense to say that not everything is a computer. Still, might there be some things that intuitively aren't computers but that turn out to be computers on even our narrow characterization?

This is always a possibility. As we saw in §3.3.3.2.1, any time that you try to make an informal concept precise, you run the risk of *including* some things under the precise concept that didn't (seem to) fall under the informal concept. You also run the risk of *excluding* some things that did. One way to react to this situation is to reject the formalization, or else to refine it so as to minimize or eliminate the "erroneous" inclusions and exclusions. But another reaction is to bite the bullet and agree to the new inclusions and exclusions: For instance, you might even come to see that something that you didn't think was a computer really was one.

In this section, we'll consider two things that may—or may not!—turn out to be computers: the brain, and the universe.

9.8.1 Is a Brain a Computer?

> [I]t is conceivable ... that brain physiology would advance so far that it would be known with empirical certainty
>
> 1. that the brain suffices for the explanation of all mental phenomena and is a machine in the sense of Turing;
> 2. that such and such is the precise anatomical structure and physiological functioning of the part of the brain which performs mathematical thinking.
>
> —Gödel, 1951; cited in Feferman 2006a, p. 146

> It is the current aim to replace, as far as possible, the human brain by an electronic digital computer. —Hopper (1952, p. 243)

> Perhaps the most intriguing examples of reactive distributed computing systems are biological systems such as cells and organisms. *We could even consider the human brain to be a biological computing system.* Formulation of appropriate models of computation for understanding biological processes is a formidable scientific challenge in the intersection of biology and computer science.
> —Aho (2011, p. 6, my italics)[15]

> The first computers were biological: they had two arms, two legs and 10 fingers. "Computer" was a job title, not the name of a machine.
> —Lu and Purcell (2016, p. 59)

We saw in §3.8 that Piccinini (2015) distinguishes computation from information processing; in particular, "Piccinini argues that systems can compute without processing information" (Shagrir, 2017, p. 607). Shagrir also notes that "A working assumption in brain and cognitive sciences is that the brain is a representational, information-processing system" (Shagrir, 2012c, p. 519). So, even if the brain is an information-processing system, it doesn't follow that it must be a computer.

Still, many people claim that the (human) brain *is* a computer. Searle thinks it is, but only because he thinks that everything is a computer. But perhaps there is a more interesting way in which the brain is a computer. Certainly, contemporary computational cognitive science uses computers as at least a metaphor for the brain. Before computers came along, there were many other physical metaphors for the brain: The brain was considered to be like a telephone system or like a plumbing system.

Further Reading:
Long before computers, the brain and nervous system were also likened to an electronic communications network, as in Fritz Kahn's artwork (https://www.nlm.nih.gov/dreamanatomy/da_g_IV-A-02.html). There are also analogies between telephone systems and computers themselves (Lewis, 1953). On metaphors for the brain, see (Squires, 1970), (Sternberg, 1990), (Gigerenzer and Goldstein, 1996), (Angier, 2010), and "The Mind is a Metaphor" website at http://mind.textdriven.com/?q=mind. (Concerning that website, see (Guernsey, 2009)].)

[15] We will discuss interactive computing in Ch. 11.

In fact, "computationalism" is sometimes taken to be the view that the brain (or the mind) *is* a computer, or that the brain (or the mind) computes, or that brain (or mental) states and processes *are* computational states and processes (Rapaport, 2012b, §2):

> The basic idea of the computer model of the mind is that the mind is the program and the brain the hardware of a computational system.
> —Searle (1990, p. 21); Searle 1992, p. 200

> The core idea of cognitive science is that our brains are a kind of computer .... Psychologists try to find out exactly what kinds of programs our brains use, and how our brains implement those programs. —Alison Gopnik (2009a, p. 43)

> Computationalism ... is the view that the functional organization of the brain (or any other functionally equivalent system) is computational, or that neural states are computational states.
> —Piccinini (2010a, p. 271; see also pp. 277–278)

Gödel thought so, too: He "view[ed] it as very likely that 'The brain functions basically like a digital computer' " (Sieg, 2007, §2).

But if one of the essential features of a computer is that it carries out computable processes by *computing* rather than (say) by some biological but non-computational technique, then it's at least logically possible that the brain is not a computer even if brain processes are computable.

How can this be? A process is computable if and only if there is an algorithm (or a system of algorithms) that specifies how that process can be carried out. But it is logically possible for a process to be computable in this sense without actually being computed.

Here are some examples:

1. Someone might come up with a computational theory of the behavior of the stock market, yet the actual stock market's behavior is determined by the individual decisions made by individual investors and not by anyone or anything executing an algorithm. That is, the behavior might be comput*able* even if it is not comput*ational*.

2. Calculations done by slide rules are done by analog means, yet the calculations themselves are clearly computable. Analog computations are not normally considered to be Turing-machine computations.

3. Hayes's magic paper is a logically, if not physically, possible example.

4. Another example might be the brain itself. Piccinini has argued that neuron firings (more specifically, "spike trains"—i.e., sequences of "action potential"—in groups of neurons) are not representable as digit strings (Piccinini, 2005, 2007a). But, because Piccinini believes that a device is not a computer unless it manipulates digit strings, and because it is generally considered that human cognition is implemented by neuron firings, it follows that the brain's cognitive functioning—even if comput*able*—is not accomplished by computa*tion*. Yet, if cognitive functions are computable (as contemporary cognitive science suggests—see Edelman

> 2008a), then there would still be algorithms that compute cognition, even if the brain doesn't do it that way.

We'll return to this theme in Chapter 19.

The philosopher David Chalmers puts the point this way:

> Is the brain a [programmable] computer ... ? Arguably. For a start, the brain can be "programmed" to implement various computations by the laborious means of conscious serial rule-following; but this is a fairly incidental ability. On a different level, it might be argued that learning provides a certain kind of programmability and parameter-setting, but this is a sufficiently indirect kind of parameter-setting that it might be argued that it does not qualify. In any case, the question is quite unimportant for our purposes. What counts is that the brain implements various complex computations, not that it is a computer.
>
> —Chalmers (2011, §2.2, esp. p. 336, my bracketed interpolation)

There are two interesting points made here. The first is that the brain can simulate a Turing machine "by ... conscious serial rule-following". The second is the last sentence: What really matters is that the brain can have input-output behavior that is computable, not that it "is" a computer. To say that it is a computer raises the question of what kind of computer it is: A Turing machine? A register machine? Something *sui generis*? And these questions seem to be of less interest than the fact that its behavior is computable.

Churchland and Sejnowski (1992) and Ballard (1997) have both written books about whether, and in what ways, the brain might be a computer. As Ballard (1997, pp. 1–2) puts it, "The key question ... is, Is computation sufficient to model the brain?". One reason this is an interesting question is that researchers in vision have wondered "how ... an incomplete description, encoded within neural states, [could] be sufficient to direct the survival and successful adaptive behavior of a living system" (Richards, 1988, as cited in Ballard 1997, p. 2). If a computational model of this ability is sufficient, then it might also be sufficient to model the brain. And this might be the case even if, as, for example, Piccinini and Bahar (2013) argue, the brain itself is *not* a computer, that is, does not behave in a computational fashion. A model of a phenomenon does not need to be identical in all respects to the phenomenon that it models, as long as it serves the purposes of the modeling. But Ballard also makes the stronger claim when he says, a few pages later, "*If the brain is performing computation*, it should obey the laws of computational theory" (Ballard, 1997, p. 6, my italics). But whether the brain performs computations is a different question from whether its performance can be modeled or described in computational terms. So, the brain doesn't have to *be* a computer in order for its behavior to be describable computationally. As Churchland and Sejnowski (1992) note, whether the brain *is* a computer—whether, that is, the brain's functioning satisfies one of the (logically equivalent) characterizations of computing—is an empirical issue.

Still, if the brain computes in some way (or "implements computations"), and if a computer is, by definition, something that computes, then we might still wonder if the brain is some kind of computer. As I once read somewhere, "The best current explanation of how a brain could instantiate this kind of system of rules and representations is that it is a kind of computer." Thus, we have here the makings of an abductive argu-

ment (that is, a scientific hypothesis) that the brain is a computer.[16] Note that this is a much more reasonable argument than Searle's or than trying to model the brain as, say, a Turing machine. And, as Marcus (2015) observes, "For most neuroscientists, this is just a bad metaphor. But it's still the most useful analogy that we have. ... The sooner we can figure out what kind of computer the brain is, the better."

Question for the Reader:
If the brain is a computer, is its "data" propositional or pictorial? For example, if you are asked how many windows are in your house, do you form a mental pictorial image of your house, or do you do the calculation in terms of numerical propositions? (For the debate on this in the cognitive-science literature, see Pylyshyn 1973, 2003; Kosslyn 2005; and the bibliography at https://www.cse.buffalo.edu//~rapaport/575/mentalimages.html.)

Further Reading:
Fitch 2005 is a good discussion by a cognitive biologist of "how the brain computes the mind" (from the Introduction).

Zenil and Hernández-Quiroz 2007 investigates the computational power of the brain and whether the brain might be a "hypercomputer" (see Ch, 11).

For a reply to Marcus (2015), see Linker 2015. Naur 2007 says that "the nervous system ... has no similarity whatever to a computer" (p. 85); and Schulman 2009 says that *minds* are not like computers.

For more on "the brain as an input-output model of the world", see Shagrir 2018a.

It is one thing to argue that brains are (or are not) computers of some kind. It is quite another to argue that they are Turing machines, in particular. The earliest suggestion to that effect is McCulloch and Pitts 1943. For a critical and historical review of that classic paper, see Piccinini 2004a. More recently, the cognitive neuroscientist Stanislas Dehaene and his colleagues have made similar arguments; see Sackur and Dehaene 2009 and Zylberberg et al. 2011.

See also Kuczynski 2015, part of a series of online articles providing background for the movie *The Imitation Game*. In order to investigate whether the human brain is a computer, he reviews Turing 1936. Among the interesting, if controversial, points that he makes are that "a recursive function is one that is defined for each of its own outputs" and that the recursive definitions of addition, multiplication, and exponentiation are the reason that "arithmetic requires no thought at all" (reminiscent of Dennett 2009a and Dennett 2013b's notion of Turing's "inversion" that we will discuss later).

[16] Recall our discussion of abduction in Ch. 2, §2.6.1.3.

9.8.2 Is the Universe a Computer?

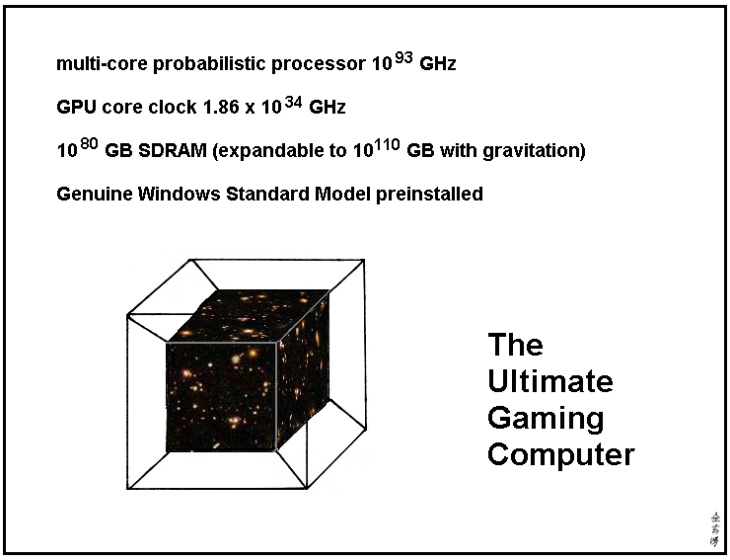


Figure 9.3: http://abstrusegoose.com/219

Might the universe itself be a computer?[17] Consider Kepler's laws of planetary motion. Are they just a computable theory that describes the behavior of the solar system? If so, then a computer that calculates with them might be said to *simulate* the solar system in the same way that any kind of program might be said to simulate a physical (or biological, or economic) process, or in the same way that an AI program might be said to simulate a cognitive process. (We'll return to this idea in Chapter 15.)

Or does *the solar system itself* compute Kepler's laws? If so, then the solar system would seem to be a (special purpose) computer (that is, a kind of Turing machine):

> A computation is a process that establishes a mapping among some symbolic domains. ... Because it involves symbols, this definition is very broad: a system instantiates a computation if its dynamics can be interpreted (by another process) as establishing the right kind of mapping.
>
> Under this definition, a stone rolling down a hillside computes its position and velocity in exactly the same sense that my notebook computes the position and the velocity of the mouse cursor on the screen (they just happen to be instantiating different symbolic mappings). Indeed, the universe in its entirety also instantiates a computation, albeit one that goes to waste for the lack of any process external to it that would make sense of what it is up to. —Edelman (2008b, pp. 182–183)

After all, if "*biological computation* is a process that occurs in nature, not merely

[17] In addition to the cartoon in Fig. 9.3, see also the satirical Google search page"Is the Universe a Computer?", http://abstrusegoose.com/115 (best viewed online!).

in computer simulations of nature" (Mitchell, 2011, p. 2), then it is at least not unreasonable that the solar system computes Kepler's Laws:

> Going further along the path of nature, suppose that we have a detailed mathematical model of some physical process such as—say—a chemical reaction; clearly we can either organise the reaction in the laboratory and observe the outcome, or we can set up the mathematical model of the reaction on a computer either as the numerical solution of a system of equations, or as a Montecarlo simulation, and we can then observe the outcome. We can all agree that when we "run the reaction" on the computer either as a numerical solution or a Montecarlo simulation, we are dealing with a computation.
>
> But why then not also consider that the laboratory experiment itself is after all only a "computational analogue" of the numerical computer experiment! In fact, the laboratory experiment will be a mixed analogue and digital phenomenon because of the actual discrete number of molecules involved, even though we may not know their number exactly. In this case, the "hardware" used for the computation are the molecules and the physical environment that they are placed in, while the software is also inscribed in the different molecules species that are involved in the reaction, via their propensities to react with each other ....
>
> —Gelenbe (2011, pp.3–4)

This second possibility does not necessarily follow from the first. As we just saw in the case of the brain, there might be a computational *theory* of some phenomenon—that is, the phenomenon might be *computable*—but the phenomenon *itself* need not be produced computationally.

> Indeed, *computational algorithms are so powerful that they can simulate virtually any phenomena, without proving anything about the computational nature of the actual mechanisms underlying these phenomena.* Computational algorithms generate a perfect description of the rotation of the planets around the sun, although the solar system does not compute in any way. In order to be considered as providing a model of the mechanisms actually involved, and not only a simulation of the end-product of mechanisms acting at a different level, computational models have to perform better than alternative, noncomputational explanations.
>
> —Perruchet and Vinter (2002, §1.3.4, p. 300, my italics)

Nevertheless, could it be the case that our solar system *is* computing Kepler's laws? Arguments along these lines have been put forth by Stephen Wolfram and by Seth Lloyd.

Further Reading:
For a different take on the question of whether the solar system computes Kepler's laws of motion, in the context of "pancomputationalsim" (the view that "every *deterministic* physical system computes *some* function"), see Campbell and Yang 2019.

9.8.2.1 Wolfram's Argument

Wolfram, developer of the Mathematica computer program, argues as follows (Wolfram, 2002b):

1. Nature is discrete.
2. Therefore, possibly it is a cellular automaton.
3. There are cellular automata that are equivalent to a Turing machine.
4. Therefore, possibly the universe is a computer.

There are a number of problems with this argument. First, why should we believe that nature (that is, the universe) is discrete? Presumably, because quantum mechanics says that it is. But some distinguished physicists deny this (Weinberg, 2002) (but see Chaitin 2006a for more on this). So, at best, for those of us who are not physicists able to take a stand on this issue, Wolfram's conclusion has to be conditional: If the universe is discrete, then possibly it is a computer.

So let's suppose (for the sake of the argument) that nature *is* discrete. Might it be a "cellular automaton"? The easiest way to think of a cellular automaton is as a two-dimensional Turing-machine tape for which the symbol in any cell is a function of the symbols in neighboring cells (https://en.wikipedia.org/wiki/Cellular_automaton). But, of course, even if a discrete universe *might* be a cellular automaton, it *need not* be. If it isn't, the argument stops here. But, if it is, then—because the third premise is mathematically true[18]—the conclusion follows validly from the premises. Premise 2 is the one most in need of justification. But even if all of the premises and (hence) the conclusion are true, it is not clear what philosophical consequences we are supposed to draw from this.

Further Reading:
For more information on cellular automata, see the relatively informal presentation in Bernhardt 2016, Ch. 5 and the more formal presentation in Burks 1970. (Philosopher and mathematician Arthur Burks was one of the people involved in the construction of ENIAC and EDVAC.)

You can read more about Wolfram and his theories at his homepage, http://www.stephenwolfram.com/, and in Wolfram 2002a and Bernhardt 2016, Chs. 5, 6. For a critical review, see Weinberg 2002. Aaronson 2011 claims that quantum computing has "overthrown" views like those of Wolfram (2002b) that "the universe itself is basically a giant computer ... by showing that if [it is, then] it's a vastly more powerful kind of computer than any yet constructed by humankind."

[18] See, for example, https://en.wikipedia.org/wiki/Turing_completeness, https://en.wikipedia.org/wiki/Rule_110, and https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

9.8.2.2 Lloyd's Argument

Seth Lloyd also argues that the universe is a computer because nature is discrete, but Lloyd's intermediate premises differ from Wolfram's. Lloyd argues as follows (Lloyd and Ng, 2004):

1. Nature is discrete. (This is "the central maxim of quantum mechanics" (p. 54).)
2. In particular, elementary particles have a "spin axis" that can be in one of two directions.
3. ∴ They encode a bit.
4. ∴ Elementary particles store bits of information.
5. Interactions between particles can flip the spin axis;
   this transforms the stored data—that is, these interactions are operations on the data.
6. ∴ (Because any physical system stores and processes information,)
   all physical systems are computers.
7. In particular, a rock is a computer.
8. Also, the entire universe is a computer.

Premise 1 matches Wolfram's fundamental premise and would seem to be a necessity for anything to be considered a digital computer. The next four premises also underlie quantum computing.

But the most serious problem with Lloyd's argument as presented here is premise 6. Is the processing sufficient to be considered to be Turing-machine-equivalent computation? Perhaps; after all, it seems that all that is happening is that cells change from 0s to 1s and vice versa. But that's not all that's involved in computing. (Or is it? Isn't that what Hayes's magic-paper hypothesis says?) What about the control structures—the grammar—of the computation?

And although Lloyd wants to conclude that everything in the universe (including the universe itself!) is a computer, note that this is not exactly the same as Searle's version of that claim. For Searle, everything can be interpreted as *any* computer program. For Lloyd, anything is a computer, "although they may not accept input or give output in a form that is meaningful to humans" (p. 55). So, for Lloyd, it's not a matter of interpretation. Moreover, "analyzing the universe in terms of bits and bytes does not replace analyzing it in conventional terms such as force and energy" (p. 54). It's not clear what the import of that is: Does he mean that the computer analysis is irrelevant? Probably not: "it does uncover new and surprising facts" (p. 54), though he is vague (in this general-audience magazine article) on what those "facts" are. Does he mean that there are different ways to understand a given object? (An object could be understood as a computer or as an object subject to the laws of physics.) That is true, but unsurprising (animals, for instance, can be understood as physical objects satisfying the laws of quantum mechanics as well as being understood as biological objects). Does

he mean that force and energy can, or should, be understood in terms of the underlying computational nature of physical objects? He doesn't say.

But Lloyd does end with a speculation on what it is that the universe is computing, namely, itself! Or, as he puts it, "computation is existence" (p. 61). As mystical as this sounds, does it mean anything different from the claim that the solar system computes Kepler's Law?

And here's an interesting puzzle for Lloyd's view, relating it to issues concerning whether a computer must halt (recall our earlier discussion in Chapters 7 and 8):

> [A]ssuming the universe is computing its own evolution . . . , does it have a finite lifetime or not? If it is infinite, then its self-computation won't get done; it never produces an answer . . . . Hence, it does not qualify as a computation.
> —Borbely (2005, p. 15)

Of course, Turing—as we saw in §8.10.3.1—would not have considered this to be a problem: Don't forget that his original *a*-machines only computed the decimal expansions of real numbers by *not* halting!

Further Reading:
Lloyd 2000 investigates "quantitative bounds to the computational power of an 'ultimate laptop' with a mass of one kilogram confined to a volume of one litre." Lloyd 2002 argues that "All physical systems register and process information. The laws of physics determine the amount of information that a physical system can register (number of bits) and the number of elementary logic operations that a system can perform (number of ops). The Universe is a physical system. The amount of information that the Universe can register and the number of elementary operations that it can have performed over its history are calculated. The Universe can have performed 10^{120} ops on 10^{90} bits (10^{120} bits including gravitational degrees of freedom)." And see Lloyd 2006, reviewed by Powell 2006 and Schmidhuber 2006 (cited in "Strongly Recommended" readings, above).

Konrad Zuse (whom we mentioned in §6.5.4) also argued that the universe is a computer; see Schmidhuber (2002).

Chaitin (2006a) argues that "everything is software, God is a computer programmer, . . . and the world is . . . a giant computer".

See also Bacon (2010).

Related to Lloyd is Bostrom's work on whether we are living in a computer simulation. If Lloyd is right, then the universe is a computer, and we are data structures in its program, brought to life as it were by its execution. If Bostrom is right, then we are data structures in someone else's program. If theists (computational theists?) are right, then we are data structures in God's program. See Bostrom 2003; for an argument that simulation theories are not scientific, see Dunning 2018.

9.9 Conclusion

So, finally, what is a computer?

At a bare minimum, we might say that a (programmable) computer is a physically plausible implementation (including a virtual implementation) of anything logically equivalent to a universal Turing machine (DC4, above). Most of the definitions that we discussed above might best be viewed as focusing on exactly what is meant by 'implementation' or which entities count as such implementations. This is something that we will return to in Chapter 14.

Two kinds of (alleged) computers are not obviously included in this sort of definition: analog computers and "hypercomputers". Because most discussions focus on "digital" computers as opposed to analog ones, I have not considered analog computers here. By 'hypercomputer', I have in mind any physical implementation (assuming that there are any) of anything capable of "hypercomputation", i.e., anything capable of "going beyond the Turing limit", that is, anything that "violates" the Church-Turing Computability Thesis.

The topics of hypercomputation and counterexamples to the Computability Thesis will be discussed in Chapters 10 and 11. But one way to incorporate these other models of computation into a unified defintion of 'computer' might be this:

(DC5) A computer is any physically plausible implementation of anything that is *at least* logically equivalent to a universal Turing machine.

In other words, if something can compute at least all Turing-computable functions, but might also be able to perform analog computations or hypercomputations, then it, too, is a computer. A possible objection to this is that an adding machine, or a calculator, or a machine that is designed to do only *sub*-Turing computation, such as a physical implementation of a finite automaton, has at least some claim to being called a 'computer'.

So another way to incorporate all such models is to go one step beyond our DC5 to:

(DC6) A computer is a physically plausible implementation of some model of computation.

Indeed, Piccinini (2018, p. 2) has more recently offered a definition along these lines. He defines 'computation' as "the processing of medium independent vehicles by a functional mechanism in accordance with a rule." (See Piccinini 2015, Ch. 7 for argumentation and more details.) This, of course, is a definition of 'computation', not 'computer'. But we can turn it inside out to get this:

Definition P3

A computer is a functional mechanism that processes medium-independent vehicles in accordance with a rule.

He explicitly cites as an advantage of this very broad definition its inclusion of "not only digital but also analog and other unconventional types of computation" (p. 3)—including hypercomputation. But Piccinini (2015, Chs. 15 & 16) also distinguishes

between the "mathematical" Church-Turing Computability Thesis and a "modest physical" thesis: "Any function that is physically computable is Turing-computable" (Piccinini, 2015, p. 264), and he argues that it is an "open empirical question" (p. 273) whether hypercomputers are possible (although he doubts that they are).

Recall Stuart C. Shapiro's definition, cited in §3.9.3:

> [The computer is a general-purpose procedure-following machine.
> —Shapiro (2001, p. 2)

Given his broad characterization of 'procedure', this fits with DC6. My only hesitation with these last three definitions is that they seem to be a bit too vague in their generosity, leaving all the work to the meaning of 'computation' or 'procedure' or 'rule'. But maybe that's exactly right. Despite its engineering history and despite its name, perhaps "computer science" is best viewed as the scientific study of computation, not (just) computers. As we saw in §3.15.2, computer science can be thought of as the scientific study of what problems can be solved, what tasks can be accomplished, and what features of the world can be understood computationally, and then to provide algorithms to show how this can be done efficiently, practically, physically, and ethically. Determining how computation can be done *physically* tells us what a computer is.

9.10 Questions for the Reader

1. (This exercise was developed by Albert Goldfain.)

   (a) The following arguments are interesting to think about in relation to the question whether everything a computer. Try to evaluate them.

   Argument 1

   P1 A Turing machine is a model of computation based on what a single human (that is, a clerk) does.

   P2 Finite automata and push-down automata are mathematical models of computation that recognize regular languages and context-free languages, respectively.

   P3 Recognizing strings in a languages is also something individual humans do.

   C1 $\therefore$ Turing machines, finite automata, and push-down automata are all models of computation based on the abilities of individuals.

   Argument 2

   John Conway's "Game of Life" is a cellular-automaton model of a society (albeit a very simplistic one):[19]

   P1 The Game of Life can be implemented in Java.

   P2 Any Java program is reducible to a Turing-machine program.

   C1 $\therefore$ The Game of Life is Turing-machine computable

   Argument 3

   P1 The Game of Life can be thought of as a model of computation.

   P2 The Game of Life is a model of the abilities of a society.

   P3 The abilities of a society exceed those of an individual.

   C1 $\therefore$ The abilities of a model of computation based on a society will exceed the abilities of a model based on the abilities of an individual.

   C2 $\therefore$ It is not the case that every Turing-machine program could be translated to a Game-of-Life "computation".

   (b) Some of the arguments in Exercise 1 may have missing premises! To determine whether the Game of Life might be a model of computation, do a Google search using the two phrases: "game of life" "turing machine"

   (c) Given an integer input (remember: everything can be encoded as an integer), how could this integer be represented as live cells on an initial grid? How might "stable" structures (remember: a 2×2 grid has 3 neighbors each) be used as "memory"? How would an output be represented?

   (d) Can Turing-machine programs be reduced to Game-of-Life computations?

[19] See http://en.wikipedia.org/wiki/Conway_Game_of_Life for the rules of this game.

2. Recall our discussion in §6.5.3 of Jacquard's looms.

   > Modern programmers would say ... [that Jacquard] loom programs are not computer programs: looms could not compute mathematical functions.
   > —Denning and Martell (2015, p. 83)

   Looms might not have been computers, but could they have been? Even if we accept the definition of a computer (program) as one that computes mathematical functions, does it follow that Jacquard looms could not be computers? Could bits be implemented as patterns in looms?

3. Which physical processes are computing processes? Are all physical processes computations?[20] Of course, if a physical process is a computation, then, presumably, the physical object carrying out that process is a computer, so this question amounts to saying that all physical objects that carry out processes are computers.

4. One argument, adapted from (Fekete and Edelman, 2011), is this:

   (a) A process is a computation iff it operates on representations.

   (b) All physical processes can represent.

   (c) ∴ All physical processes are computations.

   Keep in mind that, even if all physical processes *can* represent, it does not follow that they all *do* represent. (Or does that suggest that "computing is in the eye of the beholder. If a rock heating up in the sun is not taken as a representer, then it is not computing, but if I use how hot it is to do something else, then the hot rock is representing and so computing."[21] Another consideration is this: Computation is done over uninterpreted marks. Whether those marks represent anything is a separate matter. I might choose to interpret them as representing something; or the computational system itself might choose to (self-?)interpret them as representing something (see (Schweizer, 2017)).

   Is this argument sound? Does this argument adequately represent Fekete & Edelman's actual argument? (See §7.9, #6.)

5. Is a (physical) implementation of a computation itself a computation?[22]
   (See our discussion of implementation in Chapter 14.)

6. > Never mind the name change—the Apple TV and iPhone are computers to the core. —(Gruber, 2007)

   Are devices such as these computers? Choose one or more definitions of 'computer' and see if Apple TVs, iPhones, etc., are computers on those definitions.

[20] Thanks to Russ Abbott and Eric Dietrich for suggesting these questions.
[21] Dietrich, personal communication, 28 June 2015.
[22] Also due to Dietrich.

7. Is a tree a computer? According to César Hidalgo, matter can can compute:

> A tree ... is a computer that knows in which direction to grow its roots and leaves. Trees know when to turn genes on and off to fight parasites, when to sprout or shed their leaves and how to harvest carbon from the air via photosynthesis. As a computer, a tree begets order in the macrostructure of its branches and the microstructures of its cells. We often fail to acknowledge trees as computers, but the fact is that trees contribute to the growth of information in our planet because they compute. —(Hidalgo, 2015, p. 75).

But what is his argument here? He doesn't seem to have a definition of 'computer'. Except for the last three words of the above quotation, one might think that his definition would be something like: A computer is an information-processing machine. Then his argument might go as follows: Trees are information-processing machines (because they "contribute to the growth of information"); hence, they are computers. But those last three words suggest that his argument goes the other way: that trees are computers; hence, they contribute to the growth of information.

Is the human race a computer? See the interview with Hidalgo in (O'Neill, 2015).

8. It seems to be correct to say that a real, physical computer such as your laptop is not a Turing machine, on the grounds that real, physical computers are finite devices (finite memory, etc.) whereas Turing machines are infinite (infinite, or at least arbitrarily long, tape, etc.).

But could it be a finite-state machine? After all, a finite-state machine is ... well ... finite!

At least one computer scientist has denied this:

> Another obvious distinction that is worth making explicit ... is the distinction between computers (which include laptops and iPads) on the one hand and their mathematical models on the other hand. Strictly speaking, then, it is wrong to say that:
>
> > A computer is a finite state machine.
>
> Once again, this is like speaking about a mathematical model (the finite state machine) as if it coincides with reality (the computer).
> —Daylight (2016, p. 14)

But consider this mathematical definition of a "graph" (paraphrased from https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)):

> ... a *graph* is an ordered pair $G = (V,E)$ comprising a set V of *vertices* ... together with a set E of edges ... which are ... [unordered pairs of members] of V (i.e., an edge is associated with two vertices, and the association takes the form of the unordered pair of the vertices).

Now consider a real-world computer network consisting of a set V of computers and a set E of pairs of computers that are networked to each other. *Is* that computer network a graph? Or is it only *modeled* as a graph?

Similarly, could we say that a real, physical computer *is* a finite-state machine if it satisfies the definition of one? It may also have other properties that the (mathematical) definition of finite-state machine lacks. For example, the computer might be made of plastic and silicon; the definition of a finite-state machine is silent about any requirements for physical composition:

> ... equating a laptop with a universal Turing machine is problematic, not primarily because the former is finite and the latter is infinite, but because the former moves when you push it and smells when you burn it while the latter can neither be displaced nor destroyed. —Daylight (2016, p. 118)

But all properties of finite-state machines will hold of physical computers, even if there are properties of physical computers that do not hold of finite-state machines (such as ringing—or failing to ring!—a real bell if its program has a 'BEEP' command).

Part III

The Church-Turing Computability Thesis

A function is said to be "effectively calculable" if its values can be found by some purely mechanical process. Although it is fairly easy to get an intuitive grasp of this idea, it is nevertheless desirable to have some more definite, mathematically expressible definition. Such a definition was first given by Gödel at Princeton in 1934 .... These functions were described as "general recursive" by Gödel. We shall not be much concerned here with this particular definition. Another definition of effective calculability has been given by Church ..., who identifies it with λ-definability. The author has recently suggested a definition corresponding more closely to the intuitive idea .... It was stated above that "a function is effectively calculable if its values can be found by some purely mechanical process". We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability† with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions are equivalent ....

---

† We shall use the expression "computable function" to mean a function calculable by a machine, and we let "effectively calculable" refer to the intuitive idea without particular identification with any one of these definitions. We do not restrict the values taken by a computable function to be natural numbers; we may for instance have computable propositional functions.

—Turing (1939, p. 166)

Chapter 10

What Is a Procedure?

Version of 6 March 2019; DRAFT © 2004–2019 by William J. Rapaport

> I believe that history will record that around the mid twentieth century many classical problems of philosophy and psychology were transformed by a new notion of process: that of a *symbolic* or *computational* process.
> —Zenon Pylyshyn (1992, p. 4)

Figure 10.1: http://www.gocomics.com/agnes/2011/11/7, ©2011 Tony Cochran

10.1 Readings:

1. Required:

   (a) Cleland, Carol E. (1993), "Is the Church-Turing Thesis True?", *Minds and Machines* 3(3) (August): 283–312.

   (b) Preston, Beth (2013), "The Centralized Control Model", Ch. 1 of Preston, Beth (2013), *A Philosophy of Material Culture: Action, Function, and Mind* (New York: Routledge): 15–43.

      i. p. 17, first full paragraph

         - Introduces three theories of "production" (producing objects by following procedures): Aristotle's, Marx's, and Dipert's.

         A. §"Aristotle", especially p. 17 to p. 18 (end of top paragraph)

         B. §"Marx", especially p. 22 (from first full paragraph) to p. 23 (end of first full paragraph)

         C. §"Dipert", especially p. 29, first paragraph

      ii. §**"The Centralized Control Model", pp. 30–32**

      iii. §**"Control & Improvisation", pp. 39–43.**

      iv. Skim the rest of Ch. 1.

2. Very Strongly Recommended:

   - Farkas, David K. (1999), "The Logical and Rhetorical Construction of Procedural Discourse", *Technical Communication* 46(1) (February): 42–54, http://faculty.washington.edu/farkas/dfpubs/Farkas-ConstructionOfProceduralDiscourse.pdf

     – An interesting look, from the point of view of a technical writer, at how to write procedures (that is, instructions).

3. Strongly Recommended:

   (a) Horsten, Leon; & Roelants, Herman (1995), "The Church-Turing Thesis and Effective Mundane Procedures", *Minds and Machines* 5(1): 1–8, https://www.researchgate.net/publication/226286207_The_Church-Turing_thesis_and_effective_mundane_procedures

      - A reply to Cleland 1993.

   (b) Cleland, Carol E. (1995), "Effective Procedures and Computable Functions", *Minds and Machines* 5(1): 9–23.

      - A reply to Horsten & Roelants 1995.

   (c) Cleland, Carol E. (2001), "Recipes, Algorithms, and Programs", *Minds and Machines* 11(2) (May): 219–237.

   (d) Cleland, Carol E. (2002), "On Effective Procedures", *Minds and Machines* 12(2) (May): 159–179, https://pdfs.semanticscholar.org/a826/2a186f9e8828abd6b90a5604d87fab5ed713.pdf

4. Recommended:
   (a) Replies to Cleland:
      i. Israel, David (2002), "Reflections on Gödel's and Gandy's Reflections on Turing's Thesis", *Minds and Machines* 12(2) (May): 181–201.
      ii. Seligman, Jeremy (2002), "The Scope of Turing's Analysis of Effective Procedures", *Minds and Machines* 12(2) (May): 203–220.
   (b) Cleland, Carol E. (guest ed.) (2002), Special Issue on Effective Procedures, *Minds and Machines* 12(2) (May), http://link.springer.com/journal/11023/12/2

10.2 Introduction

Let us take stock of where we are. We proposed two possible answers to the question of what CS is. The first was that it is the scientific study of *computers*, and, in Chapters 6 and 9, we considered what a computer is, examining the history and philosophy of computers. The second possible answer was that perhaps CS is the scientific study of *computing*, that is, of algorithms, and, in Chapters 7 and 8, we investigated what algorithms are, their history, and their analysis in terms of Turing machines.

Algorithms—including procedures and recipes—can fail for many reasons; here are three of them:

1. They can omit crucial steps (as in Figure 10.1).
2. They can fail to be specific enough (or they can make too many assumptions) (recall Figure 7.10).
3. They can be highly context dependent or ambiguous (recall the first instruction in Figure 7.6).[1]

The general theme of the next few chapters is to challenge various parts of the informal definition of 'algorithm':

- Does it have to be a *finite* procedure?
- Does it have to be "*effective*"?
  - Does it have to halt?
  - Does it have to solve the problem? (What about heuristics?)
- Does it have to be *unambiguous or precisely described*? (What about recipes?)

In this chapter, we will look at one kind of objection to the Thesis, namely, that there is a more general notion—the notion of a "procedure". The objection takes the form of claiming that there are "procedures" that *are* computable in the *informal* sense but that are *not* computable by Turing machines. (For convenience, from now on we will use the expression 'Turing-computable' to mean "computable by anything logically equivalent to a Turing machine", that is, anything computable according to the classical theory of computability or recursive functions.)

In the next chapter, we will look at a related, but slightly different kind of objection, namely, that some functions that are *not* Turing–computable—such as the Halting Problem—*can* be computed in a formal, mathematical sense of "computation" that goes "beyond" Turing computation. This is called 'hypercomputation'.

[1] Or read the instructions at https://www.shopyourway.com/energizer-6v-led-utility-lantern/162752012. For more humorous versions of algorithms, see the cartoons archived at http://www.cse.buffalo.edu/~rapaport/510/alg-cartoons.html.

10.2.1 The Church-Turing Computability Thesis

> The [Church-Turing] thesis was a great step toward understanding algorithms, but it did not solve the problem [of] what an algorithm is.
> —Blass and Gurevich (2003, p. 2)

Recall from §7.5.4, that "Church's Thesis" is, roughly, the claim that the informal notion of "algorithm" or "effective computation" is equivalent to (or is completely captured by, or can be completely analyzed in terms of) Church's lambda calculus. More precisely:

> **Definition 2.1. Church's Thesis (First Version, unpublished, 1934).**
> A function is effectively calculable if and only if it is λ-definable.
> —Soare (2009, p. 372)

Later, Church reformulated it in terms of recursive functions:

> **Definition 2.2. Church's Thesis [1936].**
> A function on the positive integers is effectively calculable if and only if it is recursive. —Soare (2009, p. 372; see also §11.1, p. 389)

And "Turing's Thesis" is, roughly, the claim that the informal notion of "algorithm" or "computability" is equivalent to (or completely captured by, or can be completely analyzed in terms of) the notion of a Turing machine. We saw several versions of Turing's Thesis in Chapter 8.[2] Here is Robert I. Soare's version:

> **Definition 3.1. Turing's Thesis [1936].**
> A function is intuitively computable (effectively calculable) if and only if it is computable by a Turing machine .... —Soare (2009, p. 373)

Turing proved that Church's lambda calculus was logically equivalent to his own *a*-machines. That is, he proved that any function that was computable by the lambda calculus was also computable by a Turing machine (more precisely, that any lambda computation could be "compiled" into a Turing machine)[3] and vice versa—that any function that was computable by a Turing machine was also computable by the lambda calculus (so that the lambda calculus and Turing machines were inter-compilable). Consequently, their theses are often combined under the name the "Church-Turing Thesis".

There are other, less well-known computability theses. One is Emil Post's version:

> **Definition 5.1. [Post's Thesis, 1943, 1944].**
> A nonempty set is effectively enumerable (listable in the intuitive sense) iff it is recursively enumerable (the range of a recursive function) or equivalently iff it is generated by a (normal) production system. —Soare (2009, p. 380)

---

[2] §§8.5.1, 8.8.1, 8.8.2.8.2, 8.9.2, and 8.10.4.

[3] I am indebted to John Case's lectures (at SUNY Buffalo, ca. 1983) on the theory of computation for this phrasing.

This may look a bit different from Church's and Turing's versions, but, as Soare (2009, p. 380) notes,

> Since recursively enumerable sets are equidefinable with partial computable functions ... Post's Thesis is equivalent to Turing's Thesis.

Consequently, Soare (2009, §12) has argued that the thesis should be called simply the "Computability Thesis", on the grounds that—given the equivalence of all mathematical models of computation (Church's, Turing's, Gödel's, Post's, etc.)—there are really many such theses, hence no reason to single out one or two names, any more than we would refer to the calculus as 'the Newton calculus' or 'the Leibniz calculus'.

On the other hand, an interesting argument to the effect that Church's Thesis should be *distinguished* from Turing's Thesis has been given by Rescorla (2007): Church's Thesis asserts that intuitively computable *number-theoretic* functions are recursive. Turing's Thesis asserts that intuitively computable *string-theoretic* functions are Turing-computable. We can only combine these into a Church-Turing Computability Thesis by adding a requirement that there be a computable semantic interpretation function between strings and numbers. (And this raises the further question whether that semantic interpretation function is intuitively computable or recursive.) However, Sieg (2000) first analyzes the (informal) "calculability of number-theoretic functions" into calculability by humans "satisfying boundedness and locality conditions"; that, in turn, is analyzed into "computability by string machine"; finally, the latter is analyzed into computability by a Turing machine. Sieg identifies "Turing's thesis" as the analysis of the first of these by the last.

The Computability Thesis, in any of its various forms, states that the *informal* notion of effective computation (or algorithm, or whatever) is equivalent to the *formal* notion of a Turing-machine program (or a lambda-definable function, or a recursive function, or whatever). The arguments in favor of the Computability Thesis are generally of the form that (1) all known informal algorithms are Turing-computable[4] and (2) all of the formal, mathematical versions of computation are logically equivalent to each other (see §7.5.4).

It has also been argued that the Computability Thesis cannot be formally proved because one "side" of it is informal, hence not capable of being part of a formal proof. Dershowitz and Gurevich (2008) have suggested that the thesis *is* capable of being proved, by providing a set of formal "postulates" for the informal notion, and then proving that Turing machines satisfy those postulates. Although this is an interesting exercise, it is not obvious that this proves the Computability Thesis. Rather, it seems to replace that Thesis with a new one, namely, that the informal notion is indeed captured by the formal postulates. But *that* thesis likewise cannot be proved for the same reason that the Computability Thesis cannot: To prove it would require using an informal notion that cannot be part of a formal proof.

[4]This puts it positively. To put it negatively, no one has yet found a universally convincing example of an informally computable function that is not also Turing-computable

Further Reading:
Dershowitz and Gurevich 2008, §§1.1 and 1.2 contain a good history of the Computability Thesis. Other philosophers and logicians who have discussed how to prove the Computability Thesis include Gandy (1988) and Sieg (2000, 2008).

Kripke (2013) (which contains a lot of useful historical remarks) offers an argument that the Thesis can be proved as a corollary of Gödel's *Completeness* Theorem. On the other hand, Folina (1998) argues (against Gandy (1988), Mendelson (1990), Stewart Shapiro (1993), Sieg (1994), and others) that the thesis is true but unprovable (perhaps as in Gödel's *In*completeness Theorem?).

Stewart Shapiro 2013 is a very readable discussion of the provability of the Computability Thesis; of the nature of the "informality", "vagueness", or "open texture" of the notion of computability; and of the difference between human, mechanical, and mathematical computability, with observations on many of the other readings discussed or mentioned in this chapter.

Others have argued that neither (1) nor (2) are even *non*-deductively good arguments for the Computability Thesis. Against (1), it can be argued that, just because all *known* informal algorithms are Turing-computable, it does not follow that *all* informal algorithms are. After all, just because Aristotle's theory of physics lasted for some 2000 years until Newton came along, it did not follow that Aristotle's physics was correct, and just because Newton's theory lasted for some 200 years until Einstein came along, it did not follow that Newton's theory was correct. So, there is no inductive reason to think that the Computability Thesis is correct any more than there is to think that Einstein's theory is.[5]

But perhaps the Computability Thesis is *neither* a formally unprovable "thesis" *nor* a formally provable one, but something else altogether. In fact, Church called his statement of what we now name "Church's Thesis" "a *definition* of effective calculability". It is worth quoting in full, including parts of his important footnote 3:

> The purpose of the present paper is to propose a definition of effective calculability[3] which is thought to correspond satisfactorily to the somewhat vague intuitive notion in terms of which problems of this class are often stated, and to show, by means of an example, that not every problem of this class is solvable.
>
> [3] ... this definition of effective calculability can be stated in either of two equivalent forms, (1) that a function of positive integers shall be called effectively calculable if it is λ-definable ..., (2) that a function of positive integers shall be called effectively calculable if it is recursive .... And the proof of equivalence of the two notions is due chiefly to Kleene, ... the present author and to J.B. Rosser .... [Note: Kleene and Rosser were Church's Ph.D. students.] The proposal to identify these notions with the intuitive notion of effective calculability is first made in the present paper ....
>
> ... The fact ... that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of the reasons adduced below for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it. —Church (1936b, p. 346, my interpolated brackets)

[5] On the question of whether *any* scientific theory is correct, on the grounds that they are all only falsifiable, see Ch. 4, §4.9.1.2.

Definitiions, of course, are not susceptible of proof.

Rather than considering it a *definition*, the philosopher and logician Richard Montague (1960, p. 430) viewed the Thesis as an *explication* of the informal notion of "effective calculability" or "algorithm". An "explication" is the replacement of an informal or vague term with a formal and more precise one. (The concept is due to the philosopher Rudolf Carnap (1956, pp. 7–8). In a similar vein, the mathematician and logician Elliott Mendelson (1990, p. 229) calls the thesis a "*rational reconstruction*" (a term also due to Carnap): "a precise, scientific concept that is offered as an equivalent of a prescientific, intuitive, imprecise notion." Mendelson goes on to argue that the Computability Thesis has the same status as the definition of a *function* as a certain *set* of ordered pairs (see §7.3.1, above) or as other (formal) definitions of (informal) mathematical concepts (logical validity, Tarski's definition of truth, the δ-ε definition of limits, etc.). Mendelson then claims that "it is completely unwarranted to say that C[hurch's] T[hesis] is unprovable just because it states an equivalence between a vague, imprecise notion ... and a precise mathematical notion" (Mendelson, 1990, p. 232). One reason he gives is that *both* sides of the equivalence are *equally* vague! He points out that "the concept of set is no clearer than that of function". Another is that the argument that all Turing-machine programs are (informally) computable is considered to be a proof, yet it involves a vague, informal concept. (Note that it is the converse claim that all informally computable functions are Turing-computable that is usually considered incapable of proof on these grounds.)

It's worth comparing the formal explication of the informal (or "folk"?) notion of algorithm as a Turing machine (or recursive functions, etc.), with other attempts to define informal concepts in scientific terms. As with any attempt at a formal explication of an informal concept (as we discussed in Chapters 3 and 9), there is never any guarantee that the formal explanation will satisfactorily capture the informal notion (usually because the informal notion is informal, vague, or "fuzzy"). The formal explication might include some things that are, pre-theoretically at least, not obviously included in the informal concept, and it might exclude some things that are, pre-theoretically, included. Many of the attempts to show that there is something wrong with the Computability Thesis fall along these lines.

Question for the Reader:
As we noted in §3.3.3.2.3, 'life' is one of these terms. One difference between the two cases is this: There are many *non*-equivalent scientific definitions of 'life'. But in the case of 'algorithm', there are many *equivalent* formalizations: Turing machines, recursive functions, lambda calculations, etc. What might have been the status of the informal notion if these had *not* turned out to be equivalent?

Further Reading:
Mendelson's paper has one of the clearest discussions of the nature of the Computability Thesis. For responses to Mendelson, see Bringsjord 1993 and Stewart Shapiro 1993.

Good general discussions of the various Computability Theses (Church's, Turing's, et al.) and their history can be found in the *Notre Dame Journal of Formal Logic*'s Special Issue on Church's Thesis 28(4) (October 1987), http://projecteuclid.org/euclid.ndjfl/1093637642; Sieg 1994, §§2–3; Davis 2004; Soare 2009, §§2, 3, and 12 ("Origins of Computabilty and Incomputability", "Turing Breaks the Stalemate", and "Renaming It the 'Computability Thesis' "); and Olszewski et al. 2006 (which includes essays by Selmer Bringsjord, Carol Cleland, B. Jack Copeland, Janet Folina, Yuri Gurevich, Andrew Hodges, Charles McCarty, Elliott Mendelson, Oron Shagrir, Stewart Shapiro, and Wilfrid Sieg, among many others).

Robin Gandy, who was Turing's only Ph.D. student, gave this statement of "Church's" Thesis [sic]:

> What is effectively calculable ["by an *abstract human being* using some mechanical aids (such as paper and pencil)"] is computable ... [where] "computable" ... mean[s] "computable by a *Turing machine*"[,] ... "abstract" indicates that the argument makes no appeal to the existence of practical limits on time and space ... [and] "effective" in the thesis serves to emphasize that the process of calculation is deterministic—not dependent on guesswork—and that it must terminate after a finite time. —Gandy (1980, pp. 123–124, my italics)

Gandy goes on to be concerned with a *mechanical* version of the Thesis: whether "What can be calculated by a machine is computable" (Gandy, 1980, p. 124), where by 'machine' he says that he is ...

> ... using the term with its nineteenth century meaning; the reader may like to imagine some glorious contraption of gleaming brass and polished mahogany, or he [sic] may choose to inspect the parts of Babbage's "Analytical Engine" which are preserved in the Science Museum at South Kensington. —Gandy (1980, p. 125).

One difference between a human (even an "abstract" one) and a machine is that the latter can easily perform parallel operations such as printing "an arbitrary number of symbols simultaneously" (Gandy, 1980, p. 125).

Interlude: A Thought Experiment:
A "thought experiment" is a mental puzzle designed to give you some intuitions about the arguments to come.

Consider this quotation from Herbert Simon:

> Natural science impinges on an artifact through two of the three terms of the relation that characterizes it: the structure of the artifact itself and the environment in which it performs. [The third "term of the relation" is an artifact's "purpose or goal" (Simon, 1996b, p. 5).] Whether a clock will in fact tell time depends on its internal construction and where it is placed. Whether a knife will cut depends on the material of its blade and the hardness of the substance to which it is applied. ... An artifact can be thought of as ... an "interface" ... between an "inner" environment, the substance and organization of the artifact itself, and an "outer" environment, the surroundings in which it operates. If the inner environment is appropriate to the outer environment, or vice versa, the artifact will serve its intended purpose. —Simon (1996b, p. 6, my bracketed interpolation)

And, presumably, otherwise not.

But more can be said, perhaps relating to the "third term" mentioned in brackets above: Consider a clock. Lewis Carroll (of *Alice in Wonderland* fame) once observed that a(n analog) clock whose hands don't work at all is better than one that loses 1 minute a day, because the former is correct twice a day, whereas the latter is correct only once every 2 years (Carroll, 1850).

Here is the thought experiment: What about the clock that has stopped and one that is consistently 5 minutes slow? The latter is *never* correct—unless you simply move the numerals on its face 5 minutes counterclockwise; then it would always be correct (The actual number is probably closer to $5\frac{1}{2}$ minutes, but this is only a thought experiment!) The (user's?) *interpretation* of the clock's output seems to be what counts. (Recall, from §9.5, Searle's emphasis on user interpretations.)

10.3 What Is a Procedure?

Herbert Simon (1962, p. 479) offers two kinds of descriptions of phenomena in the world: *state* descriptions and *process* descriptions:

> The former characterize the world as sensed; they provide the criteria for identifying objects .... The latter characterize the world as acted upon; they provide the means for producing or generating objects having the desired characteristics.

The "desired characteristics" to be produced are, presumably, given by a state description. His example of a state description is "A circle is the locus of all points equidistant from a given point"; his example of a process description is "To construct a circle, rotate a compass with one arm fixed until the other arm has returned to its staring point". (Recall our discussion in §3.14.4 of Euclid's *Elements*, which was originally written in terms of "process descriptions".) Process descriptions describe procedures.

State descriptions seem to be part of "science", whereas process descriptions seem to be part of "engineering" and certainly part of "computational thinking". Consider this related claim of Michael Rescorla (2014b, §2, p. 1279):

> To formulate ... [Euclid's GCD algorithm], Knuth uses natural language augmented with some mathematical symbols. For most of human history, this was basically the only way to formulate mechanical instructions. The computer revolution marked the advent of rigorous *computational formalisms*, which allow one to state mechanical instructions in a precise, unambiguous, canonical way.

In other words, CS developed formal methods for making the notion and expression of procedures mathematically precise. That's what makes it a science of procedures.

Stuart C. Shapiro's (2001) more general notion of "procedure" (which we looked at in §3.9.3) characterizes " 'procedure' as the most general term for the way 'of going about the accomplishment of something' ", citing the Merriam-Webster *Third New International Dictionary*.[6] This includes serial algorithms as well as parallel algorithms (which are not "step by step", or serial), operating systems (which don't halt), heuristics (which "are not guaranteed to produce the correct answer"), musical scores (which are open to interpretation by individual performers), and recipes (which are also open to interpretation as well as being notoriously vague; see §10.4). Thus, Turing machines (or Turing-machine programs)—that is, (serial) algorithms as analyzed in §7.4—are only a special case of procedures. Given this definition, Shapiro claims that procedures are "natural phenomena that may be, and are, objectively measured, principally in terms of the amount of time they take ... and in terms of the amount of resources they require." He gives no argument for the claim that they are natural phenomena, but this seems reasonable. First, they don't seem to be "social" phenomena in the sense in which institutions such as money or governments are (Searle, 1995). Some of them might be, but the concept of a procedure in general surely seems to be logically prior to any specific procedure (such as the procedure for doing long division or for baking a cake). Second, there are surely some procedures "in nature", such as a bird's procedure for building a nest. Insofar as a natural science is one that deals with "objectively measurable phenomena" (again citing Webster; http://www.merriam-webster.com/dictionary/natural%20science), it follows trivially that CS is the natural science of procedures. (Recall the discussion of this point and the references cited in §3.9.3.)

In this chapter, we are focusing on this more general notion of 'procedure'.

Further Reading:
In conjunction with Shapiro's observation that CS education can help you write better cookbooks (see §3.9.3), Farkas (1999) offers some interesting (non-CS) advice on how to write procedures.

[6] http://www.merriam-webster.com/dictionary/procedure

10.4 Two Challenges to the Computability Thesis

Some philosophers have challenged the Computability Thesis, arguing that there are things that *are* intuitively algorithms but that are *not* Turing machines. In this section, we will look at two of these, due to Carol Cleland and to Beth Preston. Interestingly, both focus on recipes, though for slightly different reasons—Cleland on the fact that recipes are carried out in the real world, and Preston on the fact that they are vague and open to interpretation by chefs.

10.4.1 Carol Cleland: Some Effective Procedures Are Not Turing Machines

In a series of papers, Carol Cleland has argued that there are effective procedures that are not Turing machines (Cleland, 1993, 1995, 2001, 2002, 2004). By 'effective procedure', she means (1) a "mundane" procedure—that is, an ordinary, everyday, or "quotidian" one—that (2) generates a causal process, that is, a procedure that physically causes (or "effects") a change in the world.

Terminological Digression and Further Reading:
There may be an unintentional pun here. As we have seen, the word 'effective' as used in the phrase 'effective procedure' is a semi-technical term that is roughly synonymous with 'algorithmic'. On the other hand, the *verb* 'to effect', as used in the phrase "to effect a change (in something)", is roughly synonymous with the verbs 'to produce' or 'to cause', and it is not directly related to 'effective' in the algorithmic sense.

And just to make things more confusing, 'effect' is also a *noun* meaning "the result of a cause". Worse, there are a verb and a noun spelled slightly differently but pronounced almost the same: 'affect'! For the difference between the *verb* 'to effect' and the *noun* 'an effect', as well as the similar-sounding verb and noun 'affect', see Rapaport 2013, §4.2.0, "affect vs. effect".

Algorithms implement mathematical functions; they transform ("change"?) inputs into outputs. But do *functions* change anything? This question is discussed at http://www.askphilosophers.org/question/1877.

According to Cleland, there are three ways to understand the Computability Thesis:

1. It applies *only* to (mathematical) functions of integers (or, possibly, also to anything representable by—that is, codable into—integers).
2. It applies to *all* (mathematical) functions (including real-valued functions).
3. It also applies to the production of mental and physical phenomena, such as is envisaged in AI or robotics.

She agrees that it cannot be proved but that it *can* be falsified. (Recall, from Ch. 4, §4.9.1, Popper's thesis that falsifiability is the mark of a science.) It can't be *proved*, because one of the two notions that the Computability Thesis says are equivalent is an informal notion, hence not capable of occurring in a formal proof. There are two

possibilities for why it can be *falsified*: According to Cleland, the Computability Thesis can be falsified by exhibiting an intuitively effective procedure, "*but not in Turing's sense*", that is "more powerful" than a Turing machine (Cleland, 1993, p. 285, my italics). Presumably, the qualification "but not in Turing's sense" simply means that it must be intuitively effective yet not capable of being carried out by a Turing machine, because, after all, that's what Turing thought his *a*-machines could do, namely, carry out any intuitively effective procedure.

But she also suggests another sense in which the Computability Thesis might be falsifiable: By exhibiting a procedure that *is* intuitively effective *in Turing's sense* yet is not Turing–computable. In other words, there might be two different kinds of counterexamples to the Computability Thesis: If Turing were alive, (1) we could show him an intuitively effective procedure that we claim is not Turing-computable, and he might agree; or (2) we could show him a procedure, and, *either* he would *not* agree that it was intuitively effective (thus denying that it was a possible counterexample), *or* he could show that, indeed, it *was* Turing-computable (showing how it is not a counterexample at all). Cleland seems to be opting for (1).

Curiously, however, she goes on to say that her "mundane procedures" *are* going to be effective "in Turing's sense" (Cleland, 1993, p. 286)! In any case, they differ from Turing-computable procedures by being *causal*. (When reading Cleland's article, you should continually ask yourself two questions: *Are* her "mundane procedures" causal? Are Turing machines *not* causal?) Here is her argument, with comments after some of the premises:

1. A "procedure" is a specification of something *to be followed* (Cleland, 1993, p. 287). This includes recipes as well as computer programs.

    - Her characterization of a procedure as something to be followed puts a focus on imperatives: You can follow an instruction that says, "Do this!". But there are other ways to characterize procedures. Stuart C. Shapiro (2001), for example, describes a procedure as a *way* to do something. But his focus is on the goal or end product; the way to do it—the way to accomplish that goal—might be to evaluate a function or to determine the truth value of a proposition, not necessarily to "follow" an imperative command.
    - You should also recall (from §8.11.1) that Turing machines don't normally "follow" any instructions! The Turing-machine table is a description of the Turing machine, but it is not something that the Turing machine consults and then executes. That only happens in a *universal* Turing machine, but, in that case, there are two different programs to consider: There is the program encoded on the universal Turing machine's tape; that program *is* consulted and followed. But there is the fetch-execute procedure that constitutes the universal Turing machine's machine table; that one is *not* consulted or followed.
    - The relationship between these two programs gives rise to several interesting issues in epistemology and the philosophy of AI: Is there any sense in which the "blindly executed" fetch-execute cycle "knows" what it is doing when it "follows" the program on the tape? (Recall the passage cited

in §9.5.3 from Nicolas D. Goodman.) Does an AI program "understand" what it does? (Should it? Could it?) We'll discuss these issues in Chapter 19 when we discuss what Daniel C. Dennett (2013b) has called "Turing's 'Strange Inversion of Reasoning' ".

2. To say that a "mundane" procedure is "effective" means, by definition, that following it always results in a certain kind of outcome (Cleland, 1993, p. 291).

    - The semi-technical notion of "effective" as it is used in the theory of computation is, as we have seen (§7.4), somewhat ambiguous. Cleland notes (1993, p. 291) that Marvin Minsky (1967) calls an algorithm 'effective' if it is "precisely described". And Church (1936b, pp. 50ff; compare p. 357) calls an algorithm 'effective' if there is a formal system that takes a formal analogue of the algorithm's input, precisely manipulates it, and yields a formal analogue of its output. Church's notion seems to combine aspects of both Minsky's and Cleland's notions.

    - A non-terminating program (either one that erroneously goes into an infinite loop or else one that computes all the digits in the decimal expansion of a real number) can be "effective" *at each step* even though it never halts. We'll return to this notion in Chapter 11, when we look at the notion of interactive computing.

3. The steps of a recipe *can* be precisely described (hence, they can be effective in Minsky's sense).

    - This is certainly a controversial claim. Note that recipes can be notoriously vague, whereas computer programs must be excruciatingly precise:

        > How do you know when a thing "just begins to boil"? How can you be sure that the milk has scorched but not burned? Or touch something too hot to touch, or tell firm peaks from stiff peaks? How do you define "chopped"? —Adam Gopnik (2009b, p. 106); compare Sheraton 1981.

      We will explore this in more detail in §10.4.2.

4. A procedure is *effective for* producing a specific output. For example, a procedure for producing fire or a procedure for producing hollandaise sauce might not be effective for producing chocolate.

    - In other words, being effective (better: being "effective *for*") is not a *property of* a procedure but a *relation between* a procedure *and* a kind of output. This might seem to be reasonable, but a procedure for producing the truth table for conjunction might also be effective for producing the truth table for disjunction by suitably reinterpreting the symbols. (See the "Digression on Conjunction and Disjunction", below.)

Digression on Conjunction and Disjunction:
Here is a truth-table for conjunction, using '0' to represent "false" and '1' to represent "true":

0	0	0
0	1	0
1	0	0
1	1	1

Note that, in the 3rd column—which represents the conjunction of the first two columns—there are three '0's and one '1', which occurs only in the line where both inputs are '1'.

And here is the analogous truth-table for disjunction:

0	0	0
0	1	1
1	0	1
1	1	1

Note that the third column has three '1's and only one '0', which occurs only in the line where both inputs are '0'.

Now suppose, instead, that we use '0' to represent "*true*" and '1' to represent "*false*". Then the *first* table represents *disjunction*, and the *second* one represents *conjunction*!

(Similar points are made by Peacocke (1995, §1, p. 231), Shagrir (2001), and Sprevak (2010, §3.3, pp. 268–269).) We'll return to this example in Chapter 17.

- Here are some more examples: A procedure that is effective for simulating a battle in a war might also be effective for simulating a particular game of chess (Fodor, 1978, p. 232). Or a procedure that is effective for computing with a mathematical lattice might also be effective for computing with a chemical lattice. (For the details on both of these, see §17.4.2.1.) And an ottoman (or a "pouf") could be (used as) either a chair[7] or a table, yet, of course, chairs and tables are usually considered to be mutually exclusive classes.
- In cases such as these, the notion of effectiveness might not be the same as Church's, because of the possibility of interpreting the output differently. How important to the notions of (intuitively) effective computation and formal computation is the *interpretation* of the output symbols? We will explore these issues in more detail in Chapter 17 ("How Do Programs Relate to the World?").

[7] Or at least a seat.

5. The effectiveness of a recipe for producing hollandaise sauce depends on causal processes in the actual world, and these causal processes are independent of the recipe (the mundane procedure) (Cleland, 1993, p. 294). Suppose that we have an algorithm (a recipe) that takes eggs, butter, and lemon juice as input, tells us to mix them, and that outputs hollandaise sauce. Suppose that on Earth the result of mixing those ingredients is an emulsion that is, in fact, hollandaise sauce. And suppose that, on the Moon, mixing them does not result in an emulsion, so that no hollandaise sauce is output (instead, the output is a messy mixture of eggs, butter, and lemon juice).

    - Now it is time to recall our thought experiment: The "effectiveness" of a clock's telling the correct time depends just as much on the orientation of the clock face as it does on the position of the hands and the internal clockwork mechanism. That is, it depends, in part, on a situation in the external world. And the orientation of the clock face is independent of the clock's internal mechanism (or "procedure").

6. Therefore, mundane processes can (must?) be effective for a given output P in the *actual* world yet not be effective for P in some other *possible* world.

    - This is also plausible. Consider a blocks-world computer program that instructs a robot how to pick up blocks and move them onto or off of other blocks (Winston, 1977). I once saw a live demo of such a program. Unfortunately, the robot failed to pick up one of the blocks, because it was not correctly placed, yet the program continued to execute "perfectly" even though the output was not what was intended (Rapaport 1995, §2.5.1, p. 62; a similar situation is discussed in Dennett 1987, Ch. 5, "Beyond Belief", p. 172).

7. Turing machines are equally effective in all possible worlds, because they are causally inert.

    - But here we have a potential equivocation on 'effective'. Turing machines are effective in the sense of being step-by-step algorithms that are precisely specified, but they are not necessarily effective *for an intended output P*: It depends on the interpretation of P in the possible world!

8. Therefore, there are mundane procedures (such as recipes for hollandaise sauce) that can produce hollandaise sauce because they result in appropriate causal processes, *but* there are no Turing machines that can produce hollandaise sauce, because Turing machines are purely formal and therefore causally inert. QED

To the objection that physical implementations of Turing machines could be causally "ert" (so to speak),[8] Cleland replies as follows: The embodied Turing machine's "actions" are not physical actions but action-*kinds*; therefore, they *are* causally *in*ert. But

[8] 'Inert' comes from the Latin prefix 'in-', meaning "not", and the Latin 'artem', meaning "skill"; so, if 'inert' means "lacking causal power", then perhaps the non-word 'ert' could mean "having causal power"; see the *OED*'s entry on 'inert' [http://www.oed.com/view/Entry/94999].

an embodied Turing machine *does* act: *Embodied* action-*kinds are* causal actions. Alternatively, a Turing machine's '0'-'1' outputs can be interpreted by a device that does have causal effects, such as a light switch or a thermostat.

Perhaps a procedure or algorithm that is "effective for P" is better understood as an algorithm *simpliciter*. In the actual world, it does P. In some other possible world, perhaps it does Q ($\neq P$). In yet another possible world, perhaps it does nothing (or loops forever). And so on. For example,

> ... if we represent the natural number n by a string of n consecutive 1s, and start the program with the read-write head scanning the leftmost 1 of the string, then the program,
>
> q_0 1 1 R q_0
> q_0 0 1 R q_1
>
> will scroll the head to the right across the input string, then add a single '1' to the end. It can, therefore, be taken to compute the successor function.
> —Aizawa (2010, p. 229)

But if the environment (the tape) is not a string of n '1's followed by a '0', then this does *not* compute the successor function. Compare this to Cleland's hollandaise sauce recipe being executed on the Moon. Hence, mundane procedures are *interpreted* Turing machine programs, so they *are* computable.

Aaron Sloman (2002, §3.2) makes a possibly useful distinction between "internal" and "external" processes: The former "include manipulation of cogs, levers, pulleys, strings, etc." The latter "include movements or rearrangements of various kinds of physical objects". So, a computer on Earth that is programmed to make hollandaise sauce and one on the Moon that is identically programmed will have the same internal processes, but different external ones (because of differences in the external environment).

A related distinction was made by linguist Noam Chomsky between *competence*: "an ideal" language user's "knowledge of his [sic] language"—and *performance*: "the actual use of language in concrete situations" (Chomsky, 1965, pp. 3–4). A computer might be *competent* to make hollandaise because of its *internal* processes, yet fail to do so because of *performance* limitations due to *external*-environmental limitations.

Does the ability of a machine to *do* something that is not Turing-computable mean that it can *compute* something that is not Turing-computable? What does physical performance have to do with computation? Surely we want to say that whatever a robot can do is computable, even if that includes cooking. But surely that's because of the "internal" processes, not the "external" ones.

> ... Turing machines are not so relevant [to AI] intrinsically as machines that are designed from the start to have interfaces to external sensors and motors with which they can interact online, unlike Turing machines which at least in their main form are totally self contained, and are designed primarily to run in ballistic mode once set up with an intial machine table and tape configuration.
> —Sloman (2002, §4.2, my bracketed interpolation)

This seems to be a distinction between abstract Turing machines and robots. And Cleland's arguments seem more relevant to robots than to Turing machines, hence have nothing really to say about the Computability Thesis (which only concerns Turing machines and their equivalents). Indeed, Copeland and Sylvan (1999, p. 46) (see also Copeland 1997) distinguish between two interpretations of the CTCT. The one that they claim was actually stated by Church and by Turing "concerned the functions that are in principle computable by an *idealised human being unaided by machinery*". This one, they claim, is correct. The other interpretation is "that the class of well-defined computations is exhausted by the computations that can be carried out by Turing machines".

So, one possible objection to Cleland is that cooking (for example) is not something that can be carried out by "an idealized human being unaided by machinery", hence the failure of a hollandaise sauce recipe on the moon is irrelevant to the correct interpretation of the CTCT.

Compare Cleland's hollandaise sauce example with the following: Suppose that we have an algorithm (a recipe) that tells us to mix eggs, butter, and lemon juice *until an emulsion is produced*, and that outputs hollandaise sauce. In the actual world, an emulsion is indeed produced, and hollandaise sauce is output. But on the moon, this algorithm goes into an infinite loop; nothing (and, in particular, no hollandaise sauce) is ever output.

One problem with this is that the "until" clause ("until an emulsion is produced") is not clearly algorithmic. How would the *computer* tell if an emulsion has been produced? This is not a clearly algorithmic, Boolean condition whose truth value can be determined by the computer simply by checking one of its switch settings (that is, a value stored in some variable). It would need sensors to determine what the external world is like. But that is a form of *interactive* computing, which we'll discuss in Chapter 11.

Further Reading:
Wells 2004, §2 contains a discussion of Cleland (1995, 2001) in addition to remarks on the relation between the Computability Thesis and the $P = NP$ problem.

10.4.2 Beth Preston: Recipes, Algorithms, and Specifications

Introductory computer science courses often use the analogy of recipes to explain what algorithms are. Recipes are clearly procedures of some kind (Sheraton, 1981), (Shapiro, 2001). But are recipes really good models of *algorithms*?

Cleland has assumed that they are. Beth Preston (2013) has a different take. She is interested in the nature of artifacts and how they are produced (or implemented) from plans (such as blueprints). Compare this to how a program (which is like a plan) is actually executed.

According to Preston, the classical view of production is that of "centralized control". The etymology of the word 'control' is of interest here: 'To control' originally meant "to check or verify (originally by comparison with a duplicate register) ..." (*OED*, http://www.oed.com/view/Entry/40563). The "duplicate register" was a

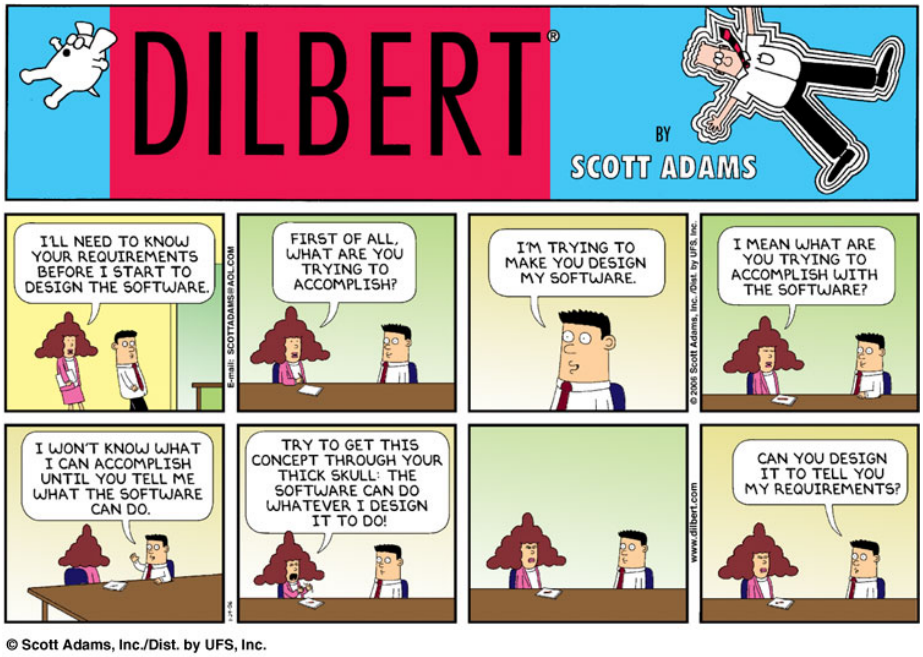


Figure 10.2: http://dilbert.com/strip/2006-01-29

"counter-roll"; to control something originally meant to faithfully copy it for the sake of verification or to regulate it. So, to implement a plan is to copy an abstract design into reality, that is, to control it. (For more on this idea of verification by comparison, see the discussion of the relation between syntax and semantics in Chapter 22.)

A "mental design" of an artifact to be produced first exists in someone's mind. This mental design "specifies all the relevant features of the" artifact to be produced (the "copy") (Preston, 2013, p. 30), "along with a set of instructions for construction" (Preston, 2013, p. 39). Then the "actual construction" of the artifact (that is, the copying of the mental design) ...

> ... is a process that faithfully follows the instructions of the construction plan, and by so doing reproduces in a material medium the features of the product specified in the design. This faithful copying relationship between the design and construction phases of production is the *control* aspect of the model.
> —Preston (2013, pp. 30–31)

Compare the way in which a program controls the operations and output of a computer.

But there is a problem: A "faithful copy" requires that ...

> ... *all* relevant features of the product [the artifact] be specified in the design .... In other words, the design is ideally supposed to be an algorithm (effective procedure) for realizing both the construction process and the product.
> —Preston (2013, p. 39, my italics and bracketed interpolation)

According to Preston, however, recipes show that this ideal model doesn't describe reality.

Recipes differ from algorithms in several ways (Preston, 2013, p. 40):

1. Recipes leave details open (for example, details about ingredients, which play the same role in recipes that data structures do in programs):

   (a) Recipes provide for *alternatives*: use "either sour cream or yogurt".

   - But couldn't a recipe, or a program for that matter, simply call for a data-analogue of a typed variable or a subroutine here? The recipe might call, not for sour cream or else yogurt, but, instead, for a "white, sour, milk-based substance".
   - And what about non-deterministic algorithms? Here is an example of a non-deterministic procedure for computing the absolute value of x using a "guarded if" control structure:[9]

     if $x \geq 0$ **then** return x;
     if $x \leq 0$ **then** return $-x$;

     Here, if $x = 0$, it does not matter which line of the program is executed. In such procedures, a detail is left open, yet we still have an *algorithm*.
   - Or consider a program that simply says to input an integer, without specifying anything else about the integer. (As a magician might say, "Think of a number, any number ..."). This could still be an algorithm. (Or maybe not! See Chapter 11.)

   (b) Recipes specify some ingredients *generically*: "use frosting", without specifying what kind of frosting.

   - Here, again, compare typed variables or named subroutines. It does not matter *how* the subroutine works; all that matters is its input-output behavior (*what* it does). And it doesn't matter *what* value the variable takes, as long as it is of the correct type. Typed variables and named subroutines are "generic", yet they appear in algorithms.

   (c) Recipes provide for *optional* ingredients: "use chopped nuts if desired".

   - But compare any conditional statement in a program that is based on user input. (On the other hand—as noted in premise (1a), above—user input *may* raise issues for interactive computing; see Chapter 11.)

2. Recipes leave construction steps (= control structure?) open: For instance, the order of steps is not necessarily specified (at best, a *partial order* is given): "add the rest" of the ingredients (where no order for adding is given for "the rest of the ingredients").

   - Again, compare non-deterministic statements, such as the guarded-if command in the above example, or programs written in languages like Lisp,

[9] In an ordinary "if" statement, when more than one Boolean condition is satisfied, the first one is executed. In a "guarded if" statement, it doesn't matter which one is executed. See Gries 1981, Ch. 10, or http://en.wikipedia.org/wiki/Guarded_Command_Language#Selection:_if

where the order of the functions in the program is not related in any way to the order in which they are evaluated when the program is executed. (A Lisp program is an (unordered) *set* of functions, not a(n ordered) *sequence* of instructions.)

3. In recipes, some necessary steps ("put these cookies on a baking sheet before baking them") can be *omitted* (that is, go unmentioned). But should the baking sheet be greased? A knowledgeable chef would know whether it has to be, so a recipe written for such a chef need not mention the obvious.

    - But the same kind of thing can occur in a program, with preconditions that are assumed but not spelled out, or details hidden in a subroutine. (Perhaps "put cookies on baking sheet" is a call to a subroutine of the form: "grease baking sheet; put cookies on it".)

4. Recipes can provide *alternative ways* to do something: "Roll in powdered sugar ... or shake in bag with ... powdered sugar".

    - Again, non-determinism is similar, as are subroutines: To say "multiply x and y" is not to specify how; to say "coat in powdered sugar" is not to specify whether this should be done by rolling or shaking.

Preston claims that the cook (that is, the CPU) is expected to do some of the design work, to supply the missing parts. So, not everything is in the design. She claims that cooks don't *faithfully follow* recipes; instead, they *improvise*, as jazz or rock musicians do. They can even change other ("fixed") parts of the recipe, because of their background knowledge, based on experience. For example, they can substitute walnuts for pecans in a recipe for pecan pie. Therefore, the constructor or executor is typically *intelligent*, in contrast to an *un*intelligent CPU (or the "unintelligent" fetch-execute cyle of a universal Turing machine; see my second comment on Cleland's premise 1, above).

But here is a different interpretation of Preston's analysis: She offers the centralized control model as a description of an *algorithm* together with a CPU that produces a *process* (that is, an algorithm being executed). But her theory of collaborative improvisation might better describe an earlier stage in the production of a process, namely, the production of an *algorithm* by a programmer from a *specification*. That is, although the execution of an algorithm might well be modeled as centralized control, nevertheless the development of an algorithm by a programmer from a specification might well be improvisatory and collaborative, precisely because specifications—like recipes—can be vague and open to interpretation.

So, recipes are more like *design specifications* for computer programs than they are like *algorithms*. In fact, my counterexamples to differentiate between algorithms and recipes just show that *either* recipes are high-level programs requiring implementation in a lower level, *or* that recipes are specifications.

Further Reading:
On the notion of a "specification" (the general requirements for an algorithm), see Turner 2011. Petroski 2010 is a nice follow-up to Preston, in which he describes how "an everyday challenge provides lessons in the processes of planning and execution".

Daly 2010—about robots that not only follow recipes, but actually cook the meals ("Fear not, humans, these robots are here to be our friends. To prove it, they serve you food")—is interesting reading in connection with what both Cleland and Preston have to say about the computability of recipes.

10.5 Discussion

So, are there good reasons for seriously doubting the Computability Thesis? We have just seen two candidates: Cleland argues that certain "mundane procedures" are effectively computable but not Turing-computable, and Preston suggests that certain recipe-like procedures of the sort typically cited as examples of effective procedures are not really algorithmic.

But against Cleland's example, we have seen that there may be a concern in how one determines what the proper output of an algorithm is, or, to put it another way, in determining the problem that an algorithm is supposed to solve. Consider the recipe for hollandaise sauce that, when correctly executed on Earth, produces hollandaise sauce but, when correctly executed on the Moon, does not. Is that recipe therefore not Turing-computable? It would seem to be Turing-computable on Earth, but not on the Moon. Or is it Turing-computable *simpliciter* (for example, no matter where it is executed), but conditions having nothing to do with the algorithm or recipe *itself* conspire to make it unsuccessful *as a recipe for hollandaise sauce* on the Moon? Is the algorithm or recipe *itself* any different?

And against Preston's example, we have seen that recipes are, perhaps, more like *specifications* for algorithms than they are like *algorithms*.

Against the view that all mathematical models of computation are logically equivalent, it can be argued that, even though the common theme underlying the equivalence of Turing machines, lambda definability, recursive functions, etc., is "robust" and of great mathematical interest, that is not reason enough to think that there might not be any *other* theory of effective computation. In the next chapter, we will look at such potential counterexamples to the Computability Thesis.

I will close this chapter with one last version of the Thesis (not to be taken too seriously!):

> The Church-Turing Thesis:
> A problem is computable just in case it wants to be solved.
> —anonymous undergraduate student in the author's course, CSE 111, "Great Ideas in Computer Science" (2000)[10]

[10] http://www.cse.buffalo.edu/~rapaport/111F04.html

Further Reading:

1. On the Computability Thesis and AI:

   (a) Kearns (1997) argues that the Computability Thesis "has no interesting consequences for human cognition" because "carrying out a procedure is a purposive, intentional activity. No actual machine does, or can do, as much."

   (b) On the other hand, Abramson (2011) argues that the Computability Thesis *is* relevant to questions about the Turing test for AI.

   (c) Rey (2012) distinguishes the Turing Thesis from the Turing Test.

2. Bowie (1973, esp. p. 67) argues that Church's Thesis is false on the grounds that the informal notion of computability is intensional but the notion of recursive functions is extensional. (Recall our discussion in §3.4 of the difference between "intensional" and "extensional".) He also argues against it on what would now be considered "hypercomputational" grounds, roughly, that, if you give a computable function a non-computable input (intensionally represented), it will compute a non-computable output (Bowie, 1973, p. 75). We'll look at "hypercomputation" in the next chapter.

   Ross (1974) points out that Bowie's definition of 'computable' is non-standard, and argues that the standard definition (essentially the one we give in Chapter 7) is *not* intensional.

3. Robin Gandy (1980, p. 124) notes that

   > Gödel has objected, against Turing's arguments, that the human mind may, by its grasp of abstract objects [by "insight and imaginative grasp"], be able to transcend mechanism.

   As Gandy notes, we need to know much more about the nature and limits of the human mind than we do in order to adequately evaluate this objection. After all, what *seems* to be "non-mechanical intelligence" need not be. (Compare the remarks above about universal Turing machiines, and see our discussion in Chapter 19 about Turing's "inversion".)

   A more detailed reply to Gödel is given by Kleene (1987, pp. 492–494), the bottom line of which is this:

   > The notion of an "effective calculation procedure" or "algorithm" (for which I believe Church's thesis) involves its being possible to convey a complete description of the effective procedure or algorithm by a finite communication, in advance of performing computations in accordance with it. (p. 493.)

   And, according to Kleene, Gödel's objection fails to satisfy that criterion (as does an objection raised by Kalmár (1959)).

Chapter 11

What Is Hypercomputation?

Version of 11 July 2019; DRAFT © 2004–2019 by William J. Rapaport

> Church, Gödel, and Turing defined ... [computation] in terms of mathematical functions ... Today, I believe we are breaking out of the era where only algorithmic processes are included in the term *computation*. —Dennis J. Frailey (2010, p. 2)

> Speculation that there may be physical processes—and so, potentially, machine-operations—whose behaviour cannot be simulated by the universal Turing machine of 1936 stretches back over a number of decades. Could a machine, or natural system, deserving the name 'hypercomputer' really exist? Indeed, is the mind—or the brain—some form of hypercomputer?
> —B. Jack Copeland (2002b, p. 462)

> We now know both that hypercomputation (or super-recursive computation) is mathematically well-understood, and that it provides a theory that according to some accounts for some real-life computation ... [is] better than the standard theory of computation at and below the "Turing Limit." ... [S]ince it's mathematically *possible* that human minds are hypercomputers, such minds *are* in fact hypercomputers. —Bringsjord and Arkoudas (2004, p. 167)

> The editors have kindly invited me to write an introduction to this special issue devoted to "hypercomputation" despite their perfect awareness of my belief that there is no such subject. —Martin Davis (2006c, p. 4)

> Nobody would be fired from a computer science department for axiomatizing analog computation or hypercomputation. Both are still in [the] purview of computer science. —Marcin Miłkowski (2018, §3.2)

11.1 Readings:

1. Required:

   (a) Copeland, B. Jack (2002), "Hypercomputation", *Minds and Machines* 12(4) (November): 461–502.

   - A good overview and survey.
   - Read pp. 461–465.
   - Read §1.5, on Putnam & Gold
   - Read §1.7, on Boolos & Jeffrey
   - Read §1.13, on Kugel
   - Read §1.18–§1.18.1, on Penrose, Turing, & Gödel
   - Read §1.24, on Cleland
   - Read all of §§2–3

   (b) Wegner, Peter (1997), "Why Interaction Is More Powerful than Algorithms", *Communications of the ACM* 40(5) (May): 80–91.

   - Many of Wegner's papers are online in various formats at his homepage: http://www.cs.brown.edu/people/pw/home.html

   (c) Kugel, Peter (2002), "Computing Machines Can't Be Intelligent (... and Turing Said So)", *Minds and Machines* 12(4) (November): 563–579, http://cs.bc.edu/~kugel/Publications/Hyper.pdf

   (d) Soare, Robert I. (2009), "Turing Oracle Machines, Online Computing, and Three Displacements in Computability Theory", *Annals of Pure and Applied Logic* 160: 368–399.

   - Preprint at http://www.people.cs.uchicago.edu/~soare/History/turing.pdf
   - Published version at http://www.sciencedirect.com/science/article/pii/S0168007209000128
   - You can *skim* some of the more technical parts (such as §§4.2.1–4.3, 4.4.1, 7–8).
   - A slightly different version appears as Soare 2013a.

   (e) Davis, Martin (2004), "The Myth of Hypercomputation", in C. Teuscher (ed.), *Alan Turing: The Life and Legacy of a Great Thinker* (Berlin: Springer): 195–212, http://www1.maths.leeds.ac.uk/~pmt6sbc/docs/davis.myth.pdf

2. Very Strongly Recommended:

   (a) Eberbach, Eugene; & Wegner, Peter (2003), "Beyond Turing Machines", *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, No. 81 (October): 279–304, http://www.eatcs.org/images/bulletin/beatcs81.pdf

   - Another useful survey, but it opens with some seriously misleading comments about the history of computation and the nature of logic.[1]

   (b) Davis, Martin (2006), "Why There Is No Such Discipline as Hypercomputation", *Applied Mathematics and Computation* 178: 4–7, http://tinyurl.com/Davis2006[2]

   (c) On "trial-and-error" machines:

   - Johnson, Kent (2004), "Gold's Theorem and Cognitive Science", *Philosophy of Science* 71 (October): 571–592, http://www.lps.uci.edu/~johnsonk/Publications/Johnson.GoldsTheorem.pdf
   - John Case's COLT [Computational Learning Theory] Page, http://www.eecis.udel.edu/~case/colt.html
   - Nowak, Martin A.; Komarova, Natalia L.; & Niyogi, Partha (2002), "Computational and Evolutionary Aspects of Language", *Nature* 417 (6 June): 611–617, http://people.cs.uchicago.edu/~niyogi/papersps/NKNnature.pdf

[1] For example, contrary to what the authors say, (1) Turing 1936 was *not* "primarily a mathematical demonstration that computers could not model mathematical problem solving" (p. 279) (although it did include a mathematical demonstration that there were *some* mathematical problems that a computer could not *solve*); and (2) Hilbert did *not* take the *Entscheidungsproblem* "as a basis that all mathematical problems could be proved true or false by logic" (p. 279), simply because mathematical *problems* cannot be "proved true or false"—rather, mathematical *propositions* might be *provable* or not, *and* they might be true or false, but truth and falsity are *semantic* notions incapable of proof, whereas provability is a *syntactic* notion.

[2] http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_APPLIED_MATHEMATICS_AND_COMPUTATION/Special_Issue_on_Hypercomputation/davis%5b1%5d.pdf

11.2 Introduction

We have seen that it can be argued that:

1. CS includes (among other things) the systematic study of *computing*,
2. computing is the mathematical study of *algorithms*,

   and

3. algorithms are best understood mathematically in terms of *Turing machines* (or anything logically equivalent to them).

But *are* algorithms best understood that way?

Let's first consider different kinds of "computation". We have already distinguished between analog and discrete (or digital) computation (see §§6.5.2 and 9.3). Within the latter, we can distinguish several "levels" of computation. There are several models of computation that are *weaker* than Turing-machine computation; let's call them "sub-Turing computation": In §7.6, we looked at primitive recursion and count-programs. But there are models that are even weaker than those: Informally, a *finite automaton* is a machine that only

> moves from left to right on a finite input tape .... [It] will have only one opportunity to scan each square in its motion from left to right, [and] nothing will be gained by permitting the device to "print" new symbols on its tape .... Thus, a finite automaton can be thought of as a very limited computing device which, after reading a string of symbols on the input tape, either accepts the input or rejects it, depending upon the state the machine is in when it has finished reading the tape.
> —Davis and Weyuker (1983, pp. 149–150, my interpolated brackets)

(For more on sub-Turing computation, see Bernhardt 2016, Ch. 3.)

Turing-machine computation (or any model of computation that is logically equivalent, such as the lambda calculus, recursive functions, register machines, and so on) is at the "top" of these levels. A reasonable question to ask is whether there are levels "above" it: Is there such a thing as "super"-Turing computation? If so, how would it affect the Computability Thesis? After all, that thesis says that any (informal) notion of "computation" is equivalent to Turing computation. Sub-Turing computation can be performed by Turing machines, simply by not using all of the "power" of Turing machines. But if super-Turing computation can do "more" than classical Turing computation—perhaps even just using Turing machines—wouldn't that be a counterexample to the Computability Thesis?

Further Reading:
Sub-Turing systems are sometimes referred to as "*hypo*computation" (from the Greek root 'hypo', meaning "under"; 'hyper' means "over"). For more on such models, see any text on the theory of computation (such as those cited in §7.5.4), as well as discussions of the "Chomsky hierarchy" (for example, https://en.wikipedia.org/wiki/Chomsky_hierarchy). Also see Lindell 2004, 2006.

Recall our discussion of Kuhn's philosophy of science in §4.9.2: To the extent that the Church-Turing Computability Thesis is the standard "paradigm" in CS, rejection of it could be considered as a Kuhnian revolutionary challenge to "normal" CS (Stepney et al. 2005; Cockshott and Michaelson 2007, §2.5, p. 235). We saw in the previous chapter that it can be argued that there might be "procedures" that are not computable by a Turing machine. But, of course, this depends on what is meant by 'procedure'. Recall from §7.4.3.3 that Hopcroft and Ullman (1969, p. 2) and Knuth (1973, p. 4) distinguish between "algorithms", which must halt, and "procedures", which need not. Hopcroft and Ullman, (p. 80, my italics) also characterize "Church's hypothesis" as the claim "that any process which could naturally be called a *procedure* can be realized by a Turing machine". Since procedures in their sense need not halt, neither need Turing machines.

Others have suggested that by relaxing one or more of the constraints on the notion of "algorithm", we can produce counterexamples to the Computability Thesis. In this chapter, we continue our look at arguments for this point of view and at some of the other kinds of procedures that are allegedly not computable by a Turing machine.

11.3 Hypercomputation

11.3.1 Copeland's Theory

In a series of papers, the logician and philosopher B. Jack Copeland (along with several co-authors) has suggested that CS "is outgrowing its traditional foundations" (such as Turing's analysis of computation) and has called for "a 21st-century overhaul of that classical analysis" (Copeland et al., 2016, pp. 34, 36). He contrasts Turing computation with "hypercomputation" (Copeland, 2002b, p. 461): He describes a Turing computation as a computation of "functions or numbers ... with paper and pencil in a finite number of steps by a human clerk working effectively". And he defines 'hypercomputation' as "the computation of functions or numbers that cannot be computed in the sense of Turing (1936)".

Further Reading and Terminological Digression:
For more on Copeland's views, see Copeland 1997, 1998; Copeland and Proudfoot 1999—which elicited a rebuttal by Turing's biographer, Andrew Hodges, at http://www.turing.org.uk/philosophy/sciam.html—Copeland and Sylvan 1999, 2000; Copeland 2002a,b; Copeland and Shagrir 2011; and Copeland et al. 2016.

Instead of calling this 'computation', Davis (2006c, p. 4) calls it a "computation-like process". I will try to reserve the term 'compute' for Turing computation, and will use "scare quotes" to signal any kind of processing that is potentially not Turing computation. I will also use them to refer to the informal notion that is the subject of the Computability Thesis.

Piccinini (2018, p. 2) defines 'computation' as "the processing of medium independent vehicles by a functional mechanism in accordance with a rule." (See Piccinini 2015, Ch. 7, for argumentation and more details.) He explicitly cites as an advantage of this very broad definition its inclusion of "not only digital but also analog and other unconventional types of computation" (p. 3)—including hypercomputation. But Piccinini (2015, Chs. 15 & 16) also distinguishes between the "mathematical" Church-Turing Computability Thesis and a "modest physical" thesis: "Any function that is physically computable is Turing-computable" (Piccinini, 2015, p. 264), and he argues that it is an "open empirical question" (p. 273) whether hypercomputers are possible (although he doubts that they are).

Yet another form of hypercomputation, which does not fit easily into the categories of this chapter, is a *generalization* of Turing computability—which can be considered to be a branch of discrete mathematics—to the real numbers, so that there can be a theory of computation within *continuous* mathematics. This has been investigated in Blum et al. 1989 and Blum 2004. Their form of computation over the real numbers contains Turing computation as a special case.

If hypercomputable functions or numbers cannot be computed by a Turing machine, can they be "computed" at all, and, if so, how? Copeland (2002b) and Copeland and Sylvan (2000, §8.1, esp. pp. 190–191) cite the following possibilities, among others:

- The constraint of data as *symbols* on paper could be relaxed. For example, Cleland's "mundane" hollandaise-sauce recipe that we looked at in the previous chapter does not take such symbols as either input or output; instead, the inputs are certain food ingredients and the output is a certain food preparation. Indeed, any computer-controlled physical process—including robotics—seems to relax this symbolic constraint.

- The primitive operations of the "computation" might not be executable by a human *working alone*, in the way that Turing's 1936 paper described. Here, there seem to be at least two possibilities:

    1. The human needs help that can only be given by a machine capable of doing something that a human could not do even *in principle*. This might include a relaxation of the constraints about a finite number of steps or a finite amount of time, or working with what Copeland and Sylvan (2000, p. 190) call "newer physics". (See §11.4.1, below.)

2. The human needs help in the form of information that is *not pre-stored* on the tape: This might include allowing data to be supplied *during* the computation, rather than requiring it all to be pre-stored on the Turing-machine tape. This is what happens in "interactive" computing and in Turing's "oracle" machines. (See §§11.4.3 and 11.4.4, below.)

- Copeland and Sylvan identify two kinds of relativity:

1. **"Logical" relativity** concerns the use of non-classical logics, such as relevance logics (see §2.6.1.1, above). Copeland and Sylvan (2000) suggest that these might lead to hypercomputation. Perhaps; but it is certainly the case that *classical* computers can compute using relevance logics (Shapiro and Rapaport, 1987; Martins and Shapiro, 1988).
2. **"Resource" relativity** includes "relativity in procedures, methods or in the devices available for computing". This includes the "newer physics" and oracle machines just mentioned. It also includes analog computing, which we will not discuss (but see §§6.5.2 and 9.3 for some references). "Relativity in procedures" might include different basic operations or instructions (in much the same way that different geometric systems might go "beyond" straightedge and compass). Does such procedural relativity necessarily go beyond (or below) Turing computability? We'll look at this in more detail in §11.4.4.

Digression on Geometry:
Recall our earlier discussions about geometry (§§3.14.4 and 7.6.1.1). Many interesting questions in geometry (considered procedurally) concern which geometrical figures can be constructed solely with operations enabled by certain basic devices. The standard devices, of course, are compass and straightedge—more precisely, *collapsible* compass and *unruled* straightedge. (A collapsible compass is the familiar one that allows you to draw circles of different radii. A straightedge is a ruler without markings of inches or centimeters.) But different systems of geometry can be studied that allow for *measuring* devices. Famously, an angle cannot be trisected using only compass and straightedge. This is an impossibility proof on a par with the Halting Problem: Using only the primitive operations of a Turing machine, you cannot write an algorithm for deciding whether an arbitrary algorithm halts. But if you allow a measuring device (such as a protractor), angle trisection is trivial. (For discussion, see https://en.wikipedia.org/wiki/Angle_trisection.) And, as we will see, if you allow a hypercomputer, the Halting Problem can be solved! Moreover, just as there are alternative primitive operations for Turing-like machines, there are alternative primitive operations for geometry: A collapsible compass can be replaced with a fixed compass. (On this, see http://en.wikipedia.org/wiki/Compass_equivalence_theorem.)

11.3.2 Questions to Think About

There are several basic questions that need to be considered:

- Hilbert's original constraints (finiteness, etc.) seem to require "computation" to be *humanly possible* computation. So, *are* hypercomputers really alternative models of *humanly* effective procedures? (And does 'effective' need to mean "*humanly* effective"?)

- Are hypercomputers *counterexamples* to the Computability Thesis?
  Or are they just *other models* of Turing computation?
  Or are they models of a more general notion of "computation" that is, nevertheless, consistent with the Computability Thesis?

- How realistic are hypercomputers? Can they physically exist?

- Is the mind or brain a hypercomputer (rather than a Turing-machine computer)?

11.3.3 Objections to Hypercomputation

Martin Davis (2004, 2006c) thinks that most of these hypercomputers are either wrong-headed or just plain silly.

Recall the "garbage in/garbage out" principle: If you allow for incorrect input, you should expect incorrect output. Similarly, if you allow for *non-computable* input to a hypercomputer, you should expect to be able to get *non-computable* output. Davis (as we will see) argues that all examples of hypercomputation involve non-computable input.

Along the same lines, Scott Aaronson (2012) argues against hypercomputation via a parallel argument that, because Turing machines can't toast bread, a toaster-enhanced Turing machine that "allows bread as a possible input and includes toasting it as a primitive operation" would be more powerful than a classic Turing machine.

Question for the Reader:
Is this similar to Cleland's argument (see §10.4.1) that a Turing machine that can produce hollandaise sauce is more powerful than a classic Turing machine?

But might there be some *intuitively* effective, yet not *Turing-machine*–effective, "computations" that *don't* take non-computable input? Let's turn to a more detailed look at some of these options.

11.4 Kinds of Hypercomputation and Hypercomputers

> Models of hyper-computation tend to be of two general types: One uses oracles or oracles in disguise, and the other uses infinite computation in finite time. —Zenil and Hernández-Quiroz (2007, p. 5)

In this section, we'll survey a few of these systems, beginning with the second kind.

11.4.1 "Newer Physics" Hypercomputers

> According to a 1992 paper, a computer operating in a Malament-Hogarth space-time or in orbit around a rotating black hole could theoretically perform non-Turing computations. —http://en.wikipedia.org/wiki/Hypercomputation

As we saw in §8.8 (and the epigraphs for Chapter 8 on p. 297), Turing's model of computation is based on what *humans* can do. Yet it is an *idealized* human whom Turing modeled, for example, one that has no limits on *space* (recall that the tape is infinite). Cleland (2004, p. 212) points out that, in that case, one could allow other idealizations, such as no limits on *speed* of computation. Copeland (1998, p. 150) agrees: "Neither Turing nor Post, in their descriptions of the devices we now call Turing machines, made much mention of time .... They listed the primitive operations that their devices perform ... but they made no mention of the *duration* of each primitive operation".

If we relax temporal restrictions that would limit humans, then we could devise a machine that could calculate each digit of a real number's decimal expansion in half of the time of the previous digit's calculation. A "Zeus machine" is a Turing machine that "accelerates" in this way: Each step is executed in half the time of the previous step (Boolos and Jeffrey, 1974). Thus, an *infinite* calculation, including the Halting Problem, could be computed in a *finite* amount of time.

However, as Bertrand Russell (1936, p. 143) observed of a very similar example, although this is not *logically* impossible, it *is* "medically" impossible! And Scott Aaronson (2018, Slide 19) has observed that it is physically impossible for another reason:

> [O]nce you get down to the Planck time of 10^{-43} seconds, you'd need so much energy to run your computer that fast that, according to our best current theories, you'd exceed what's called the Schwarzschild radius, and your computer would collapse to a black hole. You don't want that to happen.

So, we might choose to ignore or reject Zeus machines on the grounds that they are "medically" and physically impossible. After all, no (physical, and certainly no biological) device can really accelerate in that way. But then, by parity of reasoning, should we reject ordinary Turing machines, on the grounds that they, too, are physically impossible, because, after all no (physical) device can really have an infinite tape or even an arbitrarily extendible tape? If so, and if an abstract Turing machine is mathematically possible, then, surely, so is an accelerating Turing machine. That would make a Zeus machine at least as plausible as a Turing machine.

But what about the physics of the actual world—relativity theory and quantum mechanics? The relativistic hypercomputer described in the epigraph seems far-fetched and certainly not practical. Here is what Aaronson (2018, Slides 18, 20) has to say about these:

> We can also base computers on that other great theory of the 20th century, relativity! The idea here is simple: you start your computer working on some really hard problem, and leave it on earth. Then you get on a spaceship and accelerate to close to the speed of light. When you get back to earth, billions of years have passed on Earth and all your friends are long dead, but at least you've got the answer to your computational problem. I don't know why more people don't try it!
>
> So OK, how about the TIME TRAVEL COMPUTER! The idea here is that, by creating a loop in time—a so-called "closed timelike curve"—you could force the universe to solve some incredibly hard computational problem, just because that's the only way to avoid a Grandfather Paradox and keep the laws of physics consistent. It would be like if you went back in time, and you told Shakespeare what plays he was going to write, and then he wrote them, and then you knew what the plays were because he wrote them ... like, DUDE.

As for quantum computation, the issue is whether it allows for the "computation" of *non*–Turing-computable functions or merely makes the computation of Turing-computable functions more *efficient*, perhaps by efficiently computing *NP* problems (Folger, 2016; Aaronson, 2018).

Further Reading:
Copeland (1998, p. 151, fn 2) distinguishes between "accelerating" Turing machines and "Zeus" machines. See also Copeland 2002a and Copeland and Shagrir 2011.

Davies 2001 argues that an accelerating computer could be built "in a continuous Newtonian universe" (as opposed to the Malament-Hogarth spacetime mentioned in the epigraph to this section), though not "in the real universe".

However, Cockshott and Michaelson (2007, §2.5, p. 235) reject such relaxations of the laws of classical physics out of hand. They also give an excellent summary of Turing machines and complexity theory, and they argue against a number of other hypercomputation proposals, including Wegner's interaction machines (which we'll discuss in §11.4.3.3, below).

Further Reading on Quantum Hypercomputation:
We won't examine quantum computing as an example of hypercomputation. But here are some readings (in addition to those in §3.5.4 on quantum computing in general) for readers who would like to pursue the topic:

Deutsch 1985 discusses the Computability Thesis in the context of quantum computation and proves that quantum computers, although faster than non-quantum computers, are not computationally more powerful. For brief discussion, see Bernhardt 2016, Ch. 4, especially p. 173, note 5.

Brassard 1995 is a commentary on Hartmanis 1995a (which we discussed in §3.13.2), arguing that probabilistic and—especially—quantum computers "may be qualitatively more powerful than classical machines"; see Hartmanis 1995b for a reply.

Scott Aaronson has written extensively on quantum computation: Aaronson 2006 is part of a course, "Quantum Computing Since Democritus". The first part discusses oracles and Turing reducibility in a very clear (but elementary) way, concluding that hypercomputation is not a serious objection to the Computability Thesis; later parts discuss AI. Aaronson 2008 says that "Quantum computers would be exceptionally fast at a few specific tasks, but it appears that for most problems they would outclass today's computers only modestly. This realization may lead to a new fundamental physical principle." Aaronson 2013a, is a book-length treatment that contains discussions of both quantum computing and hypercomputation. And Aaronson 2014 asks, "If there's no predeterminism in quantum mechanics, can it output numbers that truly have no pattern?"

Hodges 2012b discusses "whether all types of computation—including that of our own minds—can be modeled as computer programs", in the context of quantum computation.

Brian Hayes (2014b, p. 24) notes that "The quantum system serves as an 'oracle,' answering questions that can be posed in a format suitable for qubit computations". (For more on oracles, see §11.4.4, below.)

Further Reading on Malament-Hogarth Hypercomputation:
And for those of you curious about the epigraph to this section, here are some suggestions:

The 1992 paper cited in the *Wikipedia* article is Hogarth 1992; his machines turn out to be related to Zeus machines. The introduction is worth quoting:

> Any computer primed to perform an infinite number of computational steps must take an eternity to complete the task, because completion in a finite time would imply an unbounded signal velocity—conflicting with relativity theory. This would seem to suggest that the full potential of these computers is available only to immortal computer users. But ... there is no reason why the computer user must remain beside the computer. If he follows a different worldline his clock will tick at a rate different to that of the computer's clock, and perhaps an extreme case could be organized in which the rates are such that the finite proper time as measured by the computer user "corresponds" to an infinite proper time as measured by the computer. In this case, and granting also that the computer can always signal to the computer user, the computer user will take only a finite time to view the eternity of the computer's life and with it the results of its computations. (pp. 173–174)

And §2 ("The Story of Dave, HAL, and Goldbach")—a science fiction tale to illustrate his argument—is very much worth reading!

Button 2009 argues that Hogarth's model of hypercomputation is *not* a counterexample to the Computability Thesis.

But Manchak 2018 argues that "there [is] a clear sense in which general relativity allows for a type of machine that can bring about a space-time structure suitable for the implementation of supertasks", that is, tasks requiring "infinitely many component steps, but which in some sense is completed in a finite amount of time." For more on supertasks, see Manchak and Roberts 2016.

11.4.2 Analog Recurrent Neural Networks

A slightly different model of hypercomputation that falls somewhere between the "newer physics" and the oracle-related models is that of Hava T. Siegelmann (1995). She proposed a "*possibly* realizable" "analog shift map" or "analog recurrent neural network"—a "highly chaotic dynamical system ... which has computational power beyond the Turing limit (super-Turing); it computes exactly like neural networks and analog machines" (p. 545, my italics). 'Super-Turing' is her term for hypercomputation.

Two questions to think about in trying to understand her proposal are (1) what, if anything, neural networks and analog computers might have in common, and (2) how neural networks are different from Turing machines—more precisely, if neural-network computations are implemented on ordinary computers, whose behavior is completely analyzable in terms of Turing machines, how would something that "computes exactly like neural networks" be a *hyper*computer? More importantly, Davis (2004, pp. 8–9) shows how "the non-computability that Siegelmann gets from her neural nets is nothing more than the non-computability she has built into them".

Further Reading:
In addition to reading her paper for the details, there is a clear (but critical) description of her system in Davis 2004, pp. 6–10. And Zenil and Hernández-Quiroz 2007 offers a mathematical analysis of analog recurrent neural networks as a way to investigate whether the brain is some kind of computer.

11.4.3 Interactive Computation

11.4.3.1 Can a Program Have Zero Inputs?

Recall our discussion in §7.4.3.3 about whether an algorithm can have zero inputs. I suggested that a program to generate the decimal expansion of a real number might not require any explicit inputs. In Chapter 8, we saw Turing discuss just such algorithms. But do such algorithms really have no inputs? Or might it be better to say that there is an ambiguity in what counts as an input? After all, a program that generates the decimal expansion of a real number doesn't need any input *from the external world*, but—because any function considered as a set of ordered *pairs* must have an input *in the sense of being the first member of each such pair*—there is always an input in *that* sense. A program that has no *external* inputs would still have inputs in the functional sense. "Interactive" computation concerns programs that *do* have external inputs.

11.4.3.2 Batch vs. Online Processing

So, let's turn from physically impossible or unrealistic machines to ones that we actually deal with on a daily basis. As I have just suggested, there are (at least?) two kinds of computing: computing with *no* external inputs and computing *with* external inputs, or "batch" processing and "online" processing (Soare, 2009, §1.3, p. 370), or "computational" programs and "reactive" programs:

> Classification of Programs
>
> There are two classes of programs:
>
> Computational Programs: Run in order to produce a final result on termination. Can be modeled as a black box.
>
> $$x \rightarrow \square \rightarrow y$$
>
> Specified in terms of input/output relations. Example: The program which computes $y = 1+3+\ldots+(2x-1)$ can be specified by the requirement $y = x^2$.
>
> Reactive Programs:
>
> Programs whose role is to maintain an ongoing interaction with their environments.
>
> ...
>
> Such programs must be specified and verified in terms of their behaviors.
>
> —Amir Pnueli, "Analysis of Reactive Systems", Lecture 1 slides, http://www.cs.nyu.edu/courses/fall02/G22.3033-004/index.htm

"Batch" or "computational" processing can be understood as the behavior of a Turing machine (including a universal Turing machine):

> The classic models of computation are analogous to minds without bodies. For Turing's machine, a calculation begins with a problem on its tape, and ends with an answer there. ... How the initial tape ... is input, and how the final one is output, are questions neither asked nor answered. These theories conform to the practice of batch computing. —Wadler (1997, pp. 240–241)

"Online" or "reactive" processing has several varieties, all of which involve interaction with the external world—the world outside of the computer: A computer might have access to a (changeable) "offline" database, it might interact with the external world via sensors or effectors (or both, of course—recall Shapiro's observations about them in §3.9.3), it might interact with another computer, it might interact with a human—or any combination of these.

Arguably, even "batch-processing" Turing machines have perceptors and effectors in the sense of having a read-write head. But these are really internal to the Turing machine, and don't necessarily "reach out" to the external world. However,

> a computer linked with a human mind is a more powerful tool than an unassisted human mind. One hope of many computer scientists is to demonstrate ... that the computer/human team can actually accomplish far more than a human alone. —Forsythe (1967a, p. 3, col. 2).

One might also ask whether such a "computer/human team" could accomplish far more than a *computer* alone, say by *interacting* with the computer while it is computing (Lohr, 2013; Steed, 2013):

> [H]umans are fundamentally social animals. This insures our survival: organisms working together can do so much more than organisms working apart or in parallel. The greatest challenge for A.I. is ... the lack of attention to teaming intelligence that would allow the pairing of humans' remarkable predictive powers with A.I.'s superior bottom-up analysis of data. —Vera (2018)

Here is the rest of what Wadler has to say:

> Today, computing scientists face their own version of the mind-body problem: how can virtual software interact with the real world? In the beginning, we merely wanted computers to extend our minds: to calculate trajectories, to sum finances, and to recall addresses. But as time passed, we also wanted computers to extend our bodies: to guide missiles, to link telephones, and to proffer menus.
>
> ...
>
> Eventually, interactive models of computation emerged, analogous to minds in bodies. ... A single input at initiation and a single output at termination are now superseded by multiple inputs and outputs distributed in time and space. These theories conform to the practice of interactive computing.
>
> Interaction is the mind-body problem of computing.[3]
>
> —Wadler (1997, pp. 240–241)

Weizenbaum (1976, Ch. 5, p. 135) interestingly distinguishes between "computers" and "robots", where the latter (but not the former) "have perceptors ... and effectors".

So, are *interactive* computers—"robots", in Weizenbaum's sense—*hyper*computers?

Terminological Digression and Further Reading:
There are several names for interactive computing, including "coupled" Turing machines (each of whose outputs serve as inputs for the other; see Copeland and Sylvan 1999 and Zenil and Hernández-Quiroz 2007), "concurrent" computation (which we discuss in §11.4.3.4.3, below), and "reactive systems" (see http://en.wikipedia.org/wiki/Reactive_system).

One of the pioneers of reactive systems, Amir Pnueli, characterizes them as

> systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. Typical examples of reactive systems are air traffic control system[s], programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor.
> —syllabus, "Analysis of Reactive Systems", New York University, Fall 2002, http://cs.nyu.edu/courses/fall02/G22.3033-004/

In Knuth 2014, Question 13, Alfred Aho asks about "reactive distributed systems that maintain an ongoing interaction with their environment—systems like the Internet, cloud computing, or even the human brain. Is there a universal model of computation for these kinds of systems?"

Note that phrasing the question this way suggests that 'computation' is a very general term that includes not only Turing computation but other kinds of "computing" as well (perhaps even hypercomputation).

Knuth's answer identifies "reactive processes" with what he called "computational methods": non-terminating, single-processor algorithms (Knuth, 1973, p. 4, see §7.4, above).

Johnson and Verdicchio 2017 discuss the role of sensors and effectors in the context of autonomous AI systems.

[3] On the mind-body problem in philosophy, see §§2.8 and 12.4.6.

11.4.3.3 Peter Wegner: Interaction Is Not Turing-Computable

Peter Wegner (1997) argues that "interaction machines" are strictly more powerful than Turing machines. Wegner (1995, p. 45) identifies interaction machines with oracle machines (which we'll look at in §11.4.4) and with "modeling reactive processes" (citing work by Pneuli).

Interaction relaxes one of the "constraints" on Turing's analysis of computation: that of being "**Isolated[:]** Computation is self-contained. No oracle is consulted, and nobody interferes with the computation either during a computation step or in between steps. The whole computation of the algorithm is determined by the initial state" (Gurevich, 2012, p. 4). This certainly suggests that interactive computation is not Turing computation. On the other hand, it could also be interpreted to mean merely that computation must be "mechanical" or "automatic", and surely[4] this could include the "mechanical" or "automatic" use of input from an external source (including an oracle).

Further Reading:
Gurevich 1999, pp. 93, 98, talks both about "algorithms that are closed in the sense that they do not interact with their environments" and about ones that do so interact (§5, pp. 111–115). He also notes that, in the case of non-deterministic algorithms, "the active environment will make the choices" (p. 116). But see §11.4.3.3, below, on non-determinism.

For example, Prasse and Rittgen (1998, p. 359) consider a program such as the following:

let b be a Boolean variable;
let x be an integer variable;
begin
  $b :=$ true;
  while b = true **do**
    input(x);
    output(x^2);
    print("Should I continue? Enter true or false:");
    input(b)
end

They say of a program such as this:

> Neglecting input/output, each iteration can be interpreted as a computation performed by a Turing machine. However, the influence of the (unknown) user input on the control flow makes it hard to determine what overall function is computed by this procedure (or if the procedure can be seen as a computation at all).
> ...
> The input will be determined only at run-time. The overall function is derived by integrating the user into the computation, thus closing the system.

[4] A word that any philosopher should surely(?) take with a grain of salt! (Dennett, 2013a, Ch. 10)

> It is evident that Turing machines cannot model this behavior directly due to the missing input/output operations. Therefore, we need models that take into account inputs and outputs at run-time to describe computer systems adequately.
> —Prasse and Rittgen (1998, p. 359)

Question for the Reader:
We could easily write a Turing-machine program that would be a version of this while-loop. Consider such a Turing machine with a tape that is initialized with all of the input (a sequence of *b*s and *x*s, encoded appropriately). This Turing machine clearly is (or executes) an algorithm in the classical sense. Now consider a Turing machine with the same program (the same machine table), but with an initially blank tape and a user who inscribes appropriate *b*s and *x*s on the tape *just before each step of the program is executed* (so the Turing machine is not "waiting" for user input, but the input is inscribed on the tape *just in time*). Is there a mathematical difference between these two Turing machines? Is there anything in Turing 1936 that rules this out?

However, Prasse and Rittgen's point is that this does *not* violate the Computability Thesis, despite Wegner's interpretation:

> Interaction machines are defined as Turing machines with input and output. Therefore, their internal behavior and expressiveness do not differ from that of equivalent Turing machines. Though Wegner leaves open the question of how the input/output mechanism works, it can be assumed that input and output involve only data transport, without any computational capabilities. Therefore, the interaction machine itself does not possess greater computational power than a Turing machine. However, through communication, the computational capabilities of other machines can be utilized. Interaction can then be interpreted as a (subroutine) call.
> —Prasse and Rittgen (1998, p. 361)

Wegner and Goldin disagree and suggest that Turing disagreed, too: They discuss "Turing's assertion [in Turing 1936] that TMs have limited power and that choice machines, which extend TMs to interactive computation, represent a distinct form of computing not modeled by TMs" (Wegner and Goldin, 2006b, p. 28, col. 1, my bracketed interpolation). So, what is a "choice" machine, and how does it differ from a Turing machine?

C(hoice) machines were introduced in Turing 1936, along with *a*(utomatic)-machines (now called 'Turing machines'). As we saw in §8.10.1, *c*-machines are Turing machines that allow for "ambiguous configurations". Recall from §8.9.1 that a "configuration" is a line number together with the currently read symbol; in other words, it is the "condition" part of the condition-action expression of a Turing-machine instruction. So, an "ambiguous configuration" is a "condition" with more than one possible "action". In a *c*-machine, "an external operator" makes an "arbitrary choice" for the next action (Turing, 1936, p. 232; see our §8.10.1).

However,

> The 'Choice Machines' from Turing's paper are just what we now call nondeterministic Turing machines. In ... [Turing 1936, p. 252, footnote ‡], Turing showed that the choice machines can be simulated by traditional Turing machines, contradicting Wegner and Goldin's claim that Turing asserted his machines have limited power. —Fortnow (2006).

In other words, *non*-deterministic Turing machines are equivalent to ordinary, or *deterministic*, Turing machines. Thus, *c*-machines are no more powerful than *a*-machines, so they don't provide counterexamples to the Computability Thesis.

Fortnow (2006) notes that there is a difference between *modeling* and *simulating*. Neither of these terms have universally accepted definitions, but we can say that one way for system S_1 to *simulate* system S_2 is simply for their input-output behaviors to match (in other words, for S_1 and S_2 to compute *the same function*). And one way for S_1 to *model* (or *emulate*) S_2 is for their *internal* behaviors to match as well, that is, for S_1 to simulate S_2 *and* for their *algorithms* to match (in other words, for S_1 and S_2 to compute the same function *in the same way*). If the only way for a Turing machine to simulate a *c*-machine is by pre-storing the possible inputs, it is arguably not *modeling* it. The non-interactive Turing machine with pre-stored input (what Soare (2009, §§1.3, 9) notes is essentially a "batch" processor) can *simulate* the interactive system even if (and here, perhaps, is Wegner and Goldin's point)—it does not *model* it. Yet another pair of terms can illuminate the relationship: An interactive Turing machine may be *extensionally* equivalent to one with all input pre-stored, but it is not *intensionally* equivalent.[5]

Fortnow (2006) goes on to point out that Turing machines also only simulate but don't model many other kinds of computation, such as "random-access memory, machines that alter their own programs, multiple processors, nondeterministic, probabilistic or quantum computation." However, "Everything computable by these and other seemingly more powerful models can also be computed by the lowly one-tape Turing machine. That is the beauty of the Church-Turing thesis." The Church-Turing Computability Thesis "doesn't try to explain *how* computation can happen, just that *when* computation happens it must happen in a way computable by a Turing machine" (Fortnow, 2006, my italics).

It is important to keep in mind that, when there are two input-output–equivalent ways to do something, it still might be the case that one of those ways has an advantage over the other for certain purposes. For example, no one would want to program an airline reservation system using the programming language of a Turing machine! Rather, a high-level language (Java?, C++?, etc.) would be much more efficient. Similarly, it is easier to prove theorems *about* an axiomatic system of logic that has only *one* rule of inference (usually modus ponens), but it is easier to prove theorems *in* a natural-deduction system of logic, which has *many* rules of inference (usually at least two for each logical connective), even if both systems are logically equivalent. (See §16.5, later in this book, for more on the difference between axiomatic and natural-deduction systems of logic.)

[5] Stuart C. Shapiro, personal communication.

11.4.3.4 Can Interaction Be Simulated by a Non-Interactive Turing Machine?

11.4.3.4.1 The Power of Interaction. Nevertheless, interaction is indeed ubiquitous and powerful. Consider, for example, the following observation by Donald Knuth:

> I can design a program that never crashes if I don't give the user any options. And if I allow the user to choose from only a small number of options, limited to things that appear on a menu, I can be sure that nothing anomalous will happen, because each option can be foreseen in advance and its effects can be checked. But if I give the user the ability to write *programs* that will combine with my own program, all hell might break loose. —Knuth (2001, pp. 189–190)

That is, a program does not have to have pre-stored all possible inputs. Here is how Herbert Simon put it, commenting on the objection to AI ...

> ... "computers can only do what you program them to do." That is correct. The behavior of a computer at any specific moment is completely determined by the contents of its memory and the symbols that are input to it at that moment. This does not mean that the programmer must anticipate and prescribe in the program the precise course of its behavior. ... [W]hat actions actually transpire depends on the successive states of the machine *and its inputs at each stage of the process*—neither of which need be envisioned in advance either by the programmer or by the machine.
> —Simon (1977, p. 1187, my italics)

And those inputs are a function of the computer's interactions with the external world!

(The objection to AI that Simon quoted is a version of the "Lovelace objection, which we'll examine in more detail in a digression in §19.4.3.)

11.4.3.4.2 Simulating a *Halting* Interaction Machine. Let's consider Fortnow's position first: If an interaction machine halts, then it can be simulated by a universal Turing machine by pre-storing all of its inputs. Here's why:

In the theory of Turing computation, there is a theorem called the *S-m-n* Theorem. Before stating it, let me introduce some notation: First, recall from §8.13 that Turing machines are enumerable—they can be counted. (In fact, they are "recursively"—that is, computably—enumerable.) So, Let 'ϕ_n' represent the nth Turing machine (in some numbering scheme for Turing machines), and let i represent its input. Here is the *S-m-n* Theorem:

$$(\exists \text{ Turing machine } s)(\forall x, y, z \in \mathbb{N})[\phi_x(y,z) = \phi_{s(x,y)}(z)]$$

This says that there exists a Turing machine s (that is, a function s that is computable by a Turing machine) that has the following property: For any three natural numbers x, y, z, the following is true: The xth Turing machine, when given *both* y *and* z as inputs, produces the same output that the $s(x,y)$th Turing machine does when given *only* z as input. But what is $s(x,y)$? It is a Turing machine that already has x and y "pre-stored" on its tape!

Here is another way to say this: First, enumerate all of the Turing machines. Second, let ϕ_x be the xth Turing machine. Suppose that it takes two inputs: x and y

(another way to say this is that its *single* input is the ordered pair $\langle x, y \rangle$). Then there exists *another* Turing machine $\phi_{s(x,y)}$—that is, we can find another Turing machine that depends on ϕ's (*two*) inputs (and the dependence is itself a Turing-computable function, s)—such that $\phi_{s(x,y)}$ is input-output–equivalent to ϕ_x when y is fixed, and which is such that $\phi_{s(x,y)}$ is a Turing machine with y (that is, with part of ϕ_x's input) *stored internally as data.*

Here is an example of these two kinds of Turing machines (with a program for $\phi_x(y,z)$ on the left and a program for $\phi_{s(x,y)}(z)$ on the right:

Algorithm *x*:
begin
  input(y,z);
    {y is input from the external world}
  output := process(y,z);
  print(*output*)
end.

Algorithm *s(x,y)*:
begin
  constant := y;
    {y is pre-stored in the program}
  input(z);
  output := process(*constant*, z);
  print(*output*)
end.

In other words, any Turing machine that takes input y from the external world (or as user input) can be *simulated* by a *different* Turing machine that has y pre-stored on its tape. That is, data can be stored effectively (that is, algorithmically) in programs; the data need not be input from the external world.[6] The Turing machine that interacts with the external world can be simulated by a different Turing machine that doesn't. So, an interaction machine that halts is no more powerful than an ordinary, non-interacting Turing machine.

But keep in mind the comment at the end of §11.4.3.3 about relative advantages: The interaction machine might be more useful in practice; the non-interacting machine might be easier to prove theorems about.

[6] In our statement of the *S-m-n* Theorem, the variable z is also being input from the external world, but it is only there for technical reasons required for the proof of the theorem in the most general case. In practice, z can also be pre-stored on the tape, or even omitted.

Further Reading:
The *S-m-n* Theorem was first stated and proved by Kleene (1952, Theorem XXIII, p. 342). It gets its name from the form that Kleene expressed it in, using a function that he called 'S_n^m'. It is also sometimes called the "Parameter Theorem" or the "Iteration Theorem" (Davis and Weyuker, 1983, p. 64). Rogers (1967, Theorem IV, p. 22) calls it the "enumeration theorem" (but distinguishes it from what *he* calls the "s-m-n theorem" (Rogers, 1967, Theorem V, pp. 23–24)).

My interpretation of the theorem is due to John Case (nd). (In lectures at the University at Buffalo in the early 1980s, Case used the mnemonic "**S**tuff-e**m**-i**n** theorem" to emphasize that the external inputs could be "stuffed into" the computer program.) Similar interpretations can be found in Cooper 2004, p. 64, and Homer and Selman 2011, p. 53. And Kfoury et al. 1982, p. 82, give a version of it stated in terms of computer programs.

Cooper (2004, p. 64) also notes that the existence of the universal Turing machine can be expressed—using our notation above (instead of his)—by taking y as a ("hardwired", non-universal) Turing machine and $s(x,y)$ as a universal Turing machine with y stored on its tape (as its software). And Rogers (1967, p. 23) notes that it "shows that the computing agent ... need not be human"—because the human "computing agent" in Turing's informal analysis can be replaced by a universal Turing machine.

So, any *interactive* program that halts could, in principle, be shown to be logically equivalent to a *non*-interactive program. That is, any interactive program that halts can be simulated by an "ordinary" Turing machine by pre-storing the external input:

> An interactive system is a system that interacts with the environment via its input/output behavior. The environment of the system is generally identified with the components which do not belong to it. Therefore, an interactive system can be referred to as an open system because it depends on external information to fulfil its task. *If the external resources are integrated into the system, the system no longer interacts with the environment and we get a new, closed system.* So, the difference between open and closed systems 'lies in the eye of the beholder'.
> —Prasse and Rittgen (1998, p. 359, col. 1, my italics);
> (Teuscher and Sipper, 2002, p. 24, make a similar observation)

The catch here is that you need to know "in advance" what the external input is going to be. Halting is important here, because, once the interactive machine halts, all of its inputs are known and can then be pre-stored on the simulating machine's tape. But the *S-m-n* Theorem does say that, once you know what that input is, you need only an ordinary Turing machine, not an interactive hypercomputer.

Philosophical Digression:
"Solipsism", as defined by Bertrand Russell (1927, p. 398), is "the view that from the events which I experience there is no valid method of inferring the character, or even the existence, of events which I do not experience." It is occasionally parodied as the view that *I* am the only thing that exists; *you* are all figments of my imagination. Note that *you* cannot make the same claim, because, after all, if solipsism is true, then you don't exist! There's a story that, at a lecture that Bertrand Russell once gave on solipsism, someone in the audience commented that it was such a good theory, why didn't more people believe it? Actually, solipsism is not really the claim that only I exist. Rather, it is the claim that I live in a world of my own, completely cut off from the external world, and so do you. This is reminiscent of the philosopher Gottfried Leibniz's "monads" (Leibniz 1714, https://en.wikipedia.org/wiki/Monadology), but that's beyond our present scope.

"*Methodological* solipsism" is a view in the philosophy of mind and of cognitive science that says that, to understand the "psychology" of a cognitive agent, it is not necessary to specify the details of the external world in which the agent is situated and which impinge on the agent's sense organs. This is not to deny that there *is* such a world or that there is such sensory input—hence the qualifier 'methodological'. Rather, it is to acknowledge (or assume) that all that is of interest psychologically or cognitively can be studied from the surface inwards, so to speak (Putnam, 1975; Fodor, 1980). That is, cognition can be studied by acting as if the brain (or the mind) only does "batch processing". (We'll come back to this in §§17.9 and 19.6.3.2.)

Consider an AI system that can understand and generate natural-language and that gets its input from the external world, that is, from a user. The point of methodological solipsism is that we could *simulate* this by building in the input (assuming a finite input). Indeed, this can be done for *any* partial recursive function, according to the *S-m-n* Theorem. If we understand methodological solipsism as the *S-m-n* Theorem, we would have an argument for methodological solipsism from the theory of computation!

11.4.3.4.3 Simulating a *Non-Halting* Interaction Machine. But suppose that our interaction machine does *not* halt—not because of a pernicious infinite loop, but (say) because it is running an operating system or an automated teller machine; such machines only halt when they are broken or being repaired.

> ***Interactive computing.*** Many systems, such as operating systems, Web servers, and the Internet itself, are designed to run indefinitely and not halt. Halting is an abnormal event for these systems. The traditional definition of computation is tied to algorithms, which halt. Execution sequences of machines running indefinitely seem to violate the definition. —Denning (2010, p. 5)

Stuart C. Shapiro has said[7] that such programs don't express algorithms, because algorithms, by definition, must halt. But even Turing's original Turing machines didn't halt: They computed infinite decimals, after all! The central idea behind the Halting Problem is to find an algorithm that distinguishes between programs that halt and those that don't. Whether halting is a Good Thing is independent of that.

[7] Personal communication, but see Shapiro 2001 and the discussion in §3.9.3, above.

Of course, any stage in the process is a finite (that is, halting) computation. (Recall Prasse and Rittgen's first sentence, quoted on p. 452, above.) Even Turing's computation of reals is a (non-halting) sequence of halting computations of successive terms of the decimal expansion.

There are two non-halting cases to consider. In the first case, the unending input stream is a number computable by a universal Turing machine. In this case, the interaction machine can *also* be simulated by a universal Turing machine. Hence, interaction in this case also does not go beyond the Computability Thesis, because—being computable—the inputs are "knowable—that is, computable—in advance". So, instead of pre-storing the individual *inputs*, we can simply pre-store a copy of the *program* that generates those inputs.

In the second case, suppose that, not only does the Turing machine not halt, but the unending input stream is *not* computable by a Turing machine. Then the interaction machine is a hypercomputer. It is only this situation—where the input is *non*-computable (hence, *not* knowable in advance, even in principle)—that we have hypercomputation.

But is it? Or is it just an oracle machine? We will see in §11.4.4 why it is not obvious that oracle-machine computation is "hyper" in any interesting sense, either.

Why might such a non-halting, non-computable, interaction machine be a hypercomputer? Its input stream might be random. Truly random numbers are not computable (Church, 1940). (For a related discussion of randomness and computability, see Chaitin 2006a.) But de Leeuw et al. 1956 showed that "the computing power of Turing machines provided with a random number generator ... could compute only functions that are already computable by ordinary Turing machines" (Davis, 2004, p. 14).

Even if not random, the input stream of such an interaction machine might be non-computable. According to Copeland and Sylvan (1999, p. 51), "A coupled Turing machine is the result of coupling a Turing machine to its environment via one or more input channels. Each channel supplies a stream of symbols to the tape as the machine operates." They give a simple proof (p. 52) that there is a coupled Turing machine "that cannot be simulated by the universal Turing machine". However, the proof involves an oracle that supplies a non-Turing computable real number, so their example falls prey to Davis's objection.

Further Reading:
van Leeuwen and Wiedermann 2000 takes up the challenge of formalizing Wegner's ideas about interaction machines, arguing that " 'interactive Turing machines with advice' ... are more powerful than ordinary Turing machines."

For a debate between a hypercomputation skeptic and a believer, see http://c2.com/cgi/wiki?InteractiveComputationIsMorePowerfulThanNonInteractive (2014).

For more by Wegner and his colleagues, see the following:

1. Wegner 1999 (in Longo 1999, the special issue of *The Monist* on philosophy of CS, cited in §1.4, above).
2. Wegner and Goldin 1999 (an unpublished attempt at formalizing some of Wegner's claims about interaction machines).
3. Wegner and Goldin 2003 discusses "Computation beyond Turing Machines". Goldin and Wegner 2004 is a sequel to Wegner 1997 and Wegner and Goldin 2003.
4. Goldin et al. 2004 "present[s] *Persistent Turing Machines* (PTMs), a new way of interpreting Turing-computation, one that is both interactive and persistent. ... It is persistent in the sense that the work-tape contents are maintained from one computation ... to the next" (from the abstract).
5. Goldin and Wegner 2008, which claims to refute the Computability Thesis.

11.4.3.4.4 Concurrent Computation. Here is another possibility: "Concurrent computation" can be thought of in two ways. In the first way, a single computer (say, a (universal) Turing machine) executes two different algorithms, not one after the other ("sequentially"), but in an interleaved fashion, in much the way that a parallel computation can be simulated on a serial computer: On a serial computer (again, think of a Turing machine), only one operation can be executed at a given time; on a parallel computer, two (or more) operations can be executed simultaneously. But one can simulate parallel computation on a serial computer by interleaving the two operations.

The difference between parallel and concurrent computation is that the latter can also be thought of in a second way: Two (or more) computers (say, non-universal Turing machines) can each be executing two different, single algorithms simultaneously in such a way that outputs from one can be used as inputs to the other. This kind of concurrent computation is what Copeland calls "coupled Turing machines".

Is concurrent computation a kind of hypercomputation? It is, if "interactive computation" is the same as that second way of thinking about concurrent computation, *and* if interactive computation is hypercomputation. At least one important theoretician of concurrent computing has said that "concurrent computation ... is in a sense the *whole* of our subject—containing sequential computing as a well-behaved special area" (Milner, 1993, p. 78) This is consistent with those views of hypercomputation that take it to be a generalization of (classical) computation. Where (classical) computation is based on an analysis of mathematical functions, Milner argues that concurrent computation must be based on a different mathematical model; that also suggests that it is, in some sense, "more" than mere classical computation. Interestingly, however, he also says that this different mathematical model must "match the functional calculus not by copying its constructions, but by emulating two of its attributes: It is

synthetic—we build systems in it, because the structure of terms represents the structure of processes; **and it is *computational*—its basic semantic notion is a step of computation**" (Milner, 1993, p. 82; italics in original, my boldface). Again, it appears to be a generalization of classical computation.

Unlike certain models of hypercomputation that either try to do something that is physically (or "medically") impossible or can only exist in black holes, concurrent computation seems more "down to earth"—more of a model of *humanly* possible processing, merely analyzing what happens when Turing machines interact.

In fact, here's an interesting way to think about it: A single Turing machine can be thought of as a model of a human computer solving a single task. But humans can work on more than one task at a time (concurrently, if not in parallel). And two or more humans can work on a single problem at the same time. Moreover, there is no reason why one human's insights ("outputs") from work on one problem couldn't be used (as "inputs") in that same human's work on a distinct problem. And there is surely no reason why one human's insights from work on a given problem couldn't be used by another human's work on that same problem. Surely even Hilbert would have accepted a proof of some mathematical proposition that was done jointly by two mathematicians. So, insofar as concurrent computation is a kind of hypercomputation, it seems to be a benign kind.

This suggests that the proper way to consider such interactive (or reactive, coupled, or concurrent) systems is *not* as some new, or "hyper", model of computation, but simply as the study of what might happen when *two or more* Turing machines interact. There's no reason to expect that they would not be more powerful *in some sense* than a single Turing machine. Clearly, if a Turing machine interacts with something that is *not* a Turing machine, then hypercomputation can be the result; this is the gist of Davis's objection.

The only case in which an interaction machine differs significantly from a Turing machine is when the interactive machine doesn't halt *and* its input stream allows for an oracle-supplied, non-computable number. But this seems to fall under Davis's objection to hypercomputation.

Further Reading:

Frenkel 1993 is a companion piece to Milner 1993, with interesting observations on AI, the semantics of programming languages, program verification, and the nature of CS.

A 1994 e-mail discussion between Peter Wegner and Carl Hewitt (a major figure in concurrent computing) is available at http://www.cse.buffalo.edu/~rapaport/510/actors-vs-church-thesis.txt

Schächter 1999 asks "How Does Concurrency Extend the Paradigm of Computation?".

For more information on concurrency, see http://en.wikipedia.org/wiki/Concurrent_computing and http://en.wikipedia.org/wiki/Concurrency_(computer_science).

11.4.4 Oracle Computation

> Let us suppose that we are supplied with some unspecified means of solving number-theoretic problems; a kind of oracle as it were. We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine. With the help of the oracle we could form a new kind of machine (call them *o*-machines), having as one of its fundamental processes that of solving a given number-theoretic problem. More definitely these machines are to behave in this way. The moves of the machine are determined as usual by a table except in the case of moves from a certain internal configuration o. If the machine is in the internal configuration o and if the sequence of symbols marked with *l* is then the well-formed formula **A**, then the machine goes into the internal configuration p or t according as it is or is not true that **A** is dual. The decision as to which is the case is referred to the oracle.[8] —Turing (1939, pp. 172–173)

An *o*(racle)-machine is a Turing machine that can "interrogate an '*oracle*' (*external database*) during the computation" (Soare, 2009, §1.3, p. 370) in order to determine its action (including its next configuration). Moreover, the database "cannot be a machine" (Turing, 1939, p. 173). If it were a "machine"—presumably an *a*-machine—then its behavior would be computable, and vice versa.

Further Reading:
Oracle machines were first described in Turing's Ph.D. dissertation at Princeton (which was completed in 1938, and which he began *after* his classic 1936 paper was published; Church was his dissertation advisor. His dissertation can be read online at http://www.dcc.fc.up.pt/~acm/turing-phd.pdf; it was published as Turing 1939, from which this section's epigraph was taken.

However, if the choice made by the oracle *were* computable, then *c*-machines could be considered as a special case of *o*-machines. If interaction is best modeled by an *oracle* machine, then Wegner and Goldin are incorrect about *choice* machines being the ones that "extend" Turing machines "to interactive computing" (see §11.4.3.3, above). In fact, according to Davis (1958, pp. 20–24), Turing machines "deal ... only with closed computations. However, it is easy to imagine a machine that halts a computation at various times and requests additional information." He then discusses relative computation and *o*-machines in the form of Turing machines that can ask whether a given integer is an element of a given set, observing that "This provides a Turing machine with a means of communication with 'the external world.' "

The external database is a "black box" that could contain the answers to questions that are not computable by an ordinary Turing *a*-machine. If a function g is computable by an *o*-machine whose oracle outputs the value of a (non–Turing-computable) function f, then it is said that g is computable *relative to* f.

[8] Here is Turing's explanation of some of the technical terms in this passage: "Every number-theoretic theorem is equivalent to a statement of the form '**A(n)** is convertible to 2 for every W.F.F. **n** representing a positive integer', **A** being a W.F.F. determined by the theorem; the property of **A** here asserted will be described briefly as '**A** is dual' " (p. 170). "Convertibility" is an equivalence relation in Church's lambda calculus.

The computer scientist Solomon Feferman (1992, p. 340, footnote 8) said this: "Several people have suggested to me that *interactive computation* exemplifies Turing's 'oracle' in practice. While I agree that the comparison is apt, I don't see how to state the relationship more precisely." However, Bertil Ekdahl has a nice example that illustrates how interactive computing is modeled by *o*-machines and relative computability. The essence of the example considers a simplified version of an airline-reservation program. Such a program is a standard example of the kind of interactive program that Wegner claims is not Turing computable, yet it is not obviously an *o*-machine, because it does not obviously ask an oracle for the solution to a non-computable problem. Suppose our simplified reservation program is this:

```
while true do
  begin
    input(passenger, destination);
    output(ticket(passenger, destination))
  end
```

Ekdahl observes that, although writing the passenger and destination information on the input tape is computable "and can equally well be done by another Turing machine", when our reservation program then " 'asks' for two new" inputs, "**which** [inputs are] going to [be written] on the tape *is not a recursive process*. ... So, the input of [passenger and destination] can be regarded as a question to an *oracle*. An oracle answers questions known in advance but the answers are not possible to reckon in advance" (Ekdahl, 1999, §3, pp. 262–263, italics in original; my boldface and interpolated brackets).

Conceivably, the "computation-like process" performed by the physics-challenging machines described in §11.4.1, above, can also be simulated (if not modeled) by oracle machines. So, the hypercomputation question seems to come down to whether *o*-machines violate the Computability Thesis. Let's look at them a bit more closely.

Feferman (1992, p. 321) notes that *o*-machines can be "generalized to that of a B-machine for any set B". Instead of Turing machines, Feferman discusses the logically equivalent register machines of Shepherdson and Sturgis 1963, which we mentioned in §9.4.1. Briefly, a register machine consists of "registers" (storage units), each of which can contain a natural number. In Feferman's version (1992, p. 316), for each register r_i, the machine has four basic operations:

1. $r_i := 0$
2. $r_i := r_i + 1$
3. **if** $r_i \neq 0$, **then** $r_i := r_i - 1$
4. **if** $r_i = 0$, **then** go to instruction j **else** go to instruction k

To turn this into a B-machine, we add one more kind of operation (p. 321):

5. **if** $r_k \in B$, **then** $r_i := 1$ **else** $r_i := 0$. In other words, a B-machine is an *o*-machine: a Turing machine together with a set B that plays the role of the oracle. The machine's program can consult oracle B to see if it contains some value r_k. The fifth operation puts a 1 or a 0 into register r_i if the oracle tells it that the value $r_k \in B$.

Essentially, this adds primitive operations to a Turing machine (or a register machine). If these operations can be simulated by the standard primitive operations of the Turing machine, then we haven't increased its power, only its expressivity, essentially by the use of named subroutines. (Recall Prasse and Rittgen's observation that "interaction can ... be interpreted as a (subroutine) call".) Turing's *o*-machines are of this type; the call to a (possibly non-computable) oracle is simply a call to a (possibly non-computable) subroutine. So, as Prasse and Rittgen say, "the machine *itself*" is just a Turing machine, and, as Davis would say, if a non-computable input is encoded in B, then a non-computable output can be encoded on its tape. If B contains the answers to problems not solvable by the Turing machine, then, of course, we have increased the machine's power.

But does that provide a counterexample to the Computability Thesis?

In fact, Feferman (1992, pp. 339–340) observes that the "built-in functions" of "actual computers" (for example, the primitive recursive functions or the primitive operations of a Turing machine) are "given by a 'black box'—which is just another name for an 'oracle'—and a program to compute a function f from one or more of these" built-in functions "is really an algorithm for computation of f relative to" those built-in functions.

To say that a set A is Turing computable from (or "Turing reducible to") a set B (written: $A \leq_t B$) is to say that x is in A iff the B-machine outputs 1 when its input is x (where output 1 means "yes, x is in A"). Davis (2006b, p. 1218) notes that, where A and B are sets of natural numbers, if $A \leq_t B$, and "if B is itself a computable set, then nothing new happens; in such a case $A \leq_t B$ just means that A is computable. But if B is non-computable, then interesting things happen."[9] According to Davis (2006c), one of the *un*interesting things, of course, is that A will then turn out to be non-computable. The *interesting* things have to do with "degrees" of *non*-computability: "can one non-computable set be more non-computable than another?" (Davis, 2006b, p. 1218).

What does that mean? As we hinted at earlier, Gödel's Incompleteness Theorem shows that there is a true statement of arithmetic that cannot be proved from Peano's axioms. What if we add that statement as a new axiom? Then we can construct a different true statement of arithmetic that cannot be proved from this new set of axioms. And we can continue on in this matter, constructing ever more powerful theories of arithmetic, with no end. Turing's dissertation and invention of oracles essentially applied the same kind of logic to computability:

> It is possible to posit the existence of an oracle, which computes a non-computable function, such as the answer to the halting problem or some equivalent. Interestingly, the halting problem still applies to such machines; that is, although they can determine whether particular Turing machines will halt on particular inputs, they cannot determine whether machines with equivalent halting oracles will themselves halt. This fact creates a hierarchy of machines according to their Turing degree, each one with a more powerful halting oracle and an even more difficult halting problem. ...
>
> With such a method, an infinite hierarchy of computational power can easily be constructed by positing the existence of oracles that perform progressively

[9] Recall from §11.4.3.4.1, above, Knuth's expression for a similar situation: "all hell might break loose"! (Knuth, 2001, pp. 189–190)

> more complex computations which cannot be performed by machines that incorporate oracles of lower power. Since a conventional Turing machine cannot solve the halting problem, a Turing machine with a Halting Problem Oracle is evidently more powerful than a conventional Turing machine because the oracle can answer the halting question. It is straightforward to define an unsolvable halting problem for the augmented machine with the same method applied to simpler halting problems that lead to the definition of a more capable oracle to solve that problem. This construction can be continued indefinitely, yielding an infinite set of conceptual machines With such a method, an infinite hierarchy of computational power can easily be constructed by positing the existence of oracles that perform progressively more complex computations which cannot be performed by machines that incorporate oracles of lower power. Since a conventional Turing machine cannot solve the halting problem, a Turing machine with a Halting Problem Oracle is evidently more powerful than a conventional Turing machine because the oracle can answer the halting question. It is straightforward to define an unsolvable halting problem for the augmented machine with the same method applied to simpler halting problems that lead to the definition of a more capable oracle to solve that problem. This construction can be continued indefinitely, yielding an infinite set of conceptual machines. ...
>
> In other words, two sets of natural numbers have the same Turing degree when the question of whether a natural number belongs to one can be decided by a Turing machine having an oracle that can answer the question of whether a number belongs to the other, and vice versa. So the Turing degree measures precisely the computability or incomputability of X. Turing reducibility induces a partial order on the Turing degrees. —Zenil and Hernández-Quiroz (2007, pp. 6–7)

Consequently, Feferman (1992, p. 321, my interpolated brackets) observes that

> the arguments for the Church-Turing Thesis lead one strongly to accept a relativized version: (C-T)^r [a set] A is effectively computable from [a set] B if (and only if) $A \leq_T B$.

Feferman then says that "Turing reducibility gives the most general concept of relative effective computability" (p. 321).

And here is Feferman on the crucial matter:

> Uniform global recursion provides a much more realistic picture of computing over finite data structures than the absolute computability picture, for finite data bases are constantly being updated. As examples, we may consider ... airline reservation systems. —Feferman (1992, p. 342)

He does, however, go on to say that "while notions of relativized (as compared to absolute) computability theory are essentially involved in actual hardware and software design, the bulk of methods and results of recursion theory have so far proved to be irrelevant to practice" (Feferman, 1992, p. 343). That certainly is congenial to Wegner's complaints.

On the other hand, Feferman (1992, p. 315) also claims that "notions of relative (rather than absolute) computability" (that is, notions based on Turing's o-machines rather than on his a-machines) have "primary significance for practice" and that these

relative notions are to be understood as "generalization[s] ... of computability [and "of the Church-Turing Thesis"] to arbitrary structures". So this seems to fly in the face of Wegner's claims that interaction is something *new*, while agreeing with the substance of his claims that interaction is more central to modern computing than Turing machines are.

Soare agrees:

> Almost all the results in theoretical computability use relative reducibility and o-machines rather than a-machines and most computing processes in the real world are potentially online or interactive. Therefore, we argue that Turing o-machines, relative computability, and online computing are the most important concepts in the subject, more so than Turing a-machines and standard computable functions since they are special cases of the former and are presented first only for pedagogical clarity to beginning students. —Soare (2009, Abstract, p. 368)

This is an interesting passage, because it could be interpreted by hypercomputation advocates as supporting their position and by anti-hypercomputationalists as supporting theirs!

In fact, a later comment in the same paper suggests the pro-hypercomputational reading:

> The original implementations of computing devices were generally offline devices such as calculators or batch processing devices. However, in recent years the implementations have been increasingly online computing devices which can access or interact with some external database or other device. The Turing o-machine is a better model to study them because the Turing a-machine lacks this online capacity. —Soare (2009, §9, p. 387)

He also says (referring to Turing 1939 and Post 1943),

> The theory of relative computability developed by Turing and Post and the o-machines provide a precise mathematical framework for database [or interactive] or online computing just as Turing a-machines provide one for offline computing processes such as batch processing. —Soare (2009, §1.3, pp. 370–371).

And he notes that oracles can model both client-server interaction as well as communication with the Web. However, the interesting point is that all of these are *extensions* of Turing machines, not entirely new notions. Moreover, Soare does not disparage, object to, or try to "refute" the Computability Thesis; rather, he celebrates it (Soare, 2009, §12).

This certainly suggests that some of the things that Copeland and Wegner say about hypercomputation are a bit hyperbolic; it suggests that both the kind of hypercomputation that takes non-computable input (supplied by an oracle) to produce non-computable output as well as the kind that is interactive are both well-studied and simple extensions of classical computation theory.

Soare's basic point on this topic seems to be this:

> **Conclusion 14.3** The subject is primarily about incomputable objects not computable ones, and has been since the 1930's. The single most important concept is

> that of relative computability to relate incomputable objects.
> —Soare (2009, §14, p. 395)

> Turing's oracle machine was developed by Post into Turing reducibility .... It is the most important concept in computability theory. Today, the notion of a local machine interacting with a remote database or remote machine is central to practical computing. —Soare (2012, p. 3290)

This is certainly in the spirit of hypercomputation without denigrating the Computability Thesis.

Further Reading:
Dennett 1995, p. 445 suggests that oracles are like his notion of "skyhooks" (which we described in §3.14.7)—magically going beyond algorithms.

For more on oracles and relative computability, see Post's original paper (Post, 1944, §11) and Dershowitz and Gurevich 2008, §5, pp. 329f. An excellent, relatively informal overview for philosopher of relative computability for philosophers is Jenny 2018, §2.

Piccinini 2003 is primarily about Turing's views on AI, but also discusses his theory of computation and the role of "oracle" machines.

Arkoudas 2008 (especially §3) argues against Copeland that a version of the Computability Thesis that is immune to hypercomputation objections follows from the "systematic predictability of observable behavior" and that even *o*-machines are, in fact, "deterministic digital computers".

11.4.5 Trial-and-Error Computation

11.4.5.1 Introduction

There is one more candidate for hypercomputaiton that is worth looking at for its intrinsic interest. It goes under many names: "trial-and-error computation", "inductive inference", "Putnam-Gold machines", and "limit computation". Here, the "constraint" that is relaxed is that we change our interpretation of what counts as the output of the "computation".

Further Reading:
Philosophers, logicians, and computer scientists have all contributed to this theory. Hilary Putnam (1965) originated the term 'trial-and-error *predicate*', and Hintikka and Mutanen (1997) echoed that in 'trial-and-error *computability*'. E. Mark Gold (1967) called such computation 'inductive inference'. Peter Kugel (2002) claims to have originated the terms 'trial-and-error *machine*' and 'Putnam-Gold machine'. Robert I. Soare (2009, §9.2, p. 388) refers to Putnam's trial-and-error predicates as "limit computable functions".

Here is how Putnam introduced "trial and error predicates": First, a "predicate" can be thought of as a Boolean-valued function. Next, as in §7.6.2.2, we'll let the notation $\overline{x}$ represent an n-tuple of variables $x_1, \ldots, x_n$, for some n. Then (paraphrasing Putnam

1965, p. 49) a predicate P is a *trial and error predicate* $=_{def}$ there is a computable function f such that, for every $\bar{x}$,

$$P(\bar{x}) = 1 \text{ iff } \lim_{y\to\infty} f(\bar{x},y) = 1,$$

and

$$P(\bar{x}) = 0 \text{ iff } \lim_{y\to\infty} f(\bar{x},y) = 0,$$

where

$$\lim_{y\to\infty} f(\bar{x},y) = k \;\; =_{def} \;\; \exists w \forall z[z \geq w \supset f(\bar{x},z) = k]$$

In other words, no matter what initial value (or values) that the function f takes, the predicate P is true (or false) iff, in the "limit" (that is, at w or "beyond"), the function $f = 1$ or $(f = 0)$. As Welch (2007, p. 770) puts it, "the eventual value of" f is 1 or 0.[10]

That is, Putnam "modifies" the notion of Turing computability ...

> ... by (1) allowing the procedure to "change its mind" any finite number of times (in terms of Turing Machines: we visualize the machine as being given an integer (or an n-tuple of integers) as input. The machine then "prints out" a finite sequence of "yesses" and "nos". The *last* "yes" or "no" is always to be the correct answer.); and (2) we give up the requirement that it be possible to tell (effectively) if the computation has terminated[.] I.e., if the machine has most recently printed "yes", then we know that the integer put in as input must be in the set *unless the machine is going to change its mind*; but we have no procedure for telling whether the machine will change its mind or not.
>
> The sets for which there exist decision procedures in this widened sense are decidable by "empirical" means—for, if we always "posit" that the most recently generated answer is correct, we will make a finite number of mistakes, but we will eventually get the correct answer. (Note, however, that even if we have gotten to the correct answer (the end of the finite sequence) we are never *sure* that we have the correct answer.) —Putnam (1965, p. 49)

In general, a trial-and-error machine is a Turing machine with input i that outputs a sequence of responses such that it is the *last* output that is "the" desired output of the machine (rather than the first, or only, output). But you don't allow any way to tell effectively if you've actually achieved the desired output, that is, if the machine has really halted.

The philosopher and psychologist William James once said, in a very different context, that ...

> ... the faith that truth exists, and that our minds can find it, may be held in two ways. We may talk of the *empiricist* way and of the *absolutist* way of believing in truth. The absolutists in this matter say that we not only can attain to knowing truth, but we can *know when* we have attained to knowing it; whilst the empiricists think that although we may attain it, we cannot infallibly know when. To *know* is

[10] To clarify: Function f takes as input an $n+1$-tuple of natural numbers (there are n xs plus 1 y), and that outputs a natural number. Each value of y ($y = 0, 1, 2, \ldots$) will, in general, yield a different value for f, but, at some point (at w, in fact), no matter how large y gets, f will remain constant with value k.

> one thing, and to know for certain *that* we know is another.
> —James (1897, §V, p. 465)

To paraphrase James:

> The faith that a problem has a computable (or algorithmic) solution exists, and that our computers can find it, may be held in two ways. We may talk of the trial-and-error way and of the Turing-algorithmic way of solving a problem. The Turing algorithmists in this matter say that we (or Turing machines) not only can solve computable problems, but we can know when we (or they) have solved them; while the trial-and-error hypercomputationalists think that although we (or our computers) may solve them, we cannot infallibly know when. For a computer to produce a solution is one thing, and for us to know for certain that it has done so is another.

Recall from §8.10.2.1 that Turing called the marks printed by a Turing machine that were *not* to be taken as output "symbols of the second kind", used only for bookkeeping. Peter Kugel (1986a) takes up this distinction:

> We distinguish an output from a result. An output is anything M ["an idealized general-purpose computing machine"] prints, whereas a result is a selection, from among the things it prints, that we agree to pay attention to. ... The difference between a computing procedure and a trial and error procedure is this[:] When we run M_p [M running under program p] as a computing procedure, we count its *first* output as its result. When we run it as a trial and error procedure, we count its *last* output as its result. —Kugel (1986a, pp. 139–140, my interpolated brackets).

(It is interesting to compare such machines to heuristic programming. In §3.15.2.3, we saw that a heuristic program was one that gave an approximate solution to a problem. It would seem that trial-and-error machines do just that: Each output is an approximation to the last output. Ideally, each output is a closer approximation than each previous output.)

In a similar vein, Kugel (2002) notes that a distinction can be made between a Turing *machine* and Turing *machinery*. Sub-Turing computation, although not *requiring* all the power of a Turing *machine*, can be accomplished using Turing *machinery*. As Hintikka and Mutanen (1997, p. 175, my interpolated brackets) put it, "there is more than one sense in which the same idealized hardware [that is, Turing *machinery*] can be used to compute a function". (Here, 'compute' does not refer to Turing computation, because trial-and-error computability "is wider than recursivity".) Gödel made a similar observation (though he wasn't talking about trial-and-error machines): "A sometimes unsuccessful procedure, if sharply defined, still is a procedure, that is a well-determined manner of proceeding" (quoted in Stewart Shapiro 2013, p. 174).

In a Turing *machine*, the *first* output is the result of its computation. But there is nothing preventing the use of Turing *machinery* and taking the *last* output of its operation as its result. You can't say that the operation of such Turing *machinery* is computation, if you accept the Computability Thesis, which identifies computation with the operation of a Turing machine. But if a trial-and-error machine can do super-Turing "computation", then it would be a hypercomputer that uses Turing *machinery* (and would not require "newer physics").

Recall our discussion of the Halting Problem from Chapter 7. In §7.7.1, we contrasted two alleged algorithms for determining whether a program *C* halts on input *i*:

Algorithm $A_H^1(C,i)$:	**Algorithm** $A_H^2(C,i)$:
begin	**begin**
if $C(i)$ halts	output 'loops'; {that is, make an initial *guess* that *C* loops}
then output 'halts'	**if** $C(i)$ halts
else output 'loops'	**then** output 'halts'; {that is, *revise* your guess}
end.	**end.**

Algorithm A_H^1 *can* be converted to the self-referential A_H^{1*} and thereby used in order to show that the Halting Problem is not Turing computable (see §7.7.2.1).

But A_H^2 could *not* be so converted. It is an example of a trial-and-error "algorithm": It makes an initial guess about the desired output, and then keeps running program *C* on a number of "trials"; if the trials produce "errors" or don't come up with a desired response, then continue to run more trials.

As Hintikka and Mutanen (1997, p. 181) note, the Halting Problem algorithm in its trial-and-error form is not computable, "even though it is obviously mechanically determined in a perfectly natural sense." They also note that this "perfectly natural sense" is part of the informal notion of computation that the Computability Thesis asserts is identical to Turing computation, and hence they conclude that the Computability Thesis "is not valid" (p. 180). (Actually, they're a bit more cautious, claiming that the informal notion is "ambiguous": "We doubt that our pretheoretical ideas of mechanical calculability are so sharp as to allow for only one explication" (p. 180).)

So, a trial-and-error machine uses Turing *machinery* to perform hypercomputations. However, trial-and-error computation is equivalent to computations by *o*-machines that solve the halting problem!

> If the computation is to determine whether or not a natural number *n* as input belongs to some set *S*, then it turns out that sets for which such "trial and error" computation is available are exactly those ... that are computable relative to ... an oracle that provides correct answers to queries concerning whether a given Turing machine ... will eventually halt. —Davis (2006a, p. 128)

So, trial-and-error computation falls prey to the same objections as other forms of hypercomputation. However, because trial-and-error computation only requires an ordinary, physically plausible Turing machine and no special oracle, it does have some other uses, which are worth looking at. Whether these are legitimate kinds of hypercomputation is something left for you to decide!

11.4.5.2 Does "Intelligence" Require Trial-and-Error Machines?

A trial and error machine can "compute" the uncomputable, but we can't reliably use the result. But what if we *have* to? When we learn to speak, we don't wait (we *can't* wait) until we fully understand our language before we start (before we *have* to start) to use it. Similarly, when we reason or make plans, we must also draw conclusions or act on the basis of incomplete information. Herbert Simon (1996a) called this "satisficing" or "bounded rationality" (see §§2.6.1.5 and 5.7, above).

One of the claims of hypercomputationalists is that some phenomena that are not Turing computable are (or might be) "computable" in some extended sense. And one of these phenomena is "intelligence", or cognition. Siegelman's version that we looked at in §11.4.1, based on neural networks, is one of these. Another, based on trial-and-error computation, is what we will look at now.

Terminological Digression:
'Intelligence' is the term that many people use—including, famously, Turing (1950)—and it is enshrined in the phrase 'artificial intelligence'. However, I prefer the more general term 'cognition', because the concept that both terms attempt to capture has little or nothing to do with "intelligence" in the sense of IQ tests. So, when you see the words 'intelligence' or 'intelligent' below, try substituting 'cognition' or 'cognitive' to see whether the meaning differs. In Chapter 19, we'll go into much more detail on what I prefer to call "computational cognition".

Kugel (2002) argues that AI will be possible using digital computers—and not requiring fancy, quantum computers or other kinds of non-digital computers—by using those digital computers in only a non–Turing-computational way. He begins his argument by observing that intelligence in general, and artificial intelligence in particular, requires "initiative", which he roughly identifies with the absence of "discipline", defined, in turn, as the ability to follow orders. (This is reminiscent of Beth Preston's views on improvisation, which we discussed in §10.4.2.) Thus, perhaps, intelligence and AI require the ability to break rules! Computation, on the other hand, requires such "discipline" (after all, as we have seen, computation certainly includes the ability to follow orders, or, at least, to behave in accordance with orders).

Moreover, Kugel argues that Turing made the same point. But did he?

Kugel quotes the following sentence:

> Intelligent behaviour presumably consists in a *departure* from the completely disciplined behaviour involved in computation, but a rather *slight* one, which does not give rise to random behaviour, or to pointless repetitive loops.
> —Turing (1950, p. 459, my italics)

However, the larger context of this passage makes it clear that Turing is thinking of a learning machine. So the "slight departure" he refers to is not so much a lack of discipline as it is the universal Turing machine's ability to change its behavior, that is, to change the software that it is running. It can't change its hardware (that is, its fetch-execute cycle). But, because the program that a universal Turing machine is executing is inscribed on the same tape that it can print on, the universal Turing machine can change that program! There is no difference between a program stored on the tape and the data also stored on the tape. (There is a difference, of course, between a *hardwired* program and data.)

This is *not* to say that computing is *not* enough for intelligence. Turing (1947) claimed that infallible entities could not be intelligent, but that fallibility allows for intelligence:

> ... fair play must be given to the machine. Instead of it sometimes giving no answer we could arrange that it gives occasional wrong answers. But the human

> mathematician would likewise make blunders when trying out new techniques. It is easy for us to regard these blunders as not counting and give him another chance, but the machine would probably be allowed no mercy. *In other words then, if a machine is expected to be infallible, it cannot also be intelligent.* There are several mathematical theorems which say almost exactly that.
> —Turing (1947, p. 394, my italics)

A few years later, Turing said something similar:

> [O]ne can show that however the machine [that is, a computer] is constructed there are bound to be cases where the machine fails to give an answer [to a mathematical question], but a mathematician would be able to. On the other hand, the machine has certain advantages over the mathematician. Whatever it does can be relied upon, assuming no mechanical 'breakdown', whereas the mathematician makes a certain proportion of mistakes. I believe that this danger of the mathematician making mistakes is an unavoidable corollary of his [sic] power of sometimes hitting upon an entirely new method. —Turing (1951, p. 256, my interpolated brackets)

Digression and Further Reading:
It's not obvious what Turing was alluding to when he said, "there are bound to be cases where the machine fails to give an answer, but a mathematician would be able to."

One possibility is that he's referring to Gödel's Incompleteness Theorem (see §6.6, footnote 6, above). If a Turing machine is programmed to prove theorems in Peano arithmetic then, by Gödel's theorem, there will be a *true* statement of arithmetic that it cannot *prove* to be a *theorem*—that is, to which it "fails to give an answer" in one sense. A human mathematician, however, could show *by other means* (but not prove as a theorem!) that the undecidable statement was *true*—that is, the human "would be able to" give an answer to the mathematical question, in a different sense.

That is, there are two ways to "give an answer": An answer can be given by "syntactically proving a theorem" or else by "semantically showing a statement to be true". For more on syntax vs. semantics, see Chapter 22.

Emil Post (1944, p. 295) had this to say about mathematics:

> The conclusion is unescapable that . . . *mathematical thinking is, and must remain, essentially creative.* To the writer's mind, this conclusion must inevitably result in at least a partial reversal of the entire axiomatic trend of the late nineteenth and early twentieth centuries, with a return to meaning and truth as being of the essence of mathematics.

For some other remarks on the mathematical abilities of humans vs. machines, see Wang 1957, p. 92; Davis 1990, 1993 (which challenge Penrose's 1989 argument that Gödel's Theorem can be viewed as a kind of hypercomputation; see the discussion in Copeland 2002b, §§1.18–1.18.1); Dennett 1995, Ch. 15 (on Penrose); and Copeland and Shagrir 2013. On Gödel, Turing, and mathematical ability, see Sieg 2007, Feferman 2011, and the excellent summary of the issues in Koellner 2018.

Sieg's essay paints the following picture suggested by some of Gödel's remarks: If individual *brains* can be considered to be Turing machines, and if individual *minds* are implemented in brains, then the "other means" that are mentioned above might be obtained from individual brains (for example, two mathematicians) working together and thus yielding a "larger" "mind". (Goodman 1984 explores this concept in the context of modal logic.) That is, each brain—considered as a formal system—can get information from other brains that it couldn't have gotten on its own. Note that this might be modelable as interactive computing or oracle computing! Dennett (1995, p. 380, my interpolated brackets) makes a similar observation when he notes that "Science . . . is not just a matter of making mistakes [as Popper suggests; see §4.9.1, above], but of making mistakes in public . . . in the hopes of getting the others to help with the corrections."

This gives support to Kugel's claims about fallibility. Such trade-offs are common: For example, as Gödel showed, certain formal arithmetic systems can be consistent (infallible?) or else complete (truthful?), but not both. An analogy is this: In the early days of cable TV (the late 1970s), there were typically two sources of information about what shows were on—*TV Guide* magazine and the local newspaper. The former was "consistent" or "infallible" in the sense that everything that it *said was* on TV was, indeed, on TV; but it was incomplete, because it did not list any cable-TV shows. The local newspaper, on the other hand, was "complete" in the sense that it included all broadcast as well as all cable-TV shows, but it was "inconsistent" or "fallible", because

it also erroneously included shows that were *not* on TV *or* cable (but there was no way of knowing which was which except by being disappointed when an advertised show was not actually on).

But the context of Turing's essays makes it clear that what Turing had in mind was the ability of *both* human mathematicians *and* computers to learn from their mistakes, so to speak, and to develop new methods for solving problems—that is, to change their "software". Turing (1947, p. 394) suggests that this might come about by "allow[ing the computer] to have contact with human beings in order that it may adapt itself to their standards", perhaps achieving such interaction through playing chess with humans.

In a later passage, Turing suggests "one feature that ... should be incorporated in the machines, and that is a 'random element' " (p. 259). This turns the computer into a kind of interactive *o*-machine that "would result in the behaviour of the machine not being by any means completely determined by the experiences to which it was subjected" (p. 259), suggesting that Turing realized that it would make it a kind of hypercomputer, but, presumably, one that would be only (small) extension of a Turing machine.

Question for the Reader:
Wouldn't the "random element" be one of "the experiences to which it was subjected"? If so, wouldn't the machine's behavior be completely determined by its experiences, even though the experiences would not be *predictable*, hence not simulatable by an ordinary Turing machine?

Digression:
It is also of interest to note that, in the same 1951 essay, Turing envisaged what has come to be known as "The Singularity":

> [It seems probable that once the machine thinking method had started, it would not take long to outstrip our feeble powers. There would be no question of the machines dying, and they would be able to converse with each other to sharpen their wits. At some stage therefore we should have to expect the machines to take control, in the way that is mentioned in Samuel Butler's *Erewhon*.
> —Turing (1951, pp. 259–260)

There is a vast literature on "The Singularity". A good place to begin is the *Wikipedia* article, https://en.wikipedia.org/wiki/Technological_singularity. For philosophical discussion, see Eden et al. 2012.

Kugel next argues that Turing computation does not suffice for intelligence, on the grounds that, if it did, it would not be able to survive! Suppose (by way of *reductio*) that Turing computation did suffice for intelligence. And suppose that a mind is a universal Turing machine with "instincts" (that is, with some built-in programs) and that it is capable of learning (that is, capable of computing new programs). To learn (that is, to compute a new program), it could *either* compute a *total* computable program (that is, one defined on all inputs) *or else* compute a *partial* computable program (that is, one that is undefined on some inputs).

Next, Kugel defines a *total machine* to be one that computes *only* total computable functions, and a *universal machine* to be one that computes *all* total computable functions and, presumably, all *partial* computable functions. Is a universal Turing machine

"total" or "universal" in Kugel's sense? According to Kugel, it can't be both: The set of total computable functions is enumerable ($f_1, f_2, \ldots$). Let P_i be a program that computes f_i, and let P be a program (machine?) that runs each P_i. Next, let P' compute $f_n + 1$. Then P' is a total computable function, but it is not among the P_i, hence it is not computed by P. That is, if P computes *only* total functions, then it can't compute *all* of them (Kugel, 2002, p. 577, note 6).

According to Kugel, a universal Turing machine is a "universal" machine (so it also computes *partial* functions). If the mind is a universal Turing machine, then there are partial functions whose values it can't compute for some inputs. And this, says Kugel would be detrimental to survival. If the mind were total, then there would be functions that it couldn't compute *at all* (namely, partial ones). This would be equally detrimental.

But, says Kugel, there is a third option: Let the mind be a universal Turing machine with "pre-computed" or "default" values for those undefined inputs. Such a machine is not a Turing machine; it is a trial-and-error machine, because it relies on intermediate outputs when it can't wait for a final result. That is, it "satisfices", because its "rationality" is "bounded", as Simon might have put it.

In other words, hypercomputation in the form of trial-and-error computation, according to Kugel, is necessary for cognition.

Further Reading:
For more on Kugel's views, see Kugel 1986a,b, 2004, 2005. Wegner and Goldin 2006a is a response to Kugel 2005.

Sloman 1996 clarifies how hypercomputation can show how some aspects of *human* cognition might not be Turing computable. (Nevertheless, the question remains whether cognition *in general* is Turing computable (or can be approximated by a Turing machine).)

Shagrir (1999, §2, pp. 132–133) observes that "The Church-Turing thesis is the claim that certain classes of computing machines, e.g., Turing-machines, compute effectively computable functions. It is not the claim that all computing machines compute solely effectively computable functions." That is, if a function is effectively computable, then it is Turing computable. But it is not the case that, if C is a computing machine that "computes" function f, then f is *effectively* computable.

This is consistent with Kugel's views on trial-and-error machines; it is also consistent with the view that what makes interaction machines more powerful than Turing machines is merely how they are coupled with either other machines or with the environment; it is not the machinery itself. Shagrir's goal is to argue for a distinction between algorithms and "computation", with the latter being a wider notion. That is, the scare-quoted term 'computes' in the last sentence of the previous paragraph doesn't refer to Turing computability, but to a more general kind of processing that can also be done on a Turing machine.

11.4.5.3 Inductive Inference

Is there a specific aspect of cognition that is not Turing computable but that *is* trial-and-error–computable? Arguably, yes: language learning.

Language learning is an example of learning a function from its values. Such learning is called "computational learning theory" or "inductive inference". Given the initial outputs of a function f—$f(1), f(2), \ldots$,—try to infer (or guess, or compute, or "compute") what function f is. This is an abstract way of describing the problem that a child faces when learning its native language: $f(t)$ is the parts of the language that the child has heard up to time t, and f is the grammar of the language.

Is learning a language computable (or hypercomputable)? Trial-and-error machines are appropriate to model this. E. Mark Gold investigated the conditions under which a *class* of languages could be said to be "learnable".

Here is Gold's Theorem (following the presentation in Johnson 2004, which also spells out many misinterpretations of the theorem by cognitive scientists): First, a "language" is defined as a subset of "all finite strings of elements from" some alphabet (Gold, 1967, pp. 448–449). Such strings can be considered to be sentences of the language. For example, a language could be "the set of *meaningful* strings of words" (that is, meaningful sentences (Gold, 1967, p. 449, my italics)), or it could be "the set of sentences that are *grammatical* in that language" (Johnson, 2004, p. 573, my italics).

Next, a *class C* of languages is "learnable" if a "learner" can learn *every* language in *C* in any "environment". But what does it mean to "learn" a language? What is a "learner"? And what is an "environment"? These turn out to be mathematical models of a human acquiring a language. So, how does a human acquire a language? One way (the usual way for a native speaker of a language—as opposed to someone learning a second language from, say, a textbook) is for the learner to hear a sequence of sentences from the language and, using a form of "inductive inference", make a "guess" as to what the language is—a guess as to what counts as a grammatical (or meaningful, or "acceptable") sentence of the language. The sentences heard are the input; the language learned is the output. The learner plays the role of a "function" that transforms the input into the output. Note, however, that as the learner hears more and more sentences, the learner's "guess" as to what the language is will change. If the learner's guesses reach a point at which there are no more changes, then the learner can be said to have learned *that* language. But note that in Gold's mathematical model, it is not a *single* language that is "learnable", but only a *class* of languages.

The mathematical model of the learner's input is called an "environment": "any infinite sequence ... of sentences ... from the target language to be learned with the requirement that every sentence of the language appears at least once in the sequence" (Johnson, 2004, p. 573). Note that this mathematical model of the input is much "stronger" than what happens with a real learner; after all, no one has ever heard *all* of the sentences of English! However, after hearing the first n sentences, the learner makes a guess, and the learner's guess after hearing the $(n+1)$st sentence may be different. The learning is modeled as a

> function that takes finite initial sequences of the environment as input, and yields as output a guess as to the target language. ... [T]he learner *learns [language] L given [environment] E* iff there is some time t_n such that at t_n and all times

> afterward, the learner correctly guesses that L is the target language present in the environment. (Gold himself called this condition 'identification in the limit'.)
> —Johnson (2004, p. 574, my interpolated brackets)

Next,

> a class C of languages has the Gold Property iff C contains (i) a countable infinity of languages L_i such that $L_i \subset L_{i+1}$ for all $i > 0$, and (ii) a further language L_∞ such that for any $i > 0$, x is a sentence of L_i only if x is a sentence of L_∞, and x is a sentence of L_∞ only if x is a sentence of L_j for some $j > 0$.
> —Johnson (2004, pp. 574–575)

In other words, L_∞ is the union of all of the other languages in C.

Finally, what Gold showed ("Gold's Theorem") was that any class of languages with the Gold Property is *unlearnable* (Johnson, 2004, p. 575). Why is this significant? Here is another version of Gold's Theorem, due to John Case:[11]

> **Corollary (Gold '67) NO** learning machine exists which can successfully learn **every** infinite, computable numerical sequence!
>
> This corollary makes computational learning theory non-trivial—there is no universally successful learning machine for the entire class of infinite, computable numerical sequences.

Case notes that Gold's Theorem applies to "**people** too if people are machines".

Further Reading:
For more on computation learning theory, (re-)read the Very Strongly Recommended website "John Case's COLT Page", http://www.eecis.udel.edu/~case/colt.html, which has a link to the entire lecture that the above "Corollary" comes from.

Gold 1965 presents the mathematics behind trial-and-error machines: "A class of problems is called decidable if there is an algorithm which will give the answer to any problem of the class after a *finite* length of time. The purpose of this paper is to discuss the classes of problems that can be solved by *infinitely* long decision procedures in the following sense: An algorithm is given which, for any problem of the class, generates an infinitely long sequence of guesses. The problem will be said to be *solved in the limit* if, after some finite point in the sequence, all the guesses are correct and the same . . . " (from the abstract, my italics).

For more information on "language learning in the limit", see Hauser et al. 2002, p. 1577, who argue for the existence of a specialized "language acquisition device" in the human brain on the grounds that Gold showed that "No known 'general learning mechanism' can acquire a natural language solely on the basis of positive or negative evidence". Whether Hauser et al. have correctly interpreted Gold is discussed in Johnson 2004, §4.

For an argument that inductive inference in particular, and machine learning in general, is *not* a method of "real learning", see Bringsjord et al. 2018.

[11] http://www.cis.udel.edu/~case/slides/colt-handout.ps

11.5 Summary

There are many kinds of sub-Turing, or "*hypo*-", computation. So, if there is any serious super-Turing, or "*hyper*-", computation, that would put classical, Turing computation somewhere in the middle. And no one disagrees that it holds a central place, given the equivalence of Turing machines to recursive functions to lambda calculation to Post-production systems, etc., and also given its modeling of human computing and its relation to Hilbert's *Entscheidungsproblem*.

*Hyper*computation seems to come in two "flavors": what I'll call "weird" hypercomputation and what I'll call "plausible" hypercomputation (to use "neutral" terms!). In the former category, I'll put "medically impossible" Zeus machines, relativistic machines that can only exist near black holes, etc. In the latter category, I'll put trial-and-error machines, interactive machines, and *o*-machines; *o*-machines are clearly a plausible extension of Turing machines, as even Turing knew.

Only the "plausible" kinds of hypercomputation seem useful. But both interaction machines and trial-and-error machines seem to be only minor extensions of the Turing analysis of computation, and their behavior is well understood and modelable by Turing's *o*-machines together with the notion of relative computability. Indeed, when you think of it (and as Feferman (1992, pp. 339–340) pointed out), all notions of computability are relative to (1) what counts as a primitive operation or basic function and (2) what count as the ways to combine them to create other operations and functions.

Two things make Turing machines (and their logical equivalents) central: The first is their power—they are provably more powerful than "*hypo*computational" models. The second is the fact that the different models of (classical) computation are logically equivalent to each other. Except for the physically "weird" hypercomputers, all other "plausible" models of hypercomputation can not only be seen as minimal (and natural) generalizations of the Turing-machine model, but all are logically equivalent to Turing's *o*-machines. And the main "problem" with those is Davis's "non-computable in"–"non-computable out" principle.

We might even suggest a generalized Computability Thesis:

> A function is "computable" iff it is computable by an *o*-machine.

Recall that Turing explicitly required that the oracle "cannot" be a Turing machine. But if we relax this constraint, then, when the oracle *is* Turing computable, this generalized thesis is just the classical one. When the oracle is *not* Turing computable, we can have non–Turing-computable—that is, "hypercomputable"—output, but only at the cost of non-computable input. However, we *can* analyze different degrees of uncomputability, as Davis, Feferman, Soare, and many others have noted.

> O-machines show us that not all that is studied in computation theory is Turing-equivalent. —Aizawa (2010, p. 230)

But note the subtle difference between saying this and saying something like: All computation is equivalent to Turing computation (which is a version of the Computability Thesis).

Fortnow (2010) nicely refutes three of the major arguments in favor of hypercomputation (including analog computation). Of most interest to us is this passage, inspired

by Turing's comment that "The real question at issue is 'What are the possible processes which can be carried out in computing a number?' " (Turing, 1936, §9, p. 249; see §8.8.2.1, above):

> Computation is about process, about the transitions made from one state of the machine to another. Computation is not about the input and the output, point A and point B, but the journey. Turing uses the computable numbers as a way to analyze the power and limitations of computation but they do not reflect computation itself. You can feed a Turing machine an infinite digits [sic] of a real number . . . , have computers interact with each other . . . , or have a computer that perform an infinite series of tasks . . . but *in all these cases the process remains the same, each step following Turing's model*. . . . So yes Virginia, the Earth is round, man has walked on the moon, Elvis is dead and *everything computable is computable by a Turing machine*. —Fortnow (2010, pp. 3, 5, my italics and interpolated brackets)

Robert Soare makes a similar observation:

> Indeed, we claim that the common conception of mechanical procedure and algorithm envisioned over this period is exactly what Turing's computor [that is, what we called the "clerk" in §8.8.2.2, footnote 4] captures. This may be viewed as roughly analogous to Euclidean geometry or Newtonian physics capturing a large part of everyday geometry or physics, but not necessarily all conceivable parts. Here, Turing has captured the notion of a function computable by a mechanical procedure, and *as yet there is no evidence for any kind of computability which is not included under this concept. If it existed, such evidence would not affect Turing's thesis about mechanical computability any more than hyperbolic geometry or Einsteinian physics refutes the laws of Euclidean geometry or Newtonian physics. Each simply describes a different part of the universe.* (Soare, 1999, pp. 9–10, my italics and interpolated brackets)

Perhaps the issue is not so much whether it is *possible* to compute the uncomputable (by extending or weakening the notion of Turing computation), but whether it is *practical* to do so. Davis (2006a, p. 126, my interpolated brackets) finds this to be ironic:

> . . . computer scientists have had to struggle wih the all-too-evident fact that from a practical point of view, Turing computability does not suffice. . . . With these [NP-complete] problems Turing computability doesn't help because, in each case, the number of steps required by the best algorithms available grows exponentially with the length of the input, making their use in practice problematical. How strange that despite this clear evidence that computability alone does not suffice for practical purposes, a movement has developed under the banner of "hypercomputation" proposing the practicality of computing the non-computable.

11.6 Questions for the Reader

1. "There are things ... bees can do that humans cannot and vice versa" (Sloman, 2002, §3.3). Does that mean that bees can do non-computable tasks? Or does 'do' mean something different from 'compute', such as physical performance? If "doing" is different from "computing", how does that affect Cleland's arguments (see §10.4.1) against the Computability Thesis?

2. If you don't allow physically impossible computations, or black-hole computations, etc., can interactive computation make the Halting Problem "computable"? Put another way, the Halting Problem is not classically computable; is it interactively "computable"?

3. The *n*-body problem is the problem of how to compute the behavior of *n* objects in space. For example, the 2-body problem concerns the relation of the Earth to the Sun (or to the Moon). The 3-body problem concerns the relation of Earth, Sun, *and* Moon. And so on. Brian Hayes (2015, esp. pp. 92–93) has suggested that one technique for simulating solutions to the *n*-body problem is to use an ordinary computer linked to a graphics processing unit that is far more powerful than the ordinary computer. Is such a combination like a Turing machine with an oracle?

4. As we will see in Chapter 19, the Turing test is interactive. If interaction is not modeled by Turing machines, how does that affect Turing's arguments about "computing machinery and intelligence"? (If you are not yet familiar with the Turing test, you might want to come back to this question after reading Chapter 19.)

5. You will probably need to study the mathematics of *o*-machines, Turing reducibility, etc., in order to give a proper answer to this question and the next, but they are worth thinking about. As I have presented it, oracles seem to play several possibly distinct roles: They can be considered to be a kind of subroutine call. They can be considered to be an input source. And they can be considered "as miraculous sources of unknowable facts" (at least, unknowable *in advance*).

   Do oracles really play all these roles? Are these roles really all distinct? And what does this conflation of roles say about my proposed "generalized Computability Thesis" in §11.5?[12]

[12] Thanks to Robin K. Hill (personal correspondence) for raising this issue and for the quoted phrase.

6. As presented in Soare 2016, p. 52, an oracle machine consists, in part, of a Turing machine together with

   > an extra 'read only' tape, called the *oracle tape*, upon which is written the characteristic function of some set A, called *oracle*, **whose symbols . . . cannot be printed over** . . . .

   Evaluate the following apparent paradox:

   (a) Interactive computing involves inputting information from, and outputting information to, the external world.

   (b) An oracle machine models interactive computing.

   (c) It is the oracle that models the external world.

   (d) Therefore, the oracle machine must be able to modify the oracle.

   (e) But, by definition, the oracle is not modifiable by the Turing machine (because it is read-only).

11.7 Further Sources of Information

1. Bringsjord, Selmer; & Zenzen, Michael (2002), "Toward a Formal Philosophy of Hypercomputation", *Minds and Machines* 12(2) (May): 259–280.
2. Burgin, Mark, & Wegner, Peter (organizers) (2003), Special Sessions on Beyond Classical Boundaries of Computability (Parts I, II, III, & IV), 2003 Spring Western Section Meeting, American Mathematical Society, http://www.ams.org/meetings/sectional/2096_progfull.html
    - Published versions of many of the papers (including papers by Bringsjord & Arkoudas, Cleland, Copeland, Kugel, and Shagrir) are online at http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_THEORETICAL_COMPUTER_SCIENCE/
3. Cooper, S. Barry (2012), "Incomputability after Alan Turing", *Notices of the AMS* 59(6): 776–784, http://www.ams.org/notices/201206/rtx120600776p.pdf
    - An interesting, illustrated, historical survey.
4. Copeland, B. Jack (guest ed.) (2002), Special Issue on Hypercomputation, *Minds and Machines* 12(4) (November).
    - Includes Bringsjord & Zenzen 2002, Copeland 2002a, Kugel 2002, Steinhart 2002.
    - Complete table of contents at http://link.springer.com/journal/11023/12/4/
5. Copeland, B. Jack (guest ed.) (2003), Special Issue on Hypercomputation (continued), *Minds and Machines* 13(1) (February).
    - Complete table of contents at: http://link.springer.com/journal/11023/13/1/
6. Steinhart, Eric (2002), "Logically Possible Machines", *Minds and Machines* 12(2) (May): 281–301.
7. Cotogno, P. (2003), "Hypercomputation and the Physical Church-Turing Thesis", *British Journal for the Philosophy of Science* 54(2): 181–224.
    - Argues that hypercomputation does not refute the Computability Thesis.
    - For a reply, see Welch, P.D. (2004), "On the Possibility, or Otherwise, of Hypercomputation", *British Journal for the Philosophy of Science* 55: 739–746.
8. Dresner, Eli (2008), "Turing-, Human- and Physical Computability: An Unasked Question", *Minds and Machines* 18(3) (Fall): 349–355.
    - This essay clarifies the the difference between Turing's mathematical model of *human* computability and a possibly more extensive notion of *physical* computability.
9. Parker, Matthew W. (2009), "Computing the Uncomputable; or, The Discrete Charm of Second-Order Simulacra", *Synthese* 169: 447–463, http://philsci-archive.pitt.edu/3905/1/Discrete_Charm_i.pdf
    - Argues that "non-computable behavior in a model ... [can be] revealed by computer simulation" (which is, of course, computable).

10. Piccinini, Gualtiero (2011), "The Physical Church-Turing Thesis: Modest or Bold?", *British Journal for the Philosophy of Science* 62: 733–769, http://www.umsl.edu/~piccininig/CT_Modest_or_Bold.pdf

    - §4 discusses hypercomputational challenges to the "modest" physical Computability Thesis, which states that "any function that is physically computable is Turing computable" (p. 734).

11. Wells, Benjamin (2004), "Hypercomputation by Definition", *Theoretical Computer Science* 317: 191–207, http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_THEORETICAL_COMPUTER_SCIENCE/05051011123316076.pdf

    - §3 contains a discussion of the relation between the Computability Thesis and the $P = NP$ problem.

Part IV

What Is a Computer Program?

Chapter 12

Algorithms, Programs, Software, and Hardware

Version of 6 March 2019; DRAFT © 2004–2019 by William J. Rapaport

> **program**: /n./ 1. A magic spell cast over a computer allowing it to turn one's input into error messages. 2. An exercise in experimental epistemology. 3. A form of art, ostensibly intended for the instruction of computers, which is nevertheless almost inevitably a failure if other programmers can't understand it.
> —*The Jargon Lexicon*, http://www.jargon.net/jargonfile/p/program.html

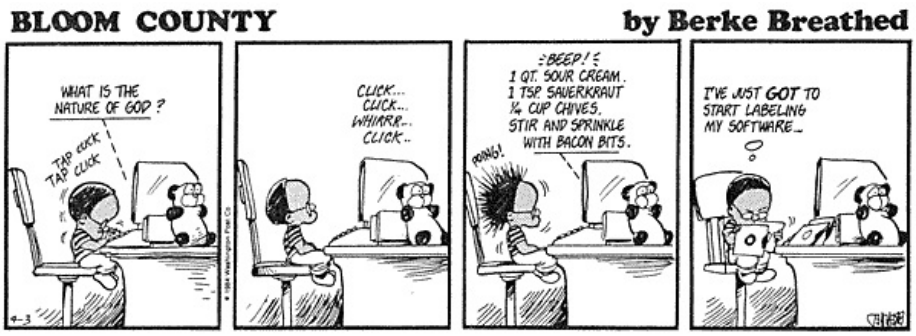


Figure 12.1: http://www.gocomics.com/bloomcounty/2010/3/15, ©1984?, Washington Post Co.

12.1 Readings:

1. Required:
   - Moor, James H. (1978), "Three Myths of Computer Science", *British Journal for the Philosophy of Science* 29(3) (September): 213–222.
     - For the purposes of Ch. 12, concentrate on §§1–2.
     - §3 is on the analog-digital distinction, which we have mentioned in earlier chapters.
     - §4 will be relevant to Ch. 15 on whether computer programs are scientific theories.
2. Very Strongly Recommended:
   - Suber, Peter (1988), "What Is Software?", *Journal of Speculative Philosophy* 2(2): 89–119.
     - Revised version at http://www.earlham.edu/~peters/writing/software.htm
3. Strongly Recommended:
   - Colburn, Timothy R. (1999), "Software, Abstraction, and Ontology", *The Monist* 82(1): 3–19.
     - Revised version appears as Colburn 2000, Ch. 12, pp. 198–209.

12.2 What Is a Computer Program?

We have explored what an *algorithm* is; we are now going to look at computer *programs*. In the course of the next few chapters, we will consider these questions:

- What *is* a computer program?
- Do computer programs "implement" algorithms?
- What is the nature of implementation?
- What are "software" and "hardware", and how are they related?
- Can computer programs be copyrighted, or should they be patented instead?
- Can (some) computer programs be considered to be scientific theories?

Typically, one tends to consider a computer program as an expression, in some language, of an algorithm. The language is typically a programming language such as Java, or Lisp, or Fortran. And a programming language is typically required to be "Turing complete", that is, to be able to express the primitive operations of a Turing machine, together with all three of the Böhm-Jacopini "grammar" rules: sequence, selection, and while-loops, as discussed in §7.5.3. (So, "*computer* languages", such as HTML, that lack one or more of these "control structures" are not "*programming* languages".)

The algorithm is something more "abstract", whereas the program that expresses it (or "implements" it in language) is something more "concrete". A program is more concrete than an algorithm in two ways: First, a program is a physical object, either written on paper or "hardwired" in a computer. Perhaps the relationship between an algorithm and a program is something like the relationship between a number and a numeral: Just as the number "two" can be expressed with many different numerals (such as '2' or 'II') and many different words (such as 'two', 'deux', or 'zwei'), so a single algorithm, such as the algorithm for binary search, can be expressed in many different programming languages.

In fact, we can't really talk about numbers or algorithms without using some kind of language, so maybe there really aren't any of these abstract things called 'algorithms' (or numbers!), just words for them. This is an ontological view in philosophy called 'nominalism' (https://plato.stanford.edu/search/searcher.py?query=nominalism). Maybe the only things that exist are programs, some of which might be written in programming languages that can be directly used to cause a computer to execute the program (or execute the algorithm?), and some of which might be written in a natural language, such as English. (The nominalist can still talk about "algorithms", understanding them as computer programs. Ask yourself whether a nominalist can still talk about numbers, understood as numerals: After all, there are infinitely many numbers, but only finitely many numerals.)

The second way that a program is more concrete than an algorithm is that a program is more detailed. Where an algorithm might simply specify how to perform a

Quicksort (https://en.wikipedia.org/wiki/Quicksort), a Quicksort program for a particular computer would have to spell out the details of how that sort would be physcially implemented in that computer. (We'll have more to say about this when we discuss implementation in Chapter 14.)

In the early days of computers, programs were not typically expressed in programming languages; rather they were "hardwired" into the computer (perhaps certain physical switches were set in certain ways). These programs were physical parts of the computer's hardware, not texts. The program could be changed by re-wiring the computer (perhaps by re-setting the switches). Yet computer programs are typically considered to be "software", not "hardware", so was such wiring (or switch-setting) a computer program?

And what about a program written on a piece of paper? Does it differ from the very same program written on a computer file? The former just sits there doing nothing. So does the latter, but the latter can be used as input to other programs on the computer that will use the information in the program to "set the switches" so that the computer can execute the program. But is the medium on which the program is written the only difference between these two programs?

Further Reading:
Gemignani 1981 is a good survey of the issues that arise when trying to say what a computer program is.

Haigh and Priestley 2016 is an interesting history of both programming and the term 'program', arguing that Ada Lovelace was probably *not* the first computer programmer and that programming computers originally had a lot in common with concert "programming" or radio "programs".

12.3 What Is a Program and Its Relation to Algorithms?

In §7.3.1, we saw that a *function* defined *extensionally* as a *set* of input-output pairs satisfying the same-input/same-output constraint could be "implemented"—made more precise or more explicit—by many different functions defined *intensionally* by a *rule*, each of which is a *description of the relationship* between the input and the output. Thus, for example, the function $f = \{(0,0),(1,3),(2,6),(3,9),\ldots\}$ can be implemented by the rule $f_1(x) = 3x$ or by the rule $f_2(x) = x+x+x$, etc.

We also saw that a *rule* could be implemented by many different *algorithms*, each of which spells out the intermediate steps that *compute* the output according to the rule. Thus, for example, the rule f_1 could be computed by either of the following algorithms:

Algorithm $A^1_{f_1}(x)$
begin
  $f_1 := 3$;
  $f_1 := f_1 * x$
end.

Algorithm $A^2_{f_1}(x)$
begin
  $f_1 := x$;
  $f_1 := 3 * f_1$
end.

And the rule f_2 could be computed by either of these algorithms:

Algorithm $A^1_{f_2}(x)$	**Algorithm** $A^2_{f_2}(x)$
begin	**begin**
$f_2 := x+x$;	$f_2 := x$;
$f_2 := f_2+x$	$f_2 := f_2+x$;
end.	$f_2 := f_2+x$
	end.

And so on. Thus, our original function f could be computed by any one of those four algorithms, among infinitely many others.

One way to consider the relationship between algorithms and programs is to continue this chain of implementations: An algorithm can be implemented by a computer program written in a high-level computer-programming language. That program can then be implemented in assembly language (which is computer-specific, and provides more detail). The assembly-language program, in turn, can be implemented in machine language. And, finally, the machine-language program can be implemented in hardware by "hardwiring" a computer—or, in more modern terminology, by using a chip designed to perform that function. Arguably, the static, hardwired program is implemented by the dynamic *process* that is created when the computer executes "the" program. (A nice description for readers who are not computer scientists can be found in Colburn 1999, pp. 6–9.)

Both algorithms and programs are normally considered to be "software", and physical implementations of them in a computer are normally considered to be "hardware". But what exactly is software, and how can it be distinguished from hardware? Many authors use 'program' and 'software' as synonyms. But if we view a program as an implementation of an algorithm (in some medium such as language or the switch settings of a computer), and if we view software as contrasted with hardware, it's not obvious that programs and software are exactly the same thing. Programs can be expressed on paper in a programming language, which seems like software. But they can also be hardwired in a physical computer, which seems like hardware. And software is not usually defined in terms of algorithms.

Amnon H. Eden (2005) offers three ways to think about what a computer program is. The first is as what C.A.R. Hoare (1986, p. 115) called "mathematical expressions". This suggests that (at least some kinds of) software are abstract, mathematical entities (or linguistic expressions thereof, perhaps more along the lines of numerals than of numbers). An example of the algorithm-program relationship that is along the lines of the number-numeral relationship is offered by Stewart Shapiro: When trying to study a dynamic *process* mathematically,

> It is common practice in mathematics ... [to associate] a class of (static) mathematical objects with the process in question and then 'transferring' results about the mathematical objects to the processes. In mathematical logic, for example, a process of deductive reasoning is studied through the use of formal deductions; in mathematical physics, the movement of particles is studied by means of an associated class of curves in geometrical space .... The technique employed in

> the theory of computability is to associate an algorithm with a written set of instructions for computation, usually in a formal language, and to study the class of algorithmic descriptions syntactically. —Stewart Shapiro (1983, p. 204)

But Eden's view is a bit different: His idea is that programs themselves are mathematical objects capable of being studied mathematically. We will return to this theme in Chapter 16.

Eden's second way to think about programs is as a "natural kind". (We discussed natural kinds in §3.3.3.) Computer programs exist, so we should study them as we find them in nature, so to speak. We can try to categorize them, identify their properties, determine their relative merits, and so on.

The third way is "as an engineered artefact".[1] Whereas the study of programs as natural kinds is more or less descriptive, the study of them as engineered artifacts is "normative": We should try to specify ways in which programs can be designed so as to be more efficient, more reliable, etc. (We'll return to this, too, in Chapter 16.)

Finally, Eden offers a fourth way: "as a cognitive artefact: Software is conceived and designed at a level of abstraction higher than the programming language". But it is not clear what he means by that: Does it mean that software is more like an algorithm than a program? Or perhaps it is something even more abstract, like a *specification* for an algorithm (as we discussed in connection with Beth Preston's views, in §10.4.2). We'll come back to this idea in §12.4.2.

Further Reading:
For more on Eden's views, see Eden 2007; Eden and Turner 2007; Turner and Eden 2007.

On the idea that some kinds of software, in particular, computer programs (and the activity of programming), are mathematical in nature, see Scherlis and Scott 1983 and our earlier discussion of CS as mathematics in §3.9.1.

12.4 What Is Software and Its Relation to Programs and to Hardware?

12.4.1 Etymology of 'Software'

The earliest use of the word 'software' in its modern sense has been traced back to the mathematician John W. Tukey (1958, p. 2):

> Today the "software" comprising the carefully planned interpretive routines, compilers, and other aspects of automative [sic] programming are at least as important to the modern electronic calculator as its "hardware" of tubes, transistors, wires, tapes and the like.

But the word is older than Tukey's use of it: The earliest cited use (in 1782, according to the *OED*) is for textiles and fabrics—literally "soft wares". A later use, dating to 1850, equated it with "vegetable and animal matters—everything that will decompose"

[1] That's the British spelling of 'artifact'.

in the realm of "rubbish-tip pickers" (Fred R. Shapiro 2000, p. 69).[2] And two years before Tukey's paper, Richard B. Carhart (1956, p. 149) equated software with the *people* who operate a computer system (where the computer system was identified as the hardware); programs (or other modern notions of software) were not mentioned.

Further Reading:
For more information on the history of the word, see http://www.oed.com/view/Entry/183938; Fred R. Shapiro 2000; Mahoney 2011, p. 238, note 20; and http://www.historyofinformation.com/expanded.php?id=936.

Mahoney 2011, Ch. 13 ("Extracts from *Software as Science—Science as Software*"), discusses the history of software; the section titled "Extract 3: Software as Science" is especially interesting.

12.4.2 Software and Music

> Is Bach's written score to the *Art of the Fugue*, perhaps with a human interpreter thrown in, the software of an organ? —Suber (1988, p. 90)

Tukey's use of the term strongly suggests that the things that count as software are more abstract than the things that count as hardware. Using a concept very similar to Eden's "cognitive artifact", Nurbay Irmak (2012) argues that software is an "abstract artifact", where an "artifact . . . is an intentional product of human activity" (pp. 55–56) and an "abstract" object is one that is "non-spatial" but that "may or may not have some temporal properties" (p. 56). Irmak likens software to another abstract (or cognitive?) artifact: musical works (§2, pp. 65ff). There are close similarities. For instance, a Turing machine (or any hardwired computer that can perform only one task) is like a music box that can play only one tune, whereas a player piano is like a universal Turing machine, capable of playing any tune encoded on its "piano roll".

One difference between software and music that Irmak points out concerns "a change or a revision on a musical work once composed" (p. 67). This raises some interesting questions: How should musical adaptations or jazzy versions of a piece of music be characterized? What about different player's interpretations? One pianist's version of, say, Bach's *Goldberg Variations* will sound very different from another's, yet, presumably, they are using the same "software". Are there analogies to these with respect to computer software? And we looked at the relationships between software and improvisational music in §10.4.2. (In addition, problems about small changes in software seem analogous to issues of personal identity through time.)

Irmak (2012, p. 68, my italics) says, "the idea that software and musical works are *created* is . . . central to our beliefs". Perhaps, but here there are similarities with issues in the philosophy of mathematics: Are theorems or proofs similarly "created"? "Mathematical Platonism" is a view in the philosophy of mathematics (championed by Gödel) that mathematical entities are mind-independent (Linnebo, 2018). So, should we take a mathematically Platonic attitude towards all of these kinds of things, and

[2] Insofar as decomposition is a form of changeability, this is, as we will see in §12.4.4.1.3, below, consistent with Moor's definition of 'software'!

revise our ordinary view that computer software and musical works are *created* in favor of a view that they are *discovered*? (The philosopher John Stuart Mill is alleged to have been depressed when he learned that there were only a finite number of possible combinations of notes, hence only a finite number of possible musical compositions (https://philosophynow.org/issues/55/Birthday_Special_John_Stuart_Mill.)

Digression and Further Reading:
There are other interesting relationships between software and art. Bond 2005, p. 123, says: "A programming language is a language after all, albeit a hightly constrained one. As such, it is a perfect medium for the poet comfortable with other highly constrained poetic forms like the sonnet or haiku". On the literary value of programs, see especially Knuth 1984. See also Schneider 2007; Chandra 2014, reviewed in Gleick 2014; and Zeke Turner 2015 (on the first art auction of computer code). For some examples of "aesthetic" or "playful" programs, see (1) "Most Adaptable Programs" (1991, 1996), http://www2.latech.edu/~acm/helloworld/multilang.html (two examples of programs that can be compiled and run in several different programming-language systems, as if there were a sentence that was grammatical and meaningful in both English and French—is there such a sentence?) and (2) "dodsond1", a text-to-"Pig Latin" translator (12th International Obfuscated C Code Contest, 1995, http://www.ioccc.org/1995/dodsond1.c)—which must be seen to be believed! (For more examples, see http://www.ioccc.org/years-spoiler.html.)

12.4.3 The Dual Nature of Programs

Our first main issue concerns the dual nature of programs: They can be considered to be both text and machine (or mechanism), both software and hardware. To clarify this dual nature, consider this problem:

> ... Bruce Schneier authored a book entitled *Applied Cryptography*, which discusses many commonly used ciphers and included source code for a number of algorithms. The State Department decided that the book was freely exportable because it had been openly published but refused permission for export of a floppy disk containing the same source code printed in the book. The book's appendices on disk are apparently munitions legally indistinguishable from a cluster bomb or laser-guided missile. ... [The] disk cannot legally leave the country, even though the original book has long since passed overseas and all the code in it is available on the Internet. —Wallich (1997, p. 42)[3]

How can a program written on paper be considered a different thing from the very same program "written" on a floppy disk? What if the paper that the program was written on was Hayes's "magic paper" (discussed in §9.6)? But isn't that similar to what a floppy disk is, at least, when it is being "read" by a computer?

Is the machine-table program of a Turing machine software, or is it hardware? It certainly seems to be hardwired. If you think that it is a kind of category mistake to talk about whether an abstract, mathematical entity such as a Turing machine can have software or hardware, then consider this: Suppose you have a physical implementation

[3]The case is discussed at length in Colburn 1999, 2000.

of a Turing machine: a hardwired, single-purpose, physical computer that (let's say) does nothing but accept two integers as input and produces their sum as output. Is the program that runs this adder software or hardware? Because the machine table of such a (physical implementation of a) Turing machine is not written down anywhere, but is part of the (physical) mechanism of the machine, it certainly seems to be more like hardware than software.

In a *universal* Turing machine, is *its* machine table (that is, its fetch-execute cycle) software, or hardware? And what about the program that is stored on its tape? By the logic of the previous paragraph, its fetch-execute machine table would be hardware, and its stored program would be software. Again, if you prefer to limit the discussion to physical computers, then consider a smartphone, one of whose apps is a calculator that can add two integers. Not only can the calculator do other mathematical operations, the smartphone itself can do many other things (play music, take pictures, make phone calls, etc.), *and* it can download new apps that will allow it to do many other things. So it can be considered to be a physical implementation of a universal Turing machine. By our previous reasoning, the program that is the smartphone's adder (calculator) is software, and the program that allows the smartphone to do all of the above is hardware.

12.4.4 Three Theories of Software

In this section, we will look at what three philosophers have had to say about software: James H. Moor (1978) argues that software is *changeable*. Peter Suber (1988) argues that it is *pure syntax*. And Timothy Colburn (1999) argues that it is a *concrete abstraction*. Keep in mind that they may be assuming that software and computer programs are the same things.

12.4.4.1 Moor's Theory of the Nature of Software

12.4.4.1.1 Levels of Understanding. For very many phenomena, a single entity can be viewed from multiple perspectives (sometimes called "levels" or "stances"). According to Moor (1978, p. 213), both computers and computer programs "can be understood on two levels": They can be understood as physical objects, subject to the laws of physics, electronics, and so on. A computer disk containing a program would be a clear example of this level. But they can also be understood on a symbolic level: A computer can be considered as a calculating device, and a computer program can be considered as a set of instructions. The text of the computer program that is engraved on the disk would be a clear example of this level.

Moor's two levels—the physical and the symbolic—are close to what Daniel C. Dennett (1971) calls the physical and design "stances". Dennett suggested that a chess-playing computer or its computer program can be understood in *three* different ways:

- From the *physical stance*, its behavior can be predicted or explained on the basis of its physical construction together with physical laws. Thus, we might say that it made (or failed to make) a certain move because logic gates #5, #7, and #8 were open, or because transistor #41 was defective.

- From the *design stance*, its behavior can be predicted or explained on the basis of information or assumptions about how it was designed or how it is expected to behave, assuming that it *was* designed to behave that way and isn't malfunctioning. Thus, we might say that it made (or failed to make) a certain move because line #73 of its program has an if-then-else statement with an infinite loop.

- From the *intentional stance*, its behavior can be predicted or explained on the basis of the language of "folk psychology": ordinary people's informal (and not necessarily scientific) theories of why people behave the way they do, expressed in the language of beliefs, desires, and intentions. For instance, I might explain your behavior by saying that (a) you desired a piece of chocolate, (b) you believed that someone would give you chocolate if you asked them for it, so (c) you formed the intention of asking me for some chocolate. Similarly, we might say that the chess-playing computer made a certain move because (a) it desired to put my king in check, (b) it believed that moving its knight to a certain square would put my king in check, and so (c) it formed the intention of moving its knight to that position.

Each of these "stances" has different advantages for dealing with the chess-playing computer: If the computer is physically broken, then the physical stance can help us repair it. If the computer is playing poorly, then the design stance can help us debug its program. If I am playing chess against the computer, then the intentional stance can help me figure out a way to beat it.

Further Reading:
For more on the intentional stance, see Dennett 1987, 2009b, and Dennett 2013a, Ch. 18.

Miller 2004 contains some interesting follow-ups from the perspective of CS to Dennett's theory of different "stances".

If you are uneasy about the intentional stance's use of psychological terms to describe computers, you might treat the terms 'desired', 'believed', and 'formed the intention' as metaphorical (Lakoff and Johnson, 1980a). We'll return to this idea in Chapter 19, but see also Dennett 2013a, Chs. 18, 21, and the discussion of "levels of description" in Newell 1980, §6. One major difference between Newell and Dennett is that the former holds that "these levels of description do not exist just in the eye of the beholder, but have a reality in … the real world" (Newell, 1980, p. 173), whereas Dennett says this:

> Intentional system theory deals just with the performance specifiations of believers while remaining silent on how the systems are to be implemented. —Dennett (1981, p. 52)

(See also Dennett 1981, p. 59, where he talks about "realizations or embodiments of a Turing machine".) We'll return to implementation in Ch 14. Figdor 2017 argues that (some of) these uses are *literally* true.

12.4.4.1.2 Moor's Definition of 'Program' Moor offers a definition of 'computer program' that is intended to be neutral with respect to the different stances of the software-hardware duality:

> a computer program is a set of instructions which a computer can follow (or at least there is an acknowledged effective procedure for putting them into a form which the computer can follow) to perform an activity. —Moor (1978, p. 214)

Let's make this a bit more explicit, in order to highlight its principal features:

Definition M1:

Let *C* be a computer.
Let *P* be a set of instructions.
Then *P is a computer program for C* $=_{def}$

1. there is an effective procedure for putting *P* in a form ...
2. ... that *C* can "follow" ...
3. ... in order to perform an activity.

We could be even more explicit:

Definition M2:

Let *C* be a computer.
Let *P* be a set of instructions.
Let *A* be an activity.
Then *P is a computer program for C to do A* $=_{def}$

1. there is an effective procedure for putting *P* in a form ...
2. ... that *C* can "follow" ...
3. ... in order to perform *A*.

Definition M2 makes the "activity" *A* a bit more perspicuous. In Chapter 7, we briefly looked at the role of an algorithm's *purpose*, and we will examine it in more detail in Chapter 17. But, for now, it will be easier to focus on Definition 1.

On that definition, *being a computer program* is not simply a *property* of some set of instructions. Rather, it is a binary *relation* between a set of instructions and a computer. (On Definition M2, it is a *ternary* relation among a set of instructions, a computer, and an activity. But here I just want to focus on the role of the computer, which is why we're just going to consider Definition M1.) As a binary relation, a set of instructions that is a computer program for one computer might not be a computer program for a different computer, perhaps because the second one lacks an effective procedure for knowing how to follow it: One computer's program might be another's noise. For instance, the Microsoft Word program that is written for an iMac computer running MacOSX differs from the Microsoft Word program that is written for a PC running Windows, because the underlying computers use different operating systems and different machine languages. This would be so even if the two programs' "look and feel" (that is, what the user sees on the screen and how the user interacts with the program) were identical.

There are some questions we can ask about Moor's definition:

12.4.4.1.2.1 Instructions. What are the instructions? Presumably, they must be algorithmic, though Moor does not explicitly say so. Is the set of instructions physical, that is, hardwired? Or are the instructions written in some language? Could they be drawn, instead—perhaps as a flowchart? Could they be spoken? Here, Moor's answer seems to be that it doesn't matter, as long as there is a way for the computer to "interpret" or "understand" the instructions and thus carry them out. (In §12.4.5, we will see that Suber makes a similar point.) Importantly, the "way" that the computer "interprets" the instructions must itself be a computable function ("an effective procedure for putting them into a form which the computer can follow"). Otherwise, it might require some kind of "built-in", *non*-computable method of "understanding" what it is doing.

Terminological Digression:
When I say that the computer has to "interpret" the instructions, I simply mean that the computer has, somehow, to be able to convert the symbols that are part of the program into actions that it performs on its "switches". This is different from the distinction in CS between "interpreted" and "compiled" programs. A "compiled" program is translated into the computer's machine language all at once, and then the computer executes the machine-language version of the program, in much the same way that an entire book might be translated from one language to another. By contrast, an "interpreted" program is translated step by step into the computer's machine language, and the computer executes each step before translating the next one, in much the same way that a UN "simultaneous translator" translates a speech sentence by sentence while the speaker is giving it. In both cases, the computer is "interpreting"—understanding—the instructions in the sense in which I used that word in the previous paragraph. That sense of 'interpret' is closer to the one in the cartoon in Fig. 12.2.

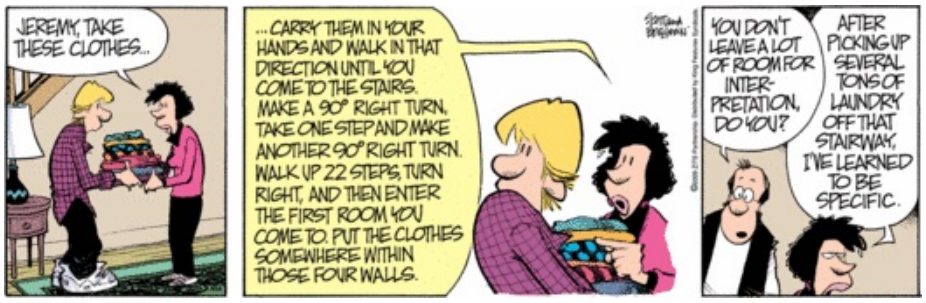


Figure 12.2: http://zitscomics.com/comics/march-14-2009/, ©2009, Zits Partnership

12.4.4.1.2.2 "Following Instructions". Moor says that computers "follow instructions". And we saw in §9.9 that Stuart C. Shapiro defined a computer as "a general-purpose procedure-*following* machine (Shapiro, 2001, p. 2, my italics). But *does* a computer "follow" instructions? Or does it merely *behave in accordance with* them?

> It is common to differentiate between satisfying a rule and following a rule (cf. Searle (1980); Wittgenstein (1958, §§185–242)). To satisfy a rule is simply to behave in such a way that fits the description of the rule—merely to conform behaviour to the rule. It is in this sense that the motion of the planets satisfy [sic] the rules embodied by classical physics. On the other hand, following a rule implies a causal link between the rule and some behaviour, and moreover that the rule is an intentional object. ... [M]erely satisfying a rule is not sufficient for following the rule. —Chow (2015, p. 1000)

Compare the human use of natural language: When we speak or write, do we "follow" the rules of grammar, in the sense of consulting them (even if unconsciously) before generating our speech or writing? Or does it make more sense to say that the rules of grammar merely *describe* our linguistic behavior? We probably do both, though the former predominates:

> We learn from psycholinguistics that ... [understanding language] involves subconscious, subpersonal, automatic, extraordinarily fast processing [that is, what we referred to as "System 1" in §3.6.1], *and that is mostly all that it involves* .... Where understanding is difficult—for example, with multiple center embedding ["A mouse that a cat that a dog chased caught ate cheese"]—it *may* be helped by "central processor," relatively slow reasoning, leading to a conscious judgment about ... [an] utterance. But such high-level processes are a very small part of language understanding.
> —Devitt and Porot (2018, p. 9, italics in original, my bracketed interpolations)

Note, however, that a computer programmed to understand and generate natural language might, in fact, speak or write by explicitly following rules of grammar that are encoded in its suite of natural-language-processing programs (Shapiro, 1989; Shapiro and Rapaport, 1991, 1995; Jurafsky and Martin, 2000). We have also seen a similar question when we considered whether the solar system "follows" Kepler's laws of planetary motion or whether the planets' movements are merely best *described* by Kepler's laws (§9.8.2). As we have seen, Turing machines—as models of hardwired, single-purpose computers—merely behave in accordance with their machine table. They don't "consult" those "instructions" and then "follow" them. On the other hand, *universal* Turing machines—as models of programmable, general-purpose computers—*can* be said to "follow" instructions. Behaving in accordance with their fetch-execute machine table, they *do* "consult" the instructions stored on their tape, and *follow* them. It might be useful to have a neutral term for what it is that a computer does when it is doing its processing, whether it is a Turing machine or a universal Turing machine; perhaps 'execute' could be used for this purpose. So, we could say that a Turing machine *executes* the instructions in its machine table, but doesn't *follow* them, whereas

a universal Turing machine *executes* the fetch-execute cycle and thereby *follows* the program encoded on its tape.

Question for the Reader:
Suppose that a universal Turing machine (or your Mac or PC) is running a program that adds two integers. What is it doing? Is it adding two integers (that is, is it executing the addition program stored on its tape)? Or is it carrying out a fetch-execute cycle? Or is it doing both? Or is it doing one *by* doing the other? And what exactly does it mean to do one thing "by" doing another?

Searle (1969, §2.5) identifies a related distinction concerning the instructions or rules themselves: Roughly, *constitutive* rules determine or define the behavior of some system, whereas *regulative* rules "regulate antecedently or independently existing forms of behavior" (p. 33). For example, the rules of grammar that linguists discover about the natural languages that we speak are *constitutive* rules; they are descriptive of the "innate" rules that we "execute" or use unconsciously, such as "Declarative sentences of English consist of a noun phrase followed by a verb phrase." The explicit rules of grammar that we have to learn in school (or that "grammar Nazis" insist that we "follow") are *regulative* rules; they recommend (or insist upon) a way to do things—for example, "Prepositions should not be used to end sentences with." In the theory of computation, the program for a Turing machine is a constitutive rule. Because the program for a *universal* Turing machine is its fetch-execute cycle, that program is a constitutive rule; but the program (the software) inscribed on its tape that a universal Turing machine is "following" is a regulative rule.

Further Reading:
Dennett (1995, pp. 372–373) discusses "two profoundly different ways of building dams: the way beavers do and the way we do." Recast in the present terminology, beavers merely behave in accordance with a genetic algorithm, whereas humans follow explicit engineering procedures.

12.4.4.1.3 Moor's Definitions of Software and Hardware. Next, Moor distinguishes between software and hardware. The informal and traditional distinction is that a computer *program* is "software" and a *computer* is "hardware". But this raises the problem of whether the "wiring" in a hardwired computer is hardware (because it involves physical wires) or software (because it is the computer's program). And, of course, it gives rise to the problem mentioned by Wallich, cited in §12.4.3. So, Moor suggests a better set of definitions:

> For a given person and computer system the software will be those programs which can be run on the computer system and which contain instructions the person can change, and the hardware will be that part of the computer system which is not software. —Moor (1978, p. 215)

In line with this, Frank Vahid (2003, p. 27, original italics, my boldface) notes that, in the early days of computing, "the frequently **changing** programs, or *software*, became distinguished from the **unchanging** hardware on which they ran."

Again, let's make this a bit more explicit, in order to highlight its features:

Definition M3:

Let *C* be a computer.
Let *P* be a person (perhaps *C*'s programmer).[4]
Let *S* be some entity (possibly a part of *C*).
Then *S is software for C and P* $=_{def}$

1. *S* is a computer program for *C*,[5] and
2. *S* is changeable by *P*.

and *H is hardware for C and P* $=_{def}$

1. *H* is (a physical) part of *C*, and
2. *H* is *not* software for *C* and *P*.

Note that *being software* is a *ternary* relation among three entities: a computer program, a person, and a computer.[6] It is not a simple property such that something either is, or else it isn't, software. In other words, software is in the eye of the beholder: One person's or one computer's software might be another's hardware!

And a physical part of a computer will be hardware for a person and that computer if *either* it is *not* a computer program *for* that computer *or* it is not *changeable* by that person.

These definitions seem to allow for the following two possibilities: First, suppose that there is a computer program *J* written in Java that runs on my computer. Even if *J* is changeable by the programmer who wrote it or the lab technician who operates the computer—and therefore *software* for that person—it will be *hardware* for me if I don't know Java or don't have access to the program so that I could change it.

And, second, if a programmer can "rewire" a computer (or directly set its "switches"), then that computer's program is software, even if it is a physical part of the computer: Software can be hardware!

Later writers have made similar observations: Vahid (2003, pp. 27, 31, 32) suggests that "the processors, memories, and buses—what we previously considered a system's unchangeable hardware—can actually be quite soft" (p. 32). What he seems to mean by this is that *embedded systems*—"hidden computing systems [that] drive the electronic products around us" (p. 27)—can be swapped for others in the larger systems that they are components of, thus becoming "changeable" in much the way that software is normally considered to be. But this seems to just be the same as the old rewiring of the early days of programming (except that, instead of changing the wires or switches, it is entire, but miniaturized, computers that are changed).

And Piccinini (2008, §§3.1.1–3.1.2, p. 40) distinguishes between "hard programmability" and "soft programmability": The former refers to "the mechanical modification

[4] Please note that, here, '*P*' refers to a *person*, not a *program* as in definitions M1 and M2.

[5] This would have to be modified to include activity *A* if we want to use Definition M2 for 'computer program'.

[6] Or it might be a *quaternary* relation among *four* things, if we include activity *A*.

of ... [a computer's] functional organization"; the latter refers to "modification involv[ing] the supply of appropriately arranged digits (instructions)" "without manually rewiring any of the components". This can be viewed either as a further breakdown of software into these two kinds, or else as limiting the changeability to "soft" changeability.

On Moor's definition, the machine table of a Turing machine—even though we think of it as the Turing machine's "program"—is part of its hardware, because it is not changeable: Were it to be changed (somehow), we would have a *different* Turing machine; its machine table is an "essential" property of the Turing machine (in the sense that we discussed in §§2.8 and 9.5.4). It is probably best to think of a Turing machine's machine table, not as a program written in a Turing-machine programming language such as we used in Chapter 8, but as the way that the "gears" of the Turing machine are arranged so that it behaves the way that it does.

This is what Samuel (1953, pp. 1226–1227) called "fixed programming":

> By fixed programming we mean the kind of programming which controls your automatic dishwasher for example. Here the sequence of operations is fixed and built into the wiring of the control or sequencing unit. Once started, the dishwasher will proceed through a regular series of operations, washing, rinsing and drying. Of course, if one wished, one could change the wiring to alter the program.

Note, however, that modern dishwashers allow for some "programming" by pushbuttons that can alter its operations. But perhaps this is more like interactive computing, as we discussed in §11.4.3.

For the universal Turing machine's software (the program stored on its tape), Samuel has this analogy:

> Suppose you wished to give your assistant a large number of instructions for manual computations all in advance. You could do this by supplying him with a prepared set of instructions, or you could dictate the instructions and have him write them down, perhaps at the top of the same sheet of paper on which he is later to perform the computations. Two different situations are here involved, although at first glance the distinction appears trivial. In the first case the instructions are stored on a separate instruction form, while in the second case they are stored by the same medium which is used for data. Both situations are found to exist in computing machines. The first case is exemplified by certain machines which use special program tapes. The second situation is becoming quite common in the newer machines and is the case usually meant when the term "stored program" is used. ...
>
> We can now go back and consider one property of the "stored program" method of operation which is rather unique and which really must be understood to appreciate the full value of such a concept. This property is that of being able to operate on the instructions themselves just as if they were ordinary data. This means that the entire course of a computation can be altered, including the operations themselves, the choice of data on which the operations are to be performed, and the location at which the results are to be stored, and this can all be done on the basis of results obtained during the course of the calculations through the use of conditional transfer instructions. —Samuel (1953, pp. 1227–1228)

Shades of Hayes's "magic paper"! (And for a humorous comment on this, see Figure 12.3.)

Figure 12.3: http://dilbert.com/strip/2011-08-04, ©2011, Scott Adams Inc.

12.4.5 Suber's Theory of the Nature of Software

Peter Suber (1988, p. 94) says that he will "use 'program' and 'software' interchangeably". This is unfortunate, because it seems to beg the question about whether all programs are software. In what follows, we will ignore this (up to a point, as you will see), and simply try to understand what he means by 'software'. His definition is straightforward and rather different from Moor's: "software is pattern *per se*, or syntactical form" (Suber, 1988, online abstract). What does he mean by this, and why does he think that it is true? Here is his argument:

1. "Software patterns ... are essentially expressed as arrays of symbols—or texts". (Suber, 1988, §2 ("Digital and Analog Patterns"), p. 91)

   > That is, all software is a text. He notes that this is a generalization of viewing them as "expressed in binary codes", and he calls such texts "digital patterns". If we think of software as a computer program written in a programming language, or even expressed as arrays of '0's and '1's, this is plausible. But what about hardwired programs? He might say that they are not software. But he might also say that, because there is no significant difference between an array of '0's and '1's and an array of switches in one of two positions, even such a hardwired program is a text. (But see premise 7, below.)

2. "The important feature of digital patterns here is ... their formal articulation of parts ...." (Suber, 1988, §2, p. 91)

   > By 'formal articulation', I will assume that he means "syntax". So, all texts have a syntax. But do they? An array of '0's and '1's that corresponds to the binary expression of the decimal part of a real number arguably has a syntax. But what about a random array? Of course, if the syntax of an array is just the properties and relations of its elements, then even a random array has a syntax.

3. "Each joint of articulation carries information for any machine designed to read it." (Suber, 1988, §2, p. 91)

   So, the syntax of digital patterns can convey information for appropriate readers. This is close to clause 2 of Moor's Definition M1: Computers have to be able to "follow" their programs, and, to do that, a computer must be able to *read* its program.

4. The Noiseless Principle: "some order may be made of any set of data points; every formal expression has at least one interpretation. ... [N]o pattern is noise to all possible machines and languages."
   (Suber, 1988, §3 ("First Formulation"), p. 94)

   This seems to be equivalent to premise 3: If "every joint of articulation carries information", then "no pattern is noise", and vice versa.

5. ∴ The executability of software is a function of its syntax, the language that it is written in, and of the machine that runs it.
   (Suber, 1988, §4 ("Executability"), especially p. 97)

   This follows from the previous premises: All software is a text; each text has a(t least one) syntax; each syntax has a(t least one) interpretation. A machine designed to "understand" that syntax can execute that software.

   Suber hypothesizes that all software must be readable and executable. Here, he is arguing that any text is executable given an appropriate syntax for it and the right language and machine to interpret that syntax.

6. Software is readable if and only if (1) it has a "physical representation ... that suits the machine that is to read it" and (2) it is "in 'machine language'."
   (Suber, 1988, §5 ("Readability"), p. 98)

   This seems to come down to the same thing as saying that there must be a machine that is capable of reading it. Just as I can't read something written in invisible ink and in Mandarin (because I can neither see it nor parse it even if I *could* see it), so the machine has to be able to "see" the text, and it has to be able to understand it. Perhaps this is best taken as a definition of 'readable'. In any case, it does not seem to add anything over and above the previous conclusion.

7. The Sensible Principle: "any pattern can be physically embodied".
   (Suber, 1988, §5, p. 100)

   So, even if hardwired programs are not "texts" (as we wondered in premise 1), they are "physical embodiments" of texts.

8. The Digital Principle: Any "pattern"—that is, any text, including an analog pattern—"can be reproduced by a digital pattern to an arbitrary degree of accuracy". (Suber, 1988, §6 ("Pattern *Per Se* Again"), p. 91)

9. ∴ Any text is readable. (Follows from the Sensible and the Digital Principles.)

10. ∴ Any text is executable. (Follows from line 5 and premise 6, clause (2).)

    > But you should ask yourself how a text that does not contain any instructions (more precisely, a text that is not an algorithm)[7] might be executable.

11. ∴ Any text is software.

    > There are two things to note here. First, recall premise 1: All software is text. Suber seems to have argued from that premise to its converse. So "software" and "text" are the same thing. Second, you might ask yourself how this relates to Searle's claim that everything is a computer!

Suber also notes that "software is *portable*". That is, "one can run the same piece on this machine and then on that machine" (Suber, 1988, §7 ("Liftability"), p. 103). Moreover, because it is essentially unembodied text, software "can be ported from one substratum to another. It is *liftable*" (Suber, 1988, §7, pp. 103–104). And it is "alterable" (Suber, 1988, §8 ("The Softness of Software"), p. 105). These are what distinguish it from hardware (Suber, 1988, §8). Alterability, of course, is what Moor cites as the essence of software. For Suber, that seems to follow from its being "pattern *per se*".

Does Suber really mean that *every* text is software—even random bits or "noise"? He claims that "software patterns do not carry their own meanings" (Suber, 1988, §6, p. 103). In other words, they are purely formal syntax, meaningless marks, symbols with no intrinsic meanings. If a computer can give meaning to a text, then it can read and execute it, according to Suber. But can any text be given a meaning by some computer? Yes, according to the Noiseless Principle.

We can summarize Suber's argument as follows: Software is text. As such, it has syntax, but no intrinsic semantics. For to be "meaningful"—readable and executable—it has to be interpreted by something else (for example, a computer) that can ascribe meaning to it (and that can execute its instructions). What about texts that are not programs (or that are not intended to be programs)? Consider a text such as this book, or random noise. If there is a device that can ascribe some meaning to such a text, then it, too, is readable and has the potential to be executable. (But what woud it mean to "execute" the chapter you are now reading?)

But texts need to be interpreted by a suitable computer: "They need only make a fruitful match with another pattern (embodied in a machine) [which] *we* create" (Suber, 1988, §6, p. 103, my italics). So, software is pure syntax and needs *another* piece of syntax to interpret it.

How can one piece of syntax "interpret" another? Recall from §9.5.1 that syntax is the study of the properties of, and relations among, symbols or uninterpreted marks. Roughly, semantics is the study of meaning, and—again, roughly—to say that a piece

[7] Don't forget that algorithms can be expressed in programming languages that are not "imperative", such as Lisp or Prolog.

of syntax *has* a meaning is to say that it is *related* to that meaning.[8] On this view, semantics is the study of the relations between two sets of entities: the syntactic objects and their meanings. But the meanings have their own properties and relations; that is, the meanings also have a syntax. So the syntax of the meanings can "interpret" the syntax of the software.

This is not far from Moor's definition: Both Moor and Suber require someone (or something) to interpret the syntax. Could a hardwired program and a written program both be software, perhaps because they have the same syntactic form? I think the answer is 'yes'. Here is a possible refinement: Software is a pattern that is read*able* and execut*able* by a machine. This is roughly Moor's definition of computer program. But, for Suber, *all* patterns are read*able* and execut*able*. The bottom line is that Suber's notion of *software* is closer to Moor's notion of computer *program*. The idea that software is pure syntax is consistent with the claim of Tenenbaum and Augenstein 1981, p. 6, that information has no meaning; recall their statement cited in §3.8. We'll come back to this idea in §14.3.3.

12.4.6 Colburn's Theory of the Nature of Software

Finally, Timothy R. Colburn (1999, 2000) argues that software is not "a machine made out of text". Thus, he would probably disagree with Hayes's definition of a computer as "magic paper". Colburn says this because he believes that there is a difference between software's "medium of description" and its "medium of execution". The former is the text in a formal language (something relatively abstract). The latter consists of circuits and semi-conductors (which are concrete). Consequently, Colburn says that software is a "concrete abstraction". But is this a single thing (a "concrete abstraction") or two things (a "medium of description" that is abstract and something else—a "medium of execution"—that is concrete)?

Colburn borrows the phrase from the title of an introductory CS textbook (Hailperin et al., 1999), which doesn't define it. All that Hailperin et al. say is that abstractions can be thought of "as actual concrete objects", and they give as an example a word processor, which they describe as an *abstraction* that is "merely [a] convenient way of describing patterns of electrical activity" *and* a "*thing* that we can buy, sell, copy, and use" (p. ix). Part of Colburn's goal is to explicate this notion of a thing that can be both abstract and concrete.

To do so, he offers several analogies to positions that philosophers have taken on the mind-body problem (see §2.8), so we might call this the "abstract-program/concrete-program problem". These positions are illustrated in Figure 12.4.

Consider various theories of *monism*, views that there is only one kind of thing: either minds or else brains, but not both. The view that there are only minds is called '*idealism*', associated primarily with the philosopher George Berkeley. The view that there are only brains is called '*materialism*' (or sometimes '*physicalism*'). Similarly, a monist with respect to software might hold that either software is abstract or else it is concrete, but it cannot be both. No matter how strong the arguments for, say,

[8] That's a controversial claim among philosophers. Some philosophers deny the existence of things that are meanings. Others would say that the meaning of a piece of syntax is the role that it plays in the language that it is part of. We'll discuss this further in Ch. 22.

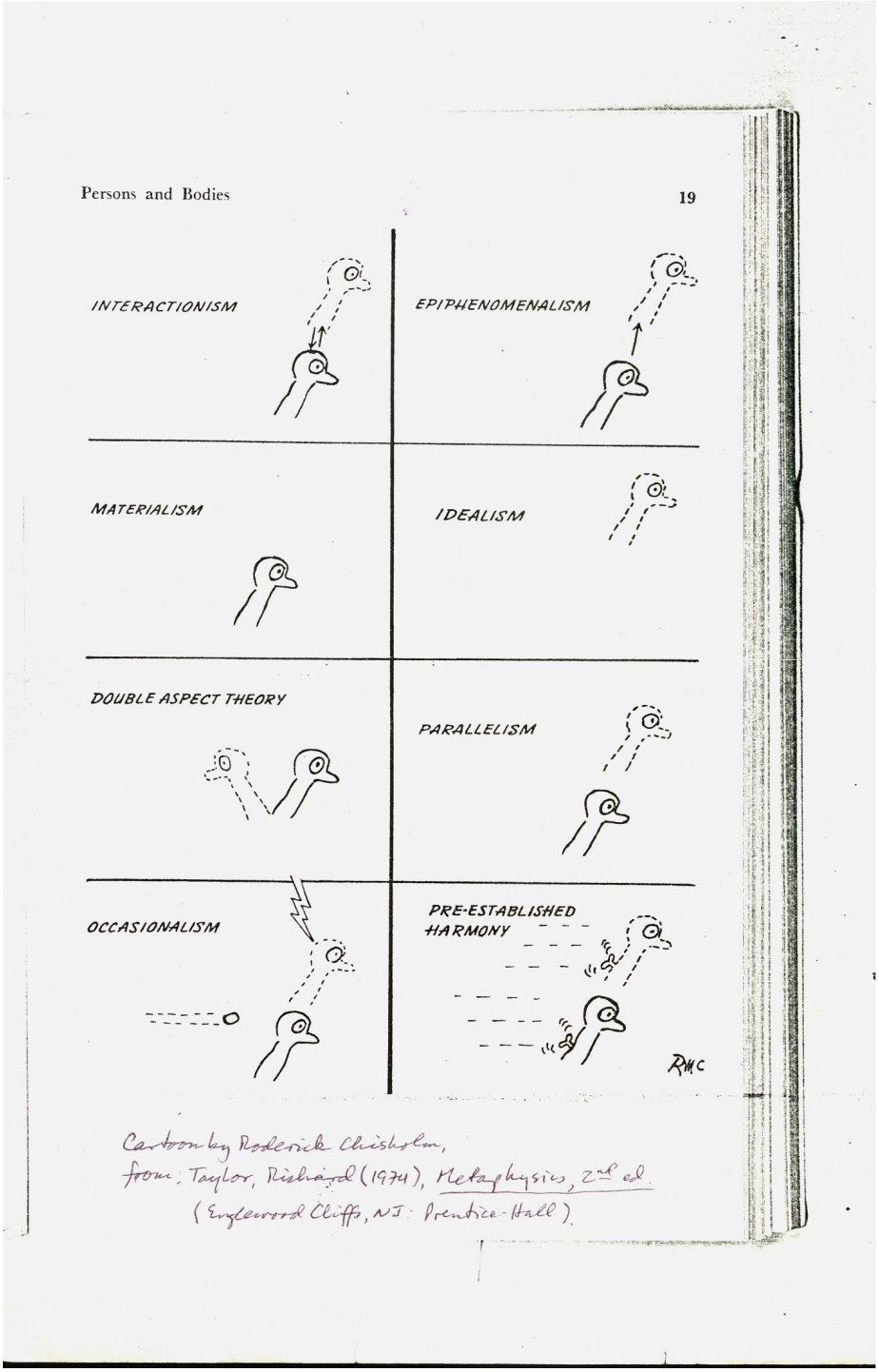


Figure 12.4: Positions on the mind-body problem (Chisholm, 1974, p. 19),
©1974, Prentice-Hall

materialism might be as the best answer so far to the mind-body problem, monism as a solution to the abstract-concrete problem fails to account for its dual nature (according to Colburn).

So let's consider various theories of *dualism*, views that there are both minds and brains. In the mind-body problem, there are several versions of dualism, differing in how they explain the relationship between minds and brains. The most famous version is called '*interactionism*', due to Descartes. This says that (1) there are minds (substances that think, but that are not physical, obeying only psychological laws); (2) there are brains (substances that are physically extended in space and that obey only physical and biological laws); and (3) minds and brains interact. The problem for interactionism as a solution to the mind-body problem is that there is no good explanation of *how* they interact. After all, one is physical and the other isn't. So you can't give a physical explanation of how they would interact, because the laws of physics don't apply to the mind. And you can't give a psychological explanation of how they interact, because the laws of psychology don't apply to the brain (according to Cartesian interactionism). Similarly, according to Colburn, when applied to the abstract-concrete problem, an interactionist perspective fails to account for the *relation between* abstractions and concrete things, presumably because the relations themselves are either abstract, in which case they don't apply to concrete things, or they are concrete and, so, don't apply to abstractions. (There is, however, a possible way out, which we will explore in depth in Chapter 14, namely, perhaps the relationship between them is one of implementation, or semantic interpretation, not unlike Suber's theory.)

A theory intermediate between monism and dualism is called the '*dual-aspect*' theory, due to Spinoza. Here, instead of saying that there are two different kinds of "substance", mental substance and physical substance, it is said that there is a single, more fundamental kind of substance of which minds and brains are two different "aspects". (For Spinoza, this more fundamental substance—which he believed had more than just the two aspects that we humans are cognizant of—was "nature", which he identified with God.) As a solution to the abstract-concrete problem, Colburn points out that we would need some way to characterize that more fundamental underlying "substance", and he doesn't think that any is forthcoming. Again, however, one alternative possibility is to think of how a single abstraction can have multiple implementations. Yet another alternative is a dual *property* view: Certain physical objects (in particular, brains) can have *both* physical *and* psychological *properties* (Chalmers, 1996a).

Finally, another family of dualisms is known as '*parallelism*': There are minds, and there are brains; they are not identical (hence this is a dualistic theory); and they do not interact. Rather, they operate in parallel, and so there is no puzzle about interaction. One version of parallelism called '*occasionalism*', says that God makes sure that, on every "occasion" when there appears to be interaction, every mental event corresponds to a parallel brain event (this keeps God awfully busy on very small matters!).

"*Pre-established harmony*"—which seems to be Colburn's favored version—says that God initially set things up so that minds and their brains work in parallel, much in the way that two clocks can keep the same time, even though neither causally influences the other. That way, God does not have to keep track of things once they have been set in motion. For Colburn, this seems to mean that implementation of an algorithm as a textual program parallels its implementation in the hardware of a physical computer:

> For the abstract/concrete problem we can replace God by the programmer who, on the one hand, by his [sic] casting of an algorithm in program text, describes a world of multiplying matrices, or resizing windows, or even processor registers; but on the other hand, by his act of typing, compiling, assembling, and link-loading, he causes a sequence of physical state changes that *electronically mirrors* his abstract world. —Colburn (1999, p. 17, my italics)

He puts this slightly differently in Colburn 2000, p. 208 (my italics), where he says that the "sequence of physical state changes ... *structurally matches* his abstract world", and he adds that "the abstract world of the computer programmer can be thought of as ticking along in preestablished synchrony with the microscopic physical events within the machine". The idea that the textual program and the physical state changes share a common structure is consistent with a view that a single abstraction can have two "parallel" implementations. But it is hard to imagine that the textual program (or even the abstract algorithm) can "tick along", because text—unlike the physical events—is static, not dynamic: It doesn't "tick".

Question for the Reader:
Do you think that "pre-established harmony" explicates "concrete abstraction"?
Is the mind a "concrete abstraction"?

A more modern take on the mind-body problem (not considered by Colburn) is "*functionalism*". Roughly, this is the view that certain abilities or purposes of the brain ("functions", but not in the sense of input-output pairs) are mental and *are* describable by the laws of psychology *in addition to* the laws of physics and biology (Putnam, 1960; Fodor, 1968; Levin, 2017). Rather than taking a position on the existence (or "ontological status") of something called "the mind", functionalism holds that what makes certain brain activity *mental* in addition to being *physical* is the role that it plays—its "function"—in the overall activity of the brain or the person.

Question for the Reader:
What would a functional solution to the abstract-concrete problem look like?
Might we say that some hardware *functions* as a computer program?

Further Reading:
Duncan 2017 surveys Moor, Suber, and Colburn, but argues that a formal ontology is needed before a useful distinction between hardware and software can be made, and that, on the Basic Formal Ontology (http://ifomis.uni-saarland.de/bfo/), "a piece of computational hardware is an ontologically independent entity, whereas a software program is an ontologically dependent entity."

12.5 Summary

Algorithms—which are abstract—can be implemented as programs (that is, as texts written in a computer-programming language). Programs, in turn, can be implemented as part of the hardware of a computer. A given algorithm can be implemented differently in different programs, and a program can be implemented differently in different computers. Both Suber's and Colburn's theories of software focus on this *implementational* aspect. Moor's theory focuses on the *changeability* of software. Presumably, the more abstract an entity is, the easier it is to change it. So the software-hardware distinction may be more of a continuum than something with a sharp boundary. Moreover, you can't really talk about a program or software *by itself*, but you have to bring in the computer or other entity that interprets it: Programs and software are *relational* notions.

We need to explore the notion of implementation in more detail, which we will do in Chapter 14, but first we are going to consider another ontological puzzle that arises from the dual nature of software, this time in the legal realm: Can programs be copyrighted? Or should they be patented instead? After all, if a program is a written text, then it can be copyrighted, but, if it is a machine, then it can be patented—yet nothing can (legally) be both copyrighted and patented!

12.6 Questions for the Reader

1. > Turing's work clearly showed the extensive interchangeability of hardware and software in computing. —Hartmanis (1993, p. 11)

   Tanenbaum 2006, p. 8, points out that hardware and software are "logically equivalent" in the sense that anything doable in hardware is doable in software, and vice versa. Similar or analogous cases of such logical equivalence of distinct things are Turing machines, the lambda-calculus, and recursive functions. Also, such an equivalent-but-different situation corresponds to the intensional-extensional distinction: Two intensionally distinct things can be extensionally identical.

   How does this equivalence or "interchangeability" relate to Moor's or Colburn's definitions of software and hardware?

2. > Academically and professionally, computer engineering took charge of the hardware, while computer science concerned itself with the software .... —Mahoney (2011, p. 108)

   If software and hardware can*not* easily be distinguished, does that mean that neither can computer engineering and computer science?

3. Find a (short) article on the mind-body problem (for example, the *Wikipedia* article at http://en.wikipedia.org/wiki/Mind-body_problem). Replace all words like 'mind', 'mental', etc., with words relating to 'software'; and replace all words like 'body', 'brain', etc., with words like 'hardware', 'computer', etc. Discuss whether your new paraphrased article makes sense, and what this says about the similarities (or differences) between the mind-body problem and the software-hardware problem.[9]

4. Recall from §11.4.3.2 that Wadler (1997, pp. 240–241) said that "Interaction is the mind-body problem of computing." He was not referring to the kind of interaction that Descartes's dualism requires; rather, he was referring to computers that interact with the real world or with an oracle, such as we discussed in Chapter 11.

   Nevertheless, how do the various positions on the mind-body problem relate to Wadler's observation?

5. In line with Suber's view that any text—even noise—is readable and executable, consider the presentation in Denning and Martell 2015, p. 40, Fig. 3.3, of Shannon's theory of the communication of information, which suggests that the encoding of a message is communicated along a channel to be decoded by a receiver. This is very similar to the idea of input being converted into output. Of special interest is the idea that "noise is any disruption that alters the signal". Wouldn't a computation performed on an input signal alter it? In that case, could one consider such a computation to be noise? Or could one consider noise to be a computation?

[9] Thanks to James Geller for this idea.

6. Recall the discussion in §12.4.2 on the relationship of software to music, art, and literature.

   What do you think Moor or Suber might say about it? Would Moor disagree? After all, art is not usually changeable. Would Suber be more sympathetic? And what about Colburn? Are any art forms "concrete abstractions"?

Chapter 13

Should Software Be Copyrighted or Patented?

Version of 6 March 2019; DRAFT © 2004–2019 by William J. Rapaport

Figure 13.1: ©2010, *American Scientist*

13.1 Readings:

1. Required:

   - Newell, Allen (1985-1986), "The Models Are Broken, the Models Are Broken", *University of Pittsburgh Law Review* 47: 1023–1031, http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=356219
     - For follow-up (including links to commentary by Donald Knuth), see: "Newell 1986: The Models Are Broken", https://web.archive.org/web/20120527101117/http://eupat.ffii.org/papers/uplr-newell86/index.en.html

2. Very Strongly Recommended:

   (a) Samuelson, Pamela (1990), "Should Program Algorithms Be Patented?", *Communications of the ACM* 33(8) (August): 23–27, https://www.law.berkeley.edu/php-programs/faculty/facultyPubsPDF.php?facID=346&pubID=94

   (b) Samuelson, Pamela; Davis, Randall; Kapor, Mitchell D.; & Reichman, J.H. (1994), "A Manifesto Concerning the Legal Protection of Computer Programs", *Columbia Law Review* 94(8, December): 2308–2431, http://scholarship.law.duke.edu/cgi/viewcontent.cgi?article=1783&context=faculty_scholarship
      - From a special issue of the *Columbia Law Review* on the legal protection of computer programs. A summary version appears as Davis et al. 1996. Other articles in that special issue elaborate on, or reply to, Samuelson et al. (1994).
      - §1 is a good overview; §2 (especially §2.2) is also good, as are §5 and the Conclusion section.

   (c) Samuelson, Pamela (2015), "Software Patents Are Falling Down", *Communications of the ACM* 58(11) (November): 27–29.

3. Strongly Recommended:

   - PolR (2009), "An Explanation of Computation Theory for Lawyers", *Groklaw* (11 November), http://www.groklaw.net/article.php?story=20091111151305785, http://www.groklaw.net/staticpages/index.php?page=20091110152507492, and http://www.groklaw.net/pdf2/ComputationalTheoryforLawyers.pdf
     - "Computers don't work the way some legal documents and court precedents say they do."
     - "The phrase 'effective method' is a term of art [in mathematics and philosophy]. This term has nothing to do with the legal meaning of 'effective' and 'method'. The fact that these two words also have a meaning in patent law is a coincidence."

13.2 Introduction

We are trying to understand what algorithms, computer programs, and software are, and how they are related to each other. One way to approach these questions is by considering the legal issues of whether any of them can be copyrighted or patented. The issue of whether software can or should be patented or copyrighted is not merely an ontological issue. It is also, perhaps even more so, an economic issue (Galbi, 1971). However, in this chapter, we are concerned more with the ontology of software than with legal or economic issues in themselves.

One of the first questions is what kind of entity might be copyrighted or patented:

algorithms?
: These seem to be abstract, mathematical entities. Can abstract, mathematical entities—such as numbers, formulas, theorems, proofs, etc.—be copyrighted or patented?

computer programs?
: These seem to be "implementations" of algorithms, expressed in a programming language. We have seen that programs might be analogous to *numerals*, whereas algorithms might be analogous to *numbers*—can numerals be copyrighted or patented?

programs written on paper?
: These might seem to be "literary works", like poems or novels, which can be copyrighted but not patented. (Recall the discussion on software and art in §12.4.2.)

programs implemented in hardware (that is, "machines")?
: If programs are linguistic implementations of algorithms, then are hardware implementations of programs thereby implementations of implementations of algorithms? And physical implementations might seem to be the kind of thing that is patentable but not copyrightable.

software?
: 'Software' might be a neutral term covering both algorithms and programs, or software might be a more controversial entity not necessarily indistinguishable from certain kinds of hardware, as we saw in the previous chapter. But only the former might be copyrightable, and only the latter might be patentable.

Because 'copyright' and 'patent' are legal terms, we need to look at their "official", legal definitions. Note what these definitions say about computer programs, procedures, methods, and processes. First, here is a lengthy excerpt from a definition of 'copyright' in an informational brochure published by the US Copyright Office:

> Copyright is a form of protection provided by the laws of the United States[1] to the authors of "original works of authorship" that are fixed in a tangible form of expression. An original work of authorship is a work that is independently created by a human author and possesses at least some minimal degree of creativity. A work is "fixed" when it is captured (either by or under the authority of an author) in a sufficiently permanent medium such that the work can be perceived, reproduced, or communicated for more than a short time. ...
>
> **What Works Are Protected?**
> Examples of copyrightable works include
>
> - Literary works
> - Musical works, including any accompanying words
> - Dramatic works, including any accompanying music
> - Pantomimes and choreographic works
> - Pictorial, graphic, and sculptural works
> - Motion pictures and other audiovisual works
> - Sound recordings, which are works that result from the fixation of a series of musical, spoken, or other sounds
> - Architectural works
>
> These categories should be viewed broadly for the purpose of registering your work. For example, computer programs ... can be registered as "literary works"; ...
>
> **What Are the Rights of a Copyright Owner?**
> Copyright provides the owner of copyright with the exclusive right to
>
> - Reproduce the work in copies or phonorecords ...
> - Prepare derivative works based upon the work
> - Distribute copies or phonorecords of the work to the public by sale or other transfer of ownership or by rental, lease, or lending
> - Perform the work publicly if it is a literary, musical, dramatic, or choreographic work; a pantomime; or a motion picture or other audiovisual work
> - Display the work publicly if it is a literary, musical, dramatic, or choreographic work; a pantomime; or a pictorial, graphic, or sculptural work. This right also applies to the individual images of a motion picture or other audiovisual work.
> - Perform the work publicly by means of a digital audio transmission if the work is a sound recording ...

[1] WJR footnote: Title 17, *U.S.Code*, https://www.copyright.gov/title17/

What Is Not Protected by Copyright?
Copyright does not protect

- Ideas, procedures, methods, systems, processes, concepts, principles, or discoveries
- Works that are not fixed in a tangible form (such as a choreographic work that has not been notated or recorded or an improvisational speech that has not been written down)
- Titles, names, short phrases, and slogans
- Familiar symbols or designs
- Mere variations of typographic ornamentation, lettering, or coloring
- Mere listings of ingredients or contents

—"Copyright Basics" (September 2017),
https://www.copyright.gov/circs/circ01.pdf

And here is a definition of 'patent' from the US Patent and Trademark Office's website (https://www.uspto.gov/):

What is a Patent?
A patent for an invention is the grant of a property right to the inventor, issued by the United States Patent and Trademark Office. ... The right conferred by the patent grant is, in the language of the statute and of the grant itself,[2] "the right to exclude others from making, using, offering for sale, or selling" the invention in the United States or "importing" the invention into the United States. What is granted is not the right to make, use, offer for sale, sell or import, but the right to exclude others from making, using, offering for sale, selling or importing the invention. ...

There are three types of patents:

1. **Utility patents** may be granted to anyone who invents or discovers any new and useful process, machine, article of manufacture, or composition of matter, or any new and useful improvement thereof;
2. **Design patents** may be granted to anyone who invents a new, original, and ornamental design for an article of manufacture; and
3. **Plant patents** may be granted to anyone who invents or discovers and asexually reproduces any distinct and new variety of plant.

— United States Patent and Trademark Office,
"General information concerning patents",
https://www.uspto.gov/patents-getting-started/general-information-concerning-patents

[2]WJR footnote: The relevant laws are the US Constitution, Article I, §8; and various laws cited under "Patent Laws" at https://www.uspto.gov/patents-getting-started/general-information-concerning-patents

On the website for utility patents, we find this:

> **Specification**
> The specification is a written description of the invention .... For inventions involving computer programming, computer program listings may be submitted as part of the specification ....
> —"Nonprovisional (Utility) Patent Application Filing Guide", https://www.uspto.gov/patents-getting-started/patent-basics/types-patent-applications/nonprovisional-utility-patent

Further Reading:
For general discussion of the issues, see: Forester and Morrison 1994, Ch. 3, pp. 57–68; D. Johnson 2001a, Ch. 6, pp. 137–167 ("Property Rights in Computer Software"); Klemens 2006; Boyle 2009.

A more detailed discussion of computer patents can be found in "Computer Systems Based on Specific Computational Models", https://www.uspto.gov/web/patents/classification/cpc/pdf/cpc-definition-G06N.pdf.

In addition to copyrights and patents, there is also a legal notion of "trade-secret protection", which we will not explore. See https://www.uspto.gov/patents-getting-started/international-protection/trade-secret-policy; Bender 1986; and Samuelson et al. 1994, §§2.2.1 and 5.3.3.

Shaw et al. 2012 is a psychological study of children's views on the ownership of ideas:

> Adults apply ownership not only to objects but also to ideas. But do people come to apply principles of ownership to ideas because of being taught about intellectual property and copyrights? ... [L]ike adults, children as young as 6 years old apply rules from ownership not only to objects but to ideas as well. (p. 1383)

13.3 Preliminary Considerations

Let's begin by trying to distinguish between algorithms and "corresponding" computer programs. Recall that algorithms specify how to compute functions. So let us begin by considering a function, that is, a set of input-output pairs that are functionally related. That function can be "implemented" by different algorithms.

What do I mean by 'implemented'? As promised, we'll explore that in Chapter 14. A synonym for 'implemented' is 'realized' (that is, made real; in French, a film director is a "réalisateur"—a "realizer"). For now, we will just say that "implementation" is the relation between a function and an algorithm that computes it: Any such algorithm "implements" that function.

Take a look at Figure 13.2. In that figure, merely as an example, I consider a function being implemented by two different serial algorithms and one parallel algorithm. For a concrete example, consider a sorting function that takes as input a set of students' names and that yields as output a sequence of those names sorted alphabetically. One serial algorithm that implements this function is Quicksort; another is merge sort. A parallel algorithm that implements the function might be a parallel version of merge sort.

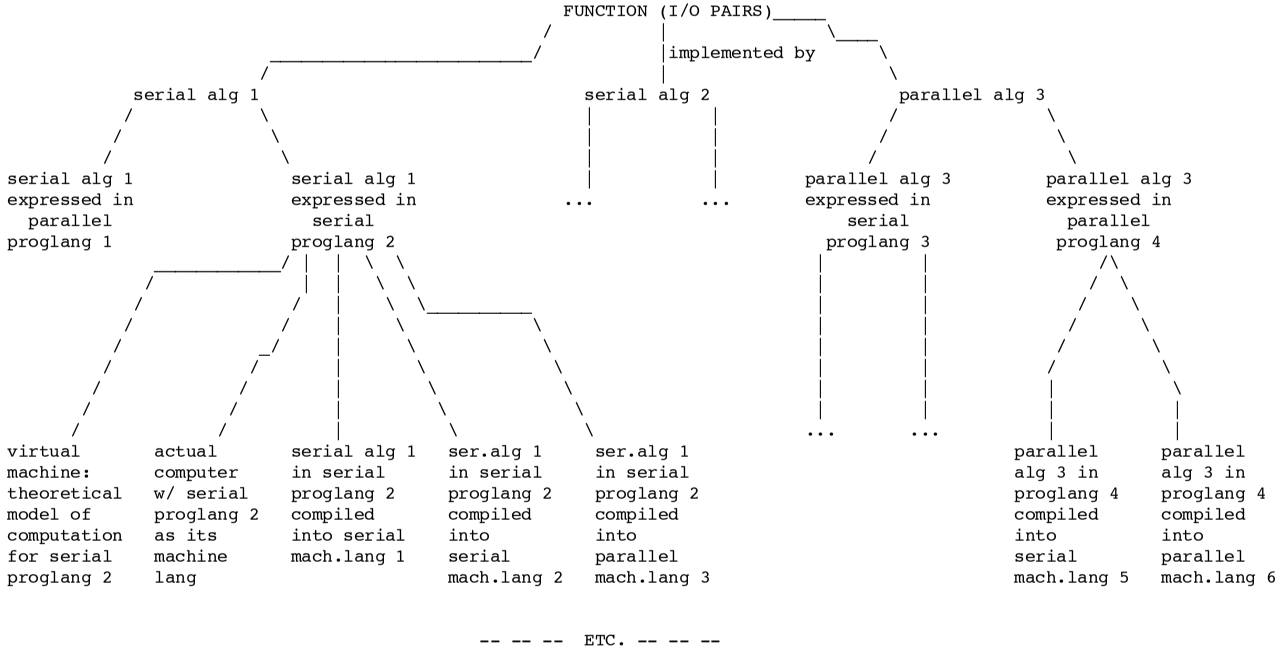


Figure 13.2:

WHICH (IF ANY) OF THESE SHOULD BE COPYRIGHTABLE?

WHICH (IF ANY) OF THESE SHOULD BE PATENTABLE?

Copyright © 2010 by William J. Rapaport (rapaport@buffalo.edu)
http://www.cse.buffalo.edu/~rapaport/584/S10/cvspat.html-20100319

Take a look at serial algorithm 1 in that figure. The figure suggests that it might be expressable in two different programming languages: a parallel programming language and a serial programming language.

Consider now our original function implemented by serial algorithm 1 and that is, in turn, expressed in serial programming language 2. There might be several ways in which that program might itself be implemented:

- on a virtual machine—a theoretical model of computation for our serial programming language 2,
- on an actual computer that uses our serial programming language 2 as its machine language,
- compiled into a serial machine language,
- compiled into a different serial machine language,
- compiled into a parallel machine language.

And now consider parallel algorithm 3 that implements our original function. It might be expressed in a *serial* programming language 3, which itself might be implemented in a virtual machine, an actual computer, or compiled into several different machine languages. Or it might be expressed in a *parallel* programming language 4, which itself might be compiled into serial machine language 5 or else into a parallel machine language. And so on.

- Which (if any) of these should be copyrightable?
- Which (if any) of these should be patentable?

13.4 Copyright

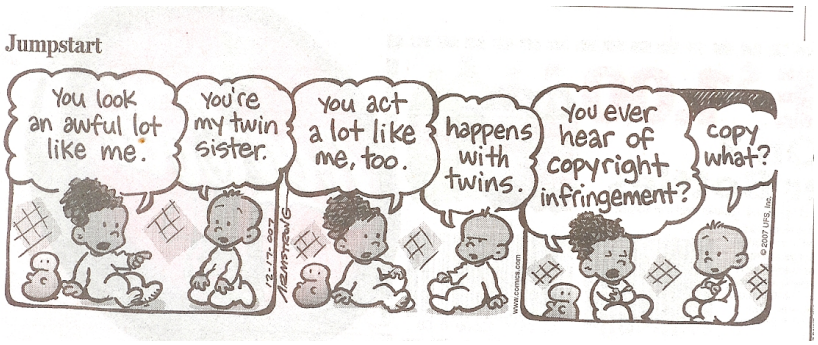


Figure 13.3: https://www.gocomics.com/jumpstart/2007/12/17, ©2007, UFS, Inc.

What does it mean to copyright something? For a person to copyright a *text* is to give that person the legal *right* to make *copies*—hence the term 'copyright'—of that text for

the purpose of protecting the *expression of ideas*. An *idea* cannot be copyrighted; only the *expression* of an idea can be copyrighted. Ideas are abstract, perhaps in the way that algorithms are abstract. But if you express that idea in language—if you "implement" it in language—then the expression of that idea in that language *can* be copyrighted, perhaps in the way that a computer program that implements an algorithm might be copyrightable.

In an essay on software copyrights published in a computing journal, Calvin N. Mooers (1975, p. 50) distinguishes between an uncopyrightable idea and a copyrightable expression of an idea:

> Where does the "expression" leave off, and the "idea" take over? The best insight into this matter comes from discussions of copyright as applied to novels and dramatic productions. In these, "expression" is considered to include the choice of incident, the personalities and development of character, the choice of names, the elaboration of the plot, the choice of locale, and the many other minor details and gimmicks used to build the story. In other words, "expression" is considered to include not only the marks, words, sentences, and so on in the work, but also all these other details or structures as they aggregate into larger and larger units to make up the expression of the entire story.
>
> In other words, after the bare abstract "idea" has been chosen (e.g., boy meets girl, boy loses girl, boy wins girl), the "expression" to which copyright applies covers the remaining elements of original choice and artistry which are supplied by the author in order for him [sic] to develop, express, and convey his version of the bare idea.

Consider some domain W of things in the world, together with their properties and the relations among the things. And consider a description D of this world in some language L. The things, properties, and relations of W can be taken as a semantic interpretation of D in L (its nouns, verbs, and modifiers). Perhaps copyright is something that applies only to the realm of language and syntax, whereas patents are things that apply only to entities in the world.

Algorithms, construed abstractly, seem more like "ideas" than like "expressions", which would suggest that they cannot be copyrighted. On this view, it would be a computer program—a text written in some language—that would be copyrightable. Programs are definitely expressions: They are "non-dramatic literary works". Why "literary"? After all, they don't seem to read like novels! (But recall the Digression in §12.4.2 on "literate computing".) But all that 'literary' means in this context is that they can be written and read. Moreover, programs can be "performed", that is, executed, just like lectures, plays, movies, or music. The relation of a process (that is, a program being executed; see §3.9) to a program is similar to the relation of a script to a play or movie, a score to a musical performance, or (perhaps) a set of slides to a lecture.

I posed the following question to a friend who is a lawyer specializing in entertainment law. First, I explained the ontological puzzle:

> Some people argue that programs are like the scripts of plays; because plays can be copyrighted (and not patented), so should programs. Others argue that programs are more like machines; because machines can be patented (but not copyrighted), so should programs. Finally, others argue that programs are unlike play

> scripts (because it's not just the text that matters—it's what they can do—and, anyway, most people can't read the "text" of a program, as they can for a play) *and* that they are unlike machines (because a computer without a program can't do anything); hence, there should be some new kind of legal protection mechanism.
>
> Presumably, if I go to a bookstore and buy a copy of the script for, say, *Oklahoma*, I'm not liable for copyright infringement if I read it, or even if I act it out with some friends (as long as I don't charge admission). Similarly, if I buy a DVD of the movie version, I'm not liable for copyright infringement if I watch it, even if I watch it with some friends (again, as long as I don't charge admission?). But, if I decide to mount a production of it for my local little theater group, I'd have to get permission.
>
> Now, my question: What is the legal basis for that permission requirement? Is it the copyright law? Or is it some other legal protection mechanism?

Here is her reply:

> Terrific question. Two things. First of all, under the copyright law, permission is required and royalties may be charged for a public performance even if no admission is charged. If you read a play with friends in your living room, that is probably not public, but if you do it anywhere else and people come to see it, even just people you know, with no admission charge, I consider that a public performance, requiring authors' permission. The legal basis for the permission requirement is in the Copyright Act (Section 106), which gives the Copyright Owner the exclusive right to "do and authorize any of the following: ... (4) in the case of literary, musical, dramatic and choreographic works, pantomines, and motion pictures and other audiovisual works, to perform the copyrighted work publicly."
>
> Section 117 relates to Limitations on exclusive rights; Computer programs.[3]
>
> I hope this is helpful.
>
> —Victoria G. Traube (personal email, 23 September 2014)

On the other hand, there is an entity that is even more abstract than an algorithm, and with respect to which it is the algorithm that appears to be a detailed "expression", namely, the *function* that the algorithm computes. So, one could argue that it is a *function* that is more like a "bare abstract idea" (boy meets girl, etc.) and that it is an *algorithm*—which would have to specify *how* the boy meets the girl, etc.—that is more like an expression (in this case, an "expression" of a function). On this view, it is the *algorithm* that would be copyrightable!

So, here are two problems to think about:

Problem #1

> Consider the input-output behavior of a program-being-executed (a "process". That is, consider how the process "looks and feels" to a user. A clever programmer could create a new algorithm (or recreate an old one) and then implement a new program that will have the same look and feel. Should that new program be separately copyrightable?
>
> Consider two computer design companies; call them 'Pear Computers' and 'Macrohard'. Pear Computers might write an operating system with "windows",

[3] WJR footnote: http://www.law.cornell.edu/uscode/text/17/117

"icons", and a "mouse". Macrohard, seeing how this operating system works, might write their own operating system that also uses "windows", "icons", and a "mouse" and that would have the same (or almost the same) functionality and the same (or almost the same) "look and feel", but it would be a different expression of that same (or almost the same) idea, hence separably copyrightable. Or would it?

Further Reading:
This is not far from the actual situation that allegedly existed between the Xerox Palo Alto Research Center and Apple Computer, on the one hand, and (later) between Apple and Microsoft, on the other. The story is that Apple "borrowed" or was inspired by Xerox PARC's creation of the graphical user interface and used it for its Mac OS, and that Microsoft "borrowed" or was inspired by Apple's version and used it for Windows. See Fisher 1989; Samuelson et al. 1994, pp. 2334–2335; Gladwell 2011; and http://en.wikipedia.org/wiki/Apple_Computer,_Inc._v._Microsoft_Corp. (note: the period after 'Corp' is part of this URL!).

More recently, Apple successfully sued Samsung for a similar infringement of its design for smartphones. See:
https://www.nytimes.com/2018/05/24/business/apple-samsung-patent-trial.html,
https://en.wikipedia.org/wiki/Apple_Inc._v._Samsung_Electronics_Co., and
https://finance.yahoo.com/news/apple-google-samsung-steals-ideas-151723862.html

Dennett's (1995, pp. 372–373) two ways of building a dam (the beaver's and the human's, which we discussed briefly in §12.4.4.1.2.2) arguably have the same "look and feel" even though the behaviors are very different.

Problem #2

It is part of copyright law that you cannot copyright an "article of utility"; that's something that can only be patented. But surely an executable program is useful; that is, it is an "article of utility". But, according to Mooers, it is a translation of a copyrightable text; hence, it, too, is copyrightable.

One diagnosis of these problems is that there is no legal definition of "look and feel" (Samuelson, 1989). Another diagnosis is that the distinction between an idea and its expression may not be applicable in the realm of software (Samuelson, 1991). We will explore this in §13.7.

It's one thing to wonder whether software in the form of a computer program is legally protected (whether by copyright, patent, or something else). But what about the programming language that the program is written in? On the one hand, a programming language can itself be viewed as a form of software (it's surely not hardware). On the other hand, while an author of a novel written in English can copyright that novel, English itself cannot be copyrighted. (But what about "created" languages, like Esperanto or Klingon?)

In a 2012 legal case, Oracle sued Google for Google's use of the Java programming language, which Oracle claimed to own the copyright on. Google's defense was

that programming languages are not copyrightable. The first decision was in favor of Google, but, on appeal, Oracle seems to have won (https://www.wired.com/story/the-case-that-never-ends-oracle-wins-latest-round-vs-google/).

Unfortunately, the issues aren't as clear cut as they might be, because it's not so much the Java language that is in question, as its "application programming interfaces" (API), which may, or may not, be considered to be part of the language itself (as opposed to software written in the language). And, to further the complications, Google apparently used an "open-source" (that is, not copyright-protected) version of the Java APIs.

Further Reading:
The original briefs by Oracle and by Google can be found at Oracle America 2012 and Google, Inc. 2012. Good overviews are Macari 2012, McAllister 2012, McSherry 2014, Samuelson 2015, and Sprigman 2015; see also https://en.wikipedia.org/wiki/Oracle_America,_Inc._v._Google,_Inc.

Samuelson 2007a suggests that copyright law needs to be reformed. Menand 2014 surveys the history of copyright laws, especially as they apply to works of art.

AI researcher David Touretzky (2008, my interpolated brackets) notes that "If code that can be directly compiled and executed may be suppressed under the DMCA [Digital Millenium Copyright Act], as Judge [Lewis A.] Kaplan asserts in his preliminary ruling, but a textual description of the same algorithm may not be suppressed, then where exactly should the line be drawn? This web site was created to explore this issue, and point out the absurdity of Judge Kaplan's position that source code can be legally differentiated from other forms of written expression." For example, the website used to offer a copy of otherwise suppressed code on a T-shirt!

13.5 Patent

Patents are weird: Not just *any*thing can be patented, but lots of strange things *can* be! I have read patents for devices or techniques that claim to be based on carefully developed scientific theories, but that I know, from personal experience, are not based on any such thing, and I have read patents that claim to provide mechanisms for completely solving problems that have not yet been solved, and may never be—such as fully automatic, high-quality machine translation.

The purposes of a patent are to foster invention and to promote the disclosure of an invention. So, the inventor's *ideas* must be made *public*. (Does that mean that they have to be implemented?)

It follows that you cannot patent something that would *inhibit* invention, namely, you cannot patent any of the following:

- abstract ideas
- mathematical processes
- scientific principles

- laws of nature (and yet genes have been patented!)
- mental processes

But you *can* patent "any new and useful process, machine, article of manufacture, or composition of matter, or any new and useful improvement thereof" (see the definition of 'utility patent' in §13.2, above). Does "process" here include software, or a program being executed?

Here are some problems to consider:

- Each step of certain cognitive algorithms (such as those used in reasoning, perception, or natural-language generation and understanding) are mental processes. Does that mean that some algorithms cannot be patented, even if others can? If so, how does one distinguish between these two kinds of algorithms?
- Lots of software are mathematical processes. Does this mean that some, but not all, software cannot be patented?
- If too much software is patented, would that *impede* invention? If so, that would be contrary to one of the explicit purposes of patent law.

In any case, algorithms cannot be patented, so copyright would seem to be the only option:

> Abstract ideas are not patentable subject matter. In order to prevent abstract ideas from being indirectly patented, there are exceptions in the law that will not allow patenting an "invention" made of printed text based on the content of the text. The implication of Gödel [numbering] ... is that the symbols that represent an abstract idea need not be printed text. The symbols can be something abstract like exponents of prime numbers. They can be something physical like electromagnetic signals or photonic signals. ... Does the law exempt abstract ideas from patenting only when they are written in text or does it exempt all physical representations? —"PolR" (2009, §"Symbols and Reducibility", my interpolated brackets)

This seems to be another version of the software-hardware paradox that we explored in §12.4.3. Here's another:

> Now consider the Church-Turing thesis that computer programs are the same thing as recursive functions. ... Would this be of consequence to the patentability of software? Developing a computer program that fulfills a specific purpose is mathematically the same thing as finding a number that has some specific properties. Can you patent the discovery of a number?
>
> Another way to look at it is that there is a list of all the possible programs.[4] We can make one by listing all possible programs in alphabetic order. Any program that one may write is sitting in this list waiting to be written. In such a case, could a program be considered an invention in the sense of patent law? Or is it a discovery because the program is a preexisting mathematical abstraction that is merely discovered?

[4] WJR footnote: Recall Mill's observation about there being only a finite number of musical compositions, mentioned in §12.4.2, above.

> This idea that programs are enumerable numbers is supported by the hardware architecture of computers. Programs are bits in memory. Strings of 0s and 1s are numbers written in binary notations. Every program reads as a number when the bits are read in this manner. Programming a computer amounts to discovering a number that suits the programmer's purpose. —"PolR" (2009, §"Enumerability")

Further Reading:
Written in a sarcastic manner by a well-known novelist, Crichton 2006 presents "A case for reforming the way we grant patents", on the grounds that natural laws should not be patentable. For follow-ups, see "LabCorp. v. Metabolite", http://www.oyez.org/cases/2000-2009/2005/2005_04_607, and Edwards 2013.

Petroski 2008b discusses the role of implementation: "As with any well-written patent, in the one granted to McHenry he is careful not to restrict his invention to only one 'embodiment' ..." (p. 192).

Samuelson 2008 asks, "Is everything under the sun made by humans patentable subject matter?", and Samuelson 2013 looks at the patentability of software.

For general comments on patents, see: Thatcher and Pingry 2007 (with commentary in Oldehoeft et al. 2007), the blog "Time to Abolish Software Patents?" (*Slashdot*, 29 February 2008, http://yro.slashdot.org/story/08/02/29/0344258/time-to-abolish-software-patents), and *New York Times* 2009.

13.6 Virtual Machines

Some of the issues involved in the copyright-vs.-patent debate concern the notion of a virtual machine. We have considered these before (in Chapter 9), but now it is time to ask what exactly a virtual machine is.

According to Denning and Martell (2015, p. 212), "A virtual machine is a simulation of one computer by another. The idea comes from the simulation principle behind Alan Turing's Universal Machine." That is, a virtual machine is a (single-purpose)[5] Turing machine that is simulated by a universal Turing machine: Let t be a Turing machine. Let u be a universal Turing machine. Encode all Turing machines using, say, Turing's coding scheme. Then encode t's code onto u's tape, along with t's data, and let u simulate t. As Copeland (1998, p. 153) says, "the universal machine will perform every operation that t will, in the same order as t (although interspersed with sequences of operations not performed by t)". The virtual t machine is a software version (a software implementation?) of the hardware t machine.

[5] Of course, nothing prevents the Turing machine that is being simulated by the universal Turing machine from itself being a universal Turing machine! This was the case with the "P88 Assembly Language Simulator" mentioned in §9.2. In fact, that was a virtual universal Turing machine simulated on another virtual universal Turing machine that was simulated on a real, physical Mac—yet another universal Turing machine!

Further Reading:
For more on virtual machines, see Popek and Goldberg 1974, Pylyshyn 1992, and—especially—Pollock 2008, who argues that *we* are virtual machines! Sloman 2008 views virtual machines as mathematical abstractions that can have causal relations with the physical world.

If software, generally speaking, is copyrightable (but not patentable), and if hardware, generally speaking, is patentable (but not copyrightable), what about a virtual machine, which is a software implementation of a piece of hardware?

Pamela Samuelson et al. (1994) argue "that programs should be viewed as virtual machines" (p. 2324), and they discuss the notion in terms of "conceptual metaphors", such as the metaphor of "paper" for a word-processing program (§1.3):

> Cutting and pasting (in a virtual sense) is so easily accomplished in a word processor that one soon begins to think of the text as a physical entity that can be picked up and moved around. (p. 2325)

By 'virtual' here, they mean "metaphorical". Note, in connection with our discussion of magic in §3.14.7, that this "virtual" paper is an "illusion": As Samuelson et al. note, we have the illusion that, when inserting text, "the old text obligingly moves over to make room for the new words" (p. 2324).

> The automated spreadsheet metaphor has so fundamentally changed the experience of using a computer that users feel they are using a spreadsheet, not just a computer. (p. 2325)

They go on to argue that, because "conceptual metaphors" (that is, virtual machines) "are remote from the program text ... and partly because of their abstract character, they would likely be regarded as unprotectable by copyright law" (p. 2326). So, does that mean that they are patentable?

Initially, "a computer operating in accordance with a particular program" was patentable "if the program is new, useful, and unobvious" (Galbi, 1971, p. 274). That is, it was the physical machine executing the program that was patentable. This was later modified to allow the "method or algorithm for solving certain problems" to be patentable "irrespective of what apparatus is used to solve the problem" (Galbi, 1971, p. 274)—a clear move away from the patentability of a piece of *hardware* to the patentability of its *software* instead. Arguably, though, this could be seen as a move to say that it is still a machine that is patentable, but it doesn't have to be a specific machine: It could be any machine that is executing the program under consideration.

In other words, the *virtual* machine might not be patentable, but—because all virtual machines must be implemented in a physical machine—the physical machine that implements the virtual machine might be patentable. But this only makes sense if the physical machine is a special-purpose one that does nothing except implement that virtual machine. If it were, instead, a universal Turing machine, then it would only be patentable as, perhaps, some new kind of universal Turing machine, but not as an implementation of the virtual machine.

However, Dennett (1995, p. 232) tells of a dedicated word processor—that is, what appears to the user to be a single-purpose Turing machine that does nothing but word processing. In fact, though, it is powered "by an all-purpose CPU"—that is, by a

universal Turing machine. But the user has no access to the universal machine's other capabilities. (It turned out to be less expensive to have the word processor run on a univeral machine than to develop a special-purpose one.)

Related to this is the following excerpt from a footnote to a 1969 ruling in a case called *Prater and Wei*:

> In one sense, a general purpose digital computer may be regarded as a storeroom of parts and/or electrical components. But once a program has been introduced, the general purpose digital computer becomes a special purpose digital computer (i.e. a specific electrical circuit with or without electromechanical components) which along with the process by which it operates, may be patented, subject, of course, to the requirements of novelty, utility and nonobviousness.
> —Galbi (1971, p. 275)

But think of a smartphone capable of running multiple "apps" or a computer capable of running multiple programs—more generally, of a universal Turing machine capable of simulating multiple Turing machines. Might (a physical implementation of) a universal Turing machine running one program be patentable, whereas the very same machine running a different program not be patentable? Is it one machine, or many?

13.7 Samuelson's Analysis

Pamela Samuelson (1990, p. 23) points out that computer scientists differ on what is the proper object of legal protection: Algorithm? Code (or program)? "Look and feel"? So, these need to be distinguished. By now, you should have a pretty good idea what algorithms and programs are. But what is "look and feel"? Is it related (or identical) to the "process", that is, the program as it is being executed? Or is it merely the output of the program?

According to Samuelson and her colleagues,

> Although programs are texts and their texts can be valuable, the most important property of programs is their behavior (i.e., the set of results brought about when program instructions are executed). ...
>
> ... the primary source of value in a program is its behavior, not its text.
> —Samuelson et al. (1994, p. 2314–2315)

Does this mean that it's the "idea", not its "expression" that counts? If so, then programs might *not* be copyrightable! I don't think so, simply because the "behavior" of a program is more than merely the "idea". (The "idea" might be more akin to the "specifications", or the input-output behavior.) But "behavior" here might be something akin to "look and feel". So, what exactly do the authors mean by "behavior"?

A second feature of programs is that the behavior of one program can be precisely reproduced by a textually entirely different program, so "program text and behavior are independent" (Samuelson et al., 1994, p. 2315).

And a third feature is that

> programs are, in fact, machines (entities that bring about useful results, i.e., behavior) that have been constructed in the medium of text (source and object code).

> The engineering designs embodied in programs could as easily be implemented in hardware as in software, and the user would be unable to distinguish between the two. —Samuelson et al. (1994, p. 2316)

A "machine constructed in text" sounds like a virtual machine. Samuelson et al. (1994, p. 2320) go on to say that "Physical machines ... produce a variety of useful behaviors, and so it is with programs." And then they add, in a footnote: "Traditional texts may tell humans how to do a task; they do not, however, perform the task." But the same is true for a program, whether written in ink on paper, engraved on a CD-ROM, entered as source code into a computer, or expressed as object code after compilation: It still takes a "physical machine" to do what the program tells it to do. A program can't be executed without some kind of machine that can *interpret* its text and that can *behave* (for example, by printing output)—in short, that can *execute* the program. So, it's not really the program by itself that can "bring about useful results", but the combination of a program and an interpreting-plus-behaving device—in short, program + executor. And it is that combination, when turned on and running, that produces the virtual machine. The physical machine translates or associates parts of the program text (however inscribed) with "switch settings" in the computer. Once those switches are set in accordance with the program, the computer—the physical machine—will perform the task.

This is related to the question that we asked in §12.4.4.1.2.2: What tasks do programmed computers perform? In particular, what does a universal Turing machine that is implementing, say, a calculator doing? Is it adding? Or is it fetching-and-executing? Is it doing both? Or is it doing something even more specific? After all, at bottom, all that a Turing machine can do is print, erase, and move. More sophisticated computers can, of course, do many other things: anything that can be decomposed into those primitive actions. At bottom, however, all they're doing are forms of "printing" (including "printing" certain pixels on a screen), "erasing", and "moving". It's worth recalling our discussion of "procedural abstraction" (§7.5.3), however, the point of which is to make it easier for programmers to tell a computer what they want to do without having to break it into these smaller parts—more precisely, without having to "implement" such higher-level instructions in terms of the bottom-level instructions *more than once*:

> The motivation behind ... [programming with abstract data types, not to mention procedural abstraction] in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions *suitable to his [sic] problem area.* —(Liskov and Zilles, 1974, p. 50, my interpolated brackets and italics)

We'll return to this idea in Chapter 14.

The point in the previous quotation about the indistinguishability of hardware and software is, of course, the point about the relation of a Turing machine to a universal Turing machine, which is a separate issue from the question about whether a program is a machine. This is clarified a bit later:

> Not only do consumers not value the text that programmers write (source code), they do not value any form of program text, not even object code. Computer sci-

> ence has long observed that software and hardware are interchangeable: Any behavior that can be accomplished with one can also be accomplished with the other. In principle, then, it would be possible for a puchaser of, say, a word processor to get a piece of electronic hardware that plugged into her computer, instead of a disk containing object code. When using that word processor, the purchaser would be unable to determine whether she was using the software or the hardware implementation. —Samuelson et al. (1994, p. 2319)

This is the point about universal Turing machines: They can simulate any (other) Turing machine, in the sense that the behavior—better: the look and feel—of a dedicated Turing machine would be indistinguishable from that of a universal Turing machine that simulated it.

Question for the Reader:
In Chapter 19, we will discuss Turing's "imitation-game" test for distinguishing a computer from a human. Could there be a Turing-test–like imitation game to distinguish between a single-purpose Turing machine and a universal Turing machine that is simulating it?

Related to this, "PolR" (2009, §"A Universal Algorithm") notes that if

> I am sued for infringement on a software patent[,] I may try defending myself arguing I did not implement the method that is covered by the patent. I implemented the lambda-calculus algorithm which is not covered by the patent. And even if it did, lambda-calculus is old. There is prior art. What happens to software patents then?
>
> Anyone that argues in favor of method patents on software must be aware that the current state of technology permits this possibility.

Samuelson et al. (1994, §§1.1.1 and 1.1.3) remark that no one can read the object code on a CD-ROM, which marks a difference between software and, say, the script of a play. But that may be the wrong analogy. What about a movie on a DVD? We can't read the DVD any more than we can read a CD-ROM (in both cases, we need a machine to interpret them, to convert them to a humanly perceivable form). This is true even in the non-digital world: We don't typically look at the individual frames of a movie on film (except, of course, when we're editing it). But when we're watching it as it is intended to be watched, we need those frames to be projected at a certain speed.

> All this stands in sharp contrast to traditional literary works which are valued because of their expression .... Programs have almost no value to users as texts. Rather, their value lies in behavior. —Samuelson et al. (1994, p. 2319)

Yes, but this is a misleading analogy. A better analogy is not between, say, a program and a novel, but between a program and musical score or a play script, and between a program's behavior and a performance of that music or play. Only musicians and literary or theatrical people would be interested in the score or the script (just as only programmers would be interested in a program text). Audiences are more interested in the performance (in all cases).

There is one major difference between programs and behavior on the one hand and scores (or scripts) and performances on the other: The very same play or score will be performed (interpreted) very differently by different performers. But the very same program should be executed identically by different computers. That, however, may have more to do with the amount of detail in the instructions than with the nature of the interpreters. (Recall our discussion about recipes in §10.4.2.)

> While conceiving of programs as texts is not incorrect, it is seriously incomplete. A crucially important characteristic of programs is that they behave; programs exist to make computers perform tasks. —Samuelson et al. (1994, p. 2316)

I agree with the first sentence. For reasons cited above, I disagree with the claim that the program *itself* behaves. But I agree with the final claim: It is the *computer* that behaves, that does something, that "performs a task". And it does so in virtue of its program. But the program *by itself* is static, not dynamic.

> Behavior is not a secondary by-product of a program, but rather an essential part of what programs are. To put the point starkly: No one would want to buy a program that did not behave, i.e., that did nothing, no matter how elegant the source code "prose" expressing that nothing. —Samuelson et al. (1994, p. 2317)

But, as we saw in §7.4, there can certainly be programs that have no output. Indeed, they say:

> Hence, every sensible program behaves. This is true even though the program has neither a user interface nor any other behavior evident to the user.
> —Samuelson et al. (1994, p. 2317)

In a footnote, they define "user interface" as "the means by which the user and the program interact to produce the behavior that will accomplish the user's tasks." So, I suspect that Samuelson et al. are using 'program' not to refer to text but to a product intended to be installed in a computer and executed. This becomes clearer later, when they point out that:

> People pay substantial sums of money for a program not because they have any intrinsic interest in what its text says, but because they value what it does … (its behavior). The primary proof that consumers buy behavior, rather than text, is that in acquiring a program, they almost never get a readable instance of the program text. —Samuelson et al. (1994, p. 2318)

A "sensible" program (that is, the program-as-product that I described above) with no user interface might be akin to a Turing machine that simply computes with no input other than what is pre-printed on its tape. Having a user interface would seem to turn a "sensible" program into an interactive program or an *o*-machine (as we discussed in Chapter 11).

> Two programs with different texts [such as Microsoft's Excel and Apple's Numbers, to use a contemporary example] can have completely equivalent behavior. A

> second comer can develop a program having idential behavior, but completely different text through . . . "black box" testing. This involves having a programmer run the program through a variety of situations, with different inputs, making careful notes about its behavior. A second programmer can use this description to develop a new program that produces the specified behavior (i.e., functionally identical to the first program) without having access to the text of the first program, let alone copying anything from it. A skilled programmer can, in other words, copy the behavior of a program exactly, without appropriating *any* of its text. —Samuelson et al. (1994, pp. 2317–2018, my interpolated brackets)

Let's grant that Excel and Numbers don't have exactly the same input-output behavior or even look-and-feel. But black-box testing could, indeed—in theory, at least—allow two completely textually different programs to have exactly the same input-output behavior (and look-and-feel), for the simple, mathematical reason that there can be more than one algorithm for implementing a function.

Note, however, that, in practice, such black-box testing is really akin to the kind of inductive inference that underlies trial-and-error machines: Unless the second programmer has a complete input-output specification, that programmer is limited to reconstructing potentially infinite behavior on the basis of a finite number of examples.

> The independence of text and behavior is one important respect in which programs differ from other copyrighted works. Imagine trying to create two pieces of music that have different notes, but that sound indistinguishable. Imagine trying to create two plays with different dialogue and characters, but that appear indistinguishable to the audience. Yet, two programs with different texts can be indistinguishable to users. —Samuelson et al. (1994, p. 2318)

This is interesting. If true, it suggests that the analogies between score and performance, or between script and play, etc. (see §14.3 for more such analogies), might not be as close as I have suggested. But is it true? First, consider a play and its script. Suppose that I write a version of *Hamlet* that incorporates all of Shakespeare's lines but adds material that is specifically designed not to be uttered during a performance (perhaps extra characters who are always offstage and silent). That would seem to be a counterexample to Samuelson et al. Second, consider two different algorithms that both compute GCDs. Is it not conceivable that, when the operations in each are decomposed to the primitive operations of a Turing machine or the basic recursive functions, the algorithms would be identical? Or consider a musical score for, say, a choral work for four voices. I could produce two different written scores simply by re-ordering the parts (or, by writing them in different musical notations). Yet their performances would not differ.

All of these quibbles notwithstanding, Samuelson et al. (1994, p. 2323) do make an interesting point about the appropriateness of copyright for legal protection of programs:

> Program text is, thus, like steel and plastic, a medium in which other works can be created. A device built in the medium of steel or plastic, if sufficiently novel, is patentable; an original sculpture built of steel or plastic is copyrightable. In these

> cases, we understand quite well that the medium in which the work is made does not determine the character of the creation. The same principle applies to software. The legal character of a work created in the medium of software should no more be determined by the medium in which it was created than would be a work made of steel or plastic. In this respect, it makes no more sense to talk about copyrighting programs than to talk about copyrighting plastic or steel; it confuses the *medium of creation* and the *artifact* created. In the case of software, the artifact created is some form of innovative behavior, whether utilitarian or fanciful.

This suggests that patent is more appropriate than copyright. But what about a program whose sole purpose is to produce an image on a screen, such as a painting or a film? Wouldn't copyright be more appropriate then? Still, the point remains that it's the behavior of the program (when executed by a machine) that is more important than the text.

> We suggest that programs should be viewed as virtual machines and that this has interesting consequences for the proper form of protection.
> —Samuelson et al. (1994, p. 2324)

Unfortunately, they don't define what they mean by "virtual machine". But if they mean that a virtual machine is a Turing machine being simulated on a universal Turing machine (as we suggested in §13.6), and if machines are patentable, then it would seem that programs understood as virtual machines should be patentable, not copyrightable.

This is the basis of their argument against the appropriateness of copyright law (Samuelson et al., 1994, p. 2350):

1. "computer programs are machines whose medium of construction is text"
2. "Copyright law does not protect the behavior of physical machines (nor their internal construction)"
3. ∴ "program behavior ... is unprotectable by copyright law on account of its functionality" (p. 2351).

But far from arguing that programs, because not copyrightable, should be patentable, they also think that patentability is inappropriate:

> The predominantly functional nature of program behavior and other industrial design aspects of programs precludes copyright protection, while the incremental nature of innovation in software largely precludes patent protection.
> —Samuelson et al. (1994, p. 2333)

They offer two arguments against patentability. Here is the first (Samuelson et al., 1994, p. 2346):

1. "Patent law requires an inventive advance over the prior art"
2. But the innovations in "functional program behavior, user interfaces, and the industrial design of programs ... are typically of an incremental sort."

3. ∴ Programs do not fall under patent law.

A similar argument applies to chip design: A form of legal protection other than patent and copyright was needed, recognized, and implemented; as a result, "Congress passed the Semiconductor Chip Protection Act of 1984".

And here is their second argument against patentability (Samuelson et al., 1994, p. 2345):

1. Patents are given for "methods of achieving results",

2. Patents are not given "for results themselves".

3. "It is ... possible to produce functionally indistinguishable program behaviors through use of more than one method."

4. ∴ A patent could be given for one method of producing a result, but that would not "prevent the use of another method".

5. If it is the result, not the method, that is the "principal source of value" of a program, then the patent on the one method would not protect the result produced by the other method

On Samuelson et al.'s suggestion that it is the virtual machine that should be legally protected (whether by copyright, patent, or something sui generis), compare this:

> For example consider this sentence from In re Alappat:[6]
>
> > We have held that such programming creates a new machine, because a general purpose computer in effect becomes a special purpose computer once it is programmed to perform particular functions pursuant to instructions from program software.
>
> In a single sentence the court tosses out the fundamental principle that makes it possible to build and sell digital computers. You don't need to create a new machine every time you perform a different computation; a single machine has the capability to perform all computations. This is what universal Turing machines are doing. —"PolR" (2009)

But the "new machine" mentioned in the ruling is a virtual machine.

[6] WJR footnote: See http://digital-law-online.info/cases/31PQ2D1545.htm

Further Reading:
Samuelson 1988 is an excellent overview of the copyright-vs.-patent dispute, arguing in favor of patent protection for software.

Samuelson 1991 notes that "Six characteristics of digital media seem likely to bring about significant changes in the law": ease of replication, ease of transmission and multiple use, plasticity (related to Moor's §12.4.4.1 notion of software's changeability), equivalence of works in digital form (referring to the fact that, when digitized, different kinds of copyrightable works all get implemented in the same medium), compactness (digitized works take up little space, hence are more prone to theft), and nonlinearity (due to the availability of hyperlinks, for instance).

Samuelson 2003 discusses "Attempting to stretch existing laws to address previously unforeseen technological issues."

"Microsoft argues that neither the intangible sequence of ones and zeros of the object code, nor the master disks onto which the object code has been loaded, should be considered a component of a patented invention .... Only when object code has actually been installed on a ... computer does it become a physical component of a physical device ..." (Samuelson, 2007b, pp.15–16).

13.8 Allen Newell's Analysis

We'll close with another person's analysis of these problems. But it is not just any one person; it is a well-respected computer scientist. And it is not just any computer scientist; it is one of the founders of the fields of AI and of cognitive science, and a winner of the Turing Award—Allen Newell—whom we have already mentioned in several previous chapters. In a paper written for a law journal, Newell (1986) argues that the "conceptual models" of algorithms and their uses are "broken", that is, that they are "inadequate" for discussions involving patents (and copyrights).

Newell's definition of 'algorithm' is very informal, but "perfectly reasonable, not arcane; we can live with it": An algorithm is "an unambiguous specification of a conditional sequence of steps or operations for solving a class of problems". This is obviously an oversimplification. But it might suffice for Newell's purposes.

Further Reading:
In addition to our discussion in §7.4, see Chisum 1986, "The Definition of an Algorithm", §B, pp. 974–977, for other definitions. For some of these, see http://www.cse.buffalo.edu/~rapaport/584/c-vs-pat.html, item 3.

Algorithms are mathematical objects, but there are non-mathematical algorithms, so, Newell points out, you cannot make a decision on the patentability of algorithms on the basis of a distinction between mathematical processes and non-mathematical ones.

Moreover, "humans think *by means of* algorithms" (Newell, 1986, p. 1025, my italics). This is a very strong version of a thesis in the philosophy of mind called "computationalism". It is, in fact, one reason why Newell is so famous: Newell and Nobel-prize winner Herbert Simon wrote one of the first AI programs, which they interpreted as showing that "humans think by means of algorithms" (Newell et al., 1958). Note that Newell doesn't say what *kind* of algorithms he has in mind here,

nor does he say whether they are Turing-machine–equivalent. His point is that some algorithms are sequences of *mental* steps; so, you can't make patentability decisions based on any alleged distinction between mental processes and algorithms.

He also points out that you cannot patent natural laws or mathematical truths (for instance, you cannot patent the integers, or addition). Are algorithms natural laws? In §§3.9.3 and 10.3, we saw that Stuart C. Shapiro and many others think that they are, if not laws, then at least entities that are part of the natural world. Are they mathematical truths? They are certainly among the items that some mathematicians study. In either case, algorithms would not be patentable. On the other hand, they *are* "applicable", hence patentable. Arguably, *all* of CS concerns application, and algorithms are certainly part of CS's theoretical foundations.

In addition, Newell points out, it is difficult to distinguish an algorithm (an abstract program) from a computer program (which adds implementation details that are not, strictly speaking, part of the algorithm), because there are only *degrees* of abstraction. For example, you could have a really good interpreter that would take an algorithm (or even a functional—input-output—specification) and execute it directly. In a sense, this is a holy grail of programming: to talk to a computer in English and have it understand you without your having to program it to understand you. (The *Star Trek* computer is the model for this; devices such as Siri, Alexa, et al., are baby steps in that direction.)

But, contrary to his own definition, Newell points out that algorithms need only be "specifications that determine the behavior of a system", not necessarily *sequences* of steps. Programs in Prolog or Lisp, for instance, are *sets*, not sequences, of statements. So, you can't identify whether something is an algorithm just by looking at it.

The bottom line is that there are two intellectual tasks: A *computer-scientific and philosophical task* is to devise good models ("ontologies") of algorithms and other computer-scientific items. A *legal task* is to devise good legal structures to protect these computational items.

Further Reading:
The ontological task is investigated by Amnon Eden (2007); Raymond Turner and Eden (2007); and Brian Cantwell Smith (1996, 2002). On ontologies more generally, see the work by philosopher Barry Smith and his colleagues at http://ontology.buffalo.edu/smith/, which includes the ontology of software vs. hardware by Duncan (2017), cited at the end of §12.4.6.

I'll close this chapter with a quotation from Newell (1986, p. 1035, my boldface):

> I think fixing the models is an important intellectual task. It will be difficult. The concepts that are being jumbled together—methods, processes, mental steps, abstraction, algorithms, procedures, determinism—ramify throughout the social and economic fabric .... The task is to get ... new models. There is a fertile field to be plowed here, to understand what models might work for the law. It is a job for lawyers and, importantly, theoretical computer scientists. **It could also use some philosophers of computation, if we could ever grow some.**

Readers of this book, take note!

Chapter 14

What Is Implementation?

Version of 7 August 2019;[1] DRAFT © 2004–2019 by William J. Rapaport

> "I wish to God these calculations had been executed by steam!"
> —Charles Babbage (1821), quoted in Swade 1993, p. 86.

> What . . . [Howard H. Aiken, who built the Harvard Mark I computer] had in mind . . . was the construction of an electromechanical machine, but the plan he outlined was not restricted to any specific type of mechanism; it embraced a broad coordination of components that could be resolved by various constructive mediums.
> —George C. Chase (1980, p. 226, my interpolated brackets)

> [W]hy wasn't Mark I an electronic device? Again, the answer is money. It was going to take a lot of money. Thousands and thousands of parts! It was very clear that this thing could be done with electronic parts, too, using the techniques of the digital counters that had been made with vacuum tubes, just a few years before I started, for counting cosmic rays. But what it comes down to is this: if [the] Monroe [Calculating Machine Co.] had decided to pay the bill, this thing would have been made out of mechanical parts. If RCA had been interested, it might have been electronic. And it was made out of tabulating machine parts because IBM was willing to pay the bill.
> —Howard H. Aiken,
> quoted by I. Bernard Cohen in Chase 1980, p. 200, my interpolated brackets

> . . . Darwin discovered the fundamental *algorithm* of evolution by natural selection, an abstract structure that can be implemented or "realized" in different materials or media. —Daniel C. Dennett (2017, p. 138)

[1] Portions of this chapter are adapted from Rapaport 1999, 2005b.

14.1 Readings:

1. Required:
   (a) Liskov, Barbara; & Zilles, Stephen (1974), "Programming with Abstract Data Types", *ACM SIGPLAN Notices* 9(4) (April): 50–59, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.3043&rep=rep1&type=pdf
   (b) Chalmers, David J. (1993), "A Computational Foundation for the Study of Cognition", *Journal of Cognitive Science* (South Korea) 12(4) (October-December 2011): 323–357, http://j-cs.org/gnuboard/bbs/board.php?bo_table=__vol012i4&wr_id=2
      i. Read §§1–2; skim the rest.
         - §2 of this paper was published (in slightly different form) as Chalmers 1995.
      ii. Originally written in 1993, the 2011 version was accompanied by commentaries (including Egan 2012; Rescorla 2012b; Scheutz 2012; Shagrir 2012a) and a reply (Chalmers, 2012).
   (c) Rapaport, William J. (1999), "Implementation Is Semantic Interpretation", *The Monist* 82(1): 109–130.
      - This article was based on a longer essay (which it alludes to as "in preparation"), which has since been published as Rapaport 2005b
2. Strongly Recommended:
   (a) Putnam, Hilary (1988), Appendix, *Representation and Reality* (Cambridge, MA: MIT Press): 121–125.
      i. "*Theorem.* Every ordinary open system is a realization of every abstract finite automaton."
      ii. Putnam's argument for this "theorem" is related to Searle's (1990) argument about the wall behind me implementing Wordstar (which we discussed in §9.5.6), but it is much more detailed.
   (b) Chalmers, David J. (1993), "Does a Rock Implement Every Finite-State Automaton?", *Synthese* 108 (1996): 309–333, http://consc.net/papers/rock.html.
      i. This is a reply to Putnam's argument, above.
      ii. Chalmers corrects "an error in my arguments" in this paper in Chalmers 2012, pp. 236–238.
3. Recommended:
   (a) Rescorla, Michael (2012), "Are Computational Transitions Sensitive to Semantics?", *Australian Journal of Philosophy* 90(4) (December): 703–721, http://www.philosophy.ucsb.edu/docs/faculty/papers/formal.pdf
   (b) Rescorla, Michael (2013), "Against Structuralist Theories of Computational Implementation", *British Journal for the Philosophy of Science* 64(4) (December): 681–707, http://philosophy.ucsb.edu/docs/faculty/papers/against.pdf
   (c) Rescorla, Michael (2014), "A Theory of Computational Implementation", *Synthese* 191: 1277–1307, http://philosophy.ucsb.edu/docs/faculty/papers/implementationfinal.pdf

14.2 Introduction

> On the one hand, we have a very elegant set of mathematical results ranging from Turing's theorem to Church's thesis to recursive function theory. On the other hand, we have an impressive set of electronic devices which we use every day. Since we have such advanced mathematics and such good electronics, we assume that somehow *somebody must have done the basic philosophical work of connecting the mathematics to the electronics. But as far as I can tell that is not the case.* On the contrary, we are in a peculiar situation where *there is little theoretical agreement among the practitioners on such absolutely fundamental questions as*, What exactly is a digital computer? What exactly is a symbol? What exactly is a computational process? *Under what physical conditions exactly are two systems implementing the same program?* —Searle (1990, p. 25, my italics)

One concept that has repeatedly cropped up in our discussions is that of "implementation":

- In §3.13.1, we read that Loui (1987, p. 176) said that CS studies (among other things) the 'implementations of ... algorithms in hardware and in software".
- In §7.5.5, we discussed the Implementation Insight: that the binary-representation, Turing-machine, and structured-programming insights can all be physically implemented.
- In Chapter 8, we cited several implementations of Turing machines: using pebbles and toilet paper, railroad and subway trains, the Game of Life, and even business cards!
- In §9.4.1, we investigated the extent to which (physical) computers were implementations of (abstract) Turing machines.
- In §9.5.6, we saw Searle (1990) saying that "the wall behind my back is right now implementing the Wordstar program", and we talked about "physical implementations of Turing machines" and "human cognition ... implemented by neuron firings".
- In §10.4.2, we saw that "to implement a plan is to copy an abstract design into reality".
- In Chapter 12, we talked about functions being implemented by algorithms, which can be implemented by programs written in high-level computer programming languages, which, in turn, can be implemented by assembly-language programs, which, in their turn, can be implemented by machine-language programs, which can, finally, be implemented in hardware. And besides such "chained" implementations of implementations, we looked at the idea of there being "multiple" implementations of a single item.
- In Chapter 13, we looked at how copyright only applies to implementations in the medium of language of abstract ideas.

- And computer programmers talk about data structures in a particular programming language implementing abstract data types (Liskov and Zilles 1974; Aho et al. 1983, §1.2). We'll go into more detail on this in §14.2.3.

So, what is an implementation?

14.2.1 Implementation vs. Abstraction

Let's begin by contrasting "implementation" with "abstraction". Abstractions are usually thought of as being non-physical; the opposite is usually said to be something that is "concrete". But, more generally, something is abstract if it *omits some details.*

> What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details.
> —Liskov and Zilles (1974, p. 51)

And, precisely because abstractions omit details, they are also *more general* than something that has those details filled in. The more details that are omitted, the more abstract (and the more general) something is. For example, "geology" is (literally) the study of the *Earth* and its physical structure, and "selenology" is (literally) the study of the *Moon* and *its* physical structure (Clarke, 1951, p. 42). If you *abstract* away from the particular heavenly body that is being studied (Earth or Moon), the result would be a more general science that studies the physical structure of a(n unspecified) heavenly body (even if it might still be *called* 'geology').

When you fill in some of the details that were omitted in an abstraction, you get an implementation of it. Indeed, the word 'implement' comes from a Latin word meaning "to fill up, to complete".[2] An implementation does not have to be "concrete"; it can itself be abstract if it doesn't fill in *all* the details. As Rosenblueth and Wiener (1945, p. 320) put it, implementation (what they call 'embodiment') is the "converse" of abstraction. But I think that it is better to say that an implementation and the abstraction that it implements lie along a spectrum.

Rosenblueth and Wiener observe that, in order to understand some part of the complex universe, scientists replace it "by a model of similar but simpler structure" (p. 316)—this is the technique of *abstraction*. Thus, one mark of being an abstraction is to be simpler than what it's an abstraction of; what it's an abstraction of (in this case, a part of the universe) will have "extra" features. These extra features might be quite important ones that are being ignored merely temporarily or for the sake of expediency, or they might be "noise"—irrelevant details, or details that arise from the *medium* of implementation. In such cases, the extra features that are not in the abstraction are often referred to as "implementation-dependent details". For example, Rescorla (2014b, §2, p. 1280) says: "A physical system usually has many notable properties besides those encoded by our computational model: colors, shapes, sizes, and so on."

There are, according to Rosenblueth and Wiener, two kinds of models: formal and material, both of which are abstractions (p. 316). Formal models are like mathematical models: They are formal symbol systems expressed in formal languages and understood in terms of their *syntax*. Recall from §9.5.1 that syntax is usually considered to

[2] For more on the etymology of 'implement', see Rapaport 1999, §2, pp. 110–111; and §4, pp. 115–116.

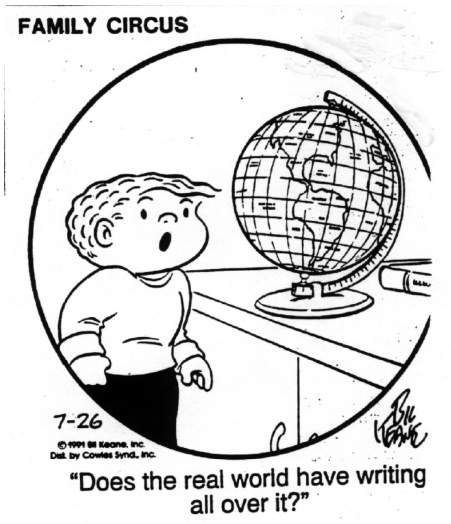


Figure 14.1: ©1991 Bil Keane, Inc.

be the study of the properties of, and the relations among, the symbols of a language; for example, the grammar of a language is its syntax. Let's call a formal system that is understood in terms of its syntax a "syntactic domain".

Material—that is, *physical*—models, however, are more like scale models (p. 317) than like symbol systems. Because such models omit some details (for example, scale models are smaller and usually made of different materials than what they are models of), they are "abstract", even though they are "concrete", or physical. But a material model can also be "more elaborate" than that which it models (p. 318). This suggests that "implementation-dependent details"—that is, parts of the model that are *not* (or are not *intended* to be) representations of the complex system—are ignored. For instance, the physical matter that the model is made of, or imperfections in it, would be ignored: One does not infer from a plastic scale model of the solar system that the solar system is made of plastic, nor from a globe that the Earth has writing on it (as in Figure 14.1).

Typically, implementations are *physical* "realizations" or "embodiments" of *non*-physical "abstractions". That is, implementation is typically understood as a relation between an abstract specification and a concrete, physical entity or process. But a real, *physical* airplane could be considered to be an implementation of a *physical* scale model "of" it. the former is complete in all details—it really flies and carries passengers, but the scale model, even though physical, lacks these abilities.

14.2.2 Implementations as Role Players

There can be multiple, different implementations of a single abstraction: Some merely fill in more or different details, but others might do so using different (usually physical) "stuff"—different media. For example, in the fairy tale, the three little pigs can be

thought of as having used a single abstract blueprint to build three different versions of the "same" house out of three different materials (in three different media): straw, sticks, and bricks.

So, abstractions can be "multiply realized"—implemented in more than one way, just as many different actors can play the same role in different productions (implementations!) of the same play. Hamlet is a role; Richard Burton occupied that role in the 1964 Broadway production of *Hamlet*, and Laurence Olivier occupied it in the 1948 film. Alternatively, we could say that Burton and Olivier "implemented" Hamlet in the "medium" of human being (and a drawing of Hamlet implements Hamlet in the medium of an animated cartoon version of the play).

The implementation-abstraction distinction is also mirrored in the "occupant"-"role" distinction made in functionalist theories of the mind (Lycan, 1990, p. 77). According to those theories, mental states and processes are "functional roles" played—or implemented—by neuron firings in brains (or perhaps computational states and processes in AI computer programs; we'll come back to this idea in §20.3).

And mathematical "structuralists" have argued that numbers are not "things" (existing in some Platonic heaven somewhere), but are more like "roles" in a mathematical structure (defined, say, by Peano's axioms) that can be "played" by many different things, such as different sets, Arabic numerals, etc. (Benacerraf, 1965). Recall our brief mention in §9.5.6 of structuralism: The natural numbers can be considered a "role" that can be "played" by any "countably infinite (recursive) set ... arranged to form an ω-sequence" (Swoyer, 1991, p. 504, note 26). In the rest of that passage (my interpolated brackets), Swoyer goes on to say that "a concrete realization [that is, an implementation of the natural numbers] would be obtained by adding a domain of individuals and assigning them as extensions [that is, as semantic interpretations] to the properties and relations in the structure" (my interpolated brackets). In other words, an implementation of the natural numbers is the same as a semantic interpretation of them.

14.2.3 Abstract Data Types

Computer scientists also use the term 'implementation' to refer to the relation between an *abstract* data type and its "implementation" or "representation" by an *abstract* data structure in a programming language. Although it is not entirely abstract, a data structure is also not entirely physical: It is part of the software, not the hardware. Moreover, one abstract data type could even be implemented in a different *abstract* data type.

What is an "abstract data type"?

> An *abstract data type* defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type.
> —Liskov and Zilles (1974, p. 51)

So, an abstract data type is a kind of abstract noun—an indefinite description of the form: "an entity that can perform actions $A_1, \ldots, A_k$, and that actions $A_{k+1}, \ldots, A_n$ can be performed on", where the A_i are new, abstract "verbs". The actual entities that satisfy such a description are implementations of the abstract data type.

Let me a bit more precise. Some programming languages, such as Lisp, do not have stacks as a built-in data structure. So, a programmer who wants to write a program that requires the use of stacks must find a substitute. In Lisp, whose only built-in data structure is a linked list, the stack would have to be built out of a linked list: Stacks in Lisp can be implemented by linked lists. Here's how:

First, a stack is a particular kind of abstract data type, often thought of as consisting of a set of items structured like an ordinary, physical stack of cafeteria trays: New items are added to the stack by "pushing" them on "top", and items can be removed from the stack only by "popping" them from the top. Thus, to define a stack, one needs (i) a way of referring to its top and (ii) operations for pushing new items onto the top and for popping items off the top. That, more or less (mostly less, since this is informal), is a stack defined as an abstract data type.

Second, a linked list ('list', for short) is itself an abstract data type. It is a sequence of items whose three basic operations are:

1. $first(l)$, which returns the first element on the list l,

2. *rest*(l), which returns a list consisting of all the original items except the first,

   and

3. *make-list*(i,l), which recursively constructs a list by putting item i at the beginning of list l.

Finally, a stack s can be implemented as a list l, where $top(s) := first(l)$, *push*$(i,s) :=$ *make-list*(i,l), and $pop(s)$ returns $top(s)$ and redefines the list to be *rest*(l).

As another example of an "abstract implementation", consider a top-down-design, stepwise-refinement (that is, a recursive development) of a computer program (see §6.5.3): Each level (each refinement) is an abstract implementation of the previous, higher-level one. Eachy of the more detailed implementation levels is less abstract than the previous one. A "concrete implementation" would be an implementation in a physical medium.

Question for the Reader:
Could this be related to what Colburn might have had in mind when he talked about a "concrete abstraction"? (Recall our discussion in §12.4.6.)

14.2.4 The Structure of Implementation

As we have seen, abstractions omit details and can be thought of as roles. Implementations fill in some of those details and can be thought of as things that play the role specified by the abstraction. Some implementations may add details ("implementation-dependent details") that do not belong to the abstraction. For example, Hamlet's age is not specified in Shakespeare's play (though he was supposed to be a college student), but Burton was about 39 when he played the role, and Olivier was about 41; those are

implementation-dependent details. Furthermore, there can be multiple implementations of a given abstraction, which differ in the "stuff" that the implementation is made of. (These are also implementation-dependent details.)

To sum up, implementation is best thought of as a three-place relation:

> *I* is an implementation *in* medium *M of* abstraction *A*.

And there are two fundamental principles concerning this relation:

Implementation Principle I: For every implementation *I*, there is an abstraction *A* such that *I* implements *A*.

Implementation Principle II: For every abstraction, there can be more than one implementation of it.

Principle I actually follows from the nature of the three-place relation; Principle II is a generalization of the principle of "multiple realizability" (Bickle, 2019). (We will return to these two principles in §19.6.2.2.)

In the next two sections, we will look at two theories that spell out more of the details about the nature of implementation. One will use the relation between syntax and semantics to illuminate implementation. The other, due to David Chalmers, was designed to reply to Searle (1990).

14.3 Implementation as Semantic Interpretation

14.3.1 What Kind of Relation Is Implementation?

One main point of the previous section is that not all examples of implementation concern the implementation of something *abstract* by something *concrete*. As we have just seen, sometimes one abstract thing can implement another abstract thing, and one concrete thing can implement another concrete thing. What we need is a more general notion. There are several candidates:

- individuation:

  > This is the relation between the lowest level of a genus-species tree (such as "dog" or "human") and individual dogs or humans: For example, my cat Bella "individuates" the species *Felis catus*). Individuation seems to be a kind of implementation: We could say that Bella is an *implementation* of *Felix catus*. But not all cases of implementation are individuations.

- instantiation:

  > This is the relation between a specific instance of something and the kind of thing that it is: For example, the specific redness of my notebook cover is an instance of the color "red"). Instantiation seems to be a kind of implementation: We could say that the specific instance of red that is my notebook's color is an *implementation* of the (abstract) color "red". But not all implementations are instantiations.

- exemplification:

  > This is the relation between a (physical) object and a property that it has: For example, Bertrand Russell exemplifies the property of being a philosopher). Exemplification seems to be a kind of implementation: We could say that Bertrand Russell is an *implementation* of a philosopher. But not all implementations are exemplifications.

- reduction:

  > This is the relation between a higher-level object and the lower-level objects that it is made of: For example, water is reducible to a molecule consisting of two atoms of hydrogen and one atom of oxygen, or, perhaps, the emotion of anger is reducible to a certain combination of neuron firings). This kind of reduction[3] seems to be a kind of implementation: We could say that water is *implemented* by H_2O, or that my anger is *implemented* by certain neuron firings in my brain and nervous system. But not all implementations are reductions.

Each of these may be implementations, but not vice versa. In other words, implementation is a more general notion than any of these. But all of them can be viewed as semantic interpretations (Rapaport, 1999, 2005b).

Let *A* be an "abstraction" (that is, something that spells out a role to be played or a generalization of the notion of an abstract data type). And let *M* be any abstract or concrete "medium". Then we can say that

> *I* is an implementation in medium *M* of *A*

iff

> *I* is a semantic interpretation in semantic domain *M* of syntactic domain *A*

Implementation *I* could be either an abstraction itself or something concrete, depending on *M*.

And, as we have seen, there could be a sequence of implementations (or what Brian Cantwell Smith (1987) calls a "correspondence continuum"; for discussion, see Rapaport 1995, §2): A stack can be implemented by a linked list, which, in turn, could be implemented in the programming language Pascal, which, in turn, could be implemented (that is, compiled into) some machine language *L*, which, in turn, could be implemented on my Mac computer. (We saw this same phenomenon in another, though related, situation back in §9.2. Sloman 1998, §2, p. 2 makes the same point about what he calls "implementation layers".) But it could also be more than a mere sequence of implementations, because there could be a *tree* of implementations, much as in the previous chapter's Figure 13.2: The very same linked list could be implemented in Java, instead of Pascal, and the Java or Pascal program could be implemented in some other machine language on some other kind of computer.

[3] There are others; see Dennett 1995, Ch. 3, §5 for a useful discussion.

The ideas that abstractions can implement other abstractions and that there can be "continua" of implementations is a consequence of what the philosopher William G. Lycan (1990, p. 78), refers to as the "relativity" of implementation:

> ... "software"/"hardware" talk encourages the idea of a bipartite Nature, divided into two levels, roughly the physiochemical and the (supervenient) "functional" or higher-organizational—as against reality, which is a multiple *hierarchy* of levels of nature .... See Nature as hierarchically organized in this way, and the "function"/"structure" distinction *goes relative*: something is a role as opposed to an occupant, a functional state as opposed to a realizer, or vice versa, only *modulo* a designated level of nature.

14.3.2 What Is Semantic Interpretation?

14.3.2.1 Formal Systems

Let's begin with the concept of a "formal system". These are sometimes called "symbol systems", "theories" (understood as a set of sentences), or "formal languages". A formal system consists of:

1. primitive (or atomic) "symbols" (sometimes called "tokens" or "markers", which can be thought of as being like the playing pieces in a board game such as Monopoly)

   > These are assumed to have no interpretation or meaning, hence my use of the term 'marker', rather than 'symbol', which many writers use to mean a marker plus its meaning (as when we say, "a wedding ring is a symbol for marriage" (Levesque, 2017, p. 108). The racecar token in Monopoly isn't interpreted as a racecar in the game; it's just a token that happens to be racecar shaped, so as to distinguish it from the token that is top-hat shaped. (And the top-hat token isn't interpreted as a top hat in the game: Even if you think that it makes sense for a racecar to travel around the Monopoly board, it makes no sense for a top hat to do so!)
   >
   > Examples of such atomic markers include the letters of an alphabet, (some of) the vocabulary of a language, (possibly) neuron firings, or even states of a computation.

2. (recursive) rules for forming new (complex, or molecular) markers, sometimes called 'well-formed formulas' (wffs)—that is, grammatically correct formulas—from "old" markers (that is, from previously formed markers), beginning with the atomic markers as the basic "building blocks".

   > These rules might be spelling rules (if the atomic markers are alphabet letters), or grammar rules (if the atomic markers are words), or bundles of synchronous neuron firings (if the atomic markers are single neuron firings).

> The molecular markers can be thought of as "strings" (that is, sequences of atomic markers), or words (if the atomic markers are letters), or sentences (if the atomic markers are words).

3. a "distinguished" (that is, singled-out) subset of wffs

   > These are usually called 'axioms'. But having axioms is optional. If English is considered as a formal system (Montague, 1970), it doesn't need axioms. But if geometry is considered as a formal system, it usually has axioms.

4. recursive rules (called 'rules of inference' or 'transformation rules') for forming ("proving") new wffs (called 'theorems') from old ones (usually, but not always, beginning with the axioms).

Digression: Formal Systems and Turing Machines.
Turing machines can be viewed as (implementations of) formal systems: Roughly, (1) the atomic markers correspond to the '0's and '1's of a Turing machine, (2) the wffs correspond to the sequences of '0's and '1's on the tape during a computation, (3) the axioms correspond to the initial string of '0's and '1's on the tape, and (4) the recursive rules of inference correspond to the instructions for the Turing machine. See Suber 1997a for details. Saul Kripke (2013) has also advocated for this point of view:

> [A] computation is a special form of mathematical argument. One is given a set of instructions, and the steps in the computation are supposed to ... follow deductively ... from the instructions as given. *So a computation is just another mathematical deduction* ... (p. 80).

14.3.2.2 Syntax

The "syntax" of such a system is the study of the *properties of* the markers of the system and the *relations among* them (but *not* any relations *between* the markers and anything outside of the system). Among these (internal) relations are the "grammatical" relations, which specify which strings of markers are well formed (according to the rules of grammar), and the "proof-theoretic" (or "logical") relations, which specify which sequences of wffs are proofs of theorems (that is, which wffs are derivable by means of the rules of inference).

Here is an analogy: Consider a new kind of toy system, consisting of Lego-like blocks that can be used to construct Transformer monsters. (This wouldn't be a very practical real toy, because things made out of Legos tend to fall apart. That's why this is a thought experiment, not a real one!) The basic Lego blocks are the primitive markers of this system. Transformer monsters that are built from Legos are the wffs of the system. And the sequences of moves that transform the monsters into trucks (and back again) are the proofs of theorems.

Further Reading:
Real examples of formal systems include propositional logic, first-order logic, Douglas Hofstadter's "MIU" system (Hofstadter, 1979), Peter Suber's "S" (Suber, 1997a, 2002), and my own "mark system" $\mathcal{L}'$ (Rapaport 2017b, §2.2; see also Chapter 22). Sometimes, the entities of such systems are called 'constructive objects'; see https://www.encyclopediaofmath.org/index.php/Constructive_object. There is a nice discussion of "marks" and "mark manipulation" systems in Kearns 1997, §2, pp. 273–274.

14.3.2.3 Semantic Interpretation

An important fact about a formal system and its syntax is that there is no mention of truth, meaning, reference, or any other "semantic" notion. These are all relations between the markers of a formal system and things that our external to the system. Such *external* relations are not part of the formal system. (They are also not part of the system of things that are outside of the formal system!) We came across this idea in §8.9.1, when we discussed Hilbert's claim that geometry could be as much about tables, chairs, and beer mugs as about points, lines, and planes. Tenenbaum and Augenstein (1981, p. 1), note that "the concept of information in computer science is similar to the concepts of point, line, and plane in geometry—they are all undefined terms about which statements can be made but which cannot be explained in terms of more elementary concepts."

But sometimes we want to "understand" a formal system. There are two ways to do that (Rapaport, 1986f, 1995). One way is to understand the system in terms of *itself*—to become familiar with the system's syntax. This can be called "syntactic understanding". Another way is to understand the system in terms of *another* system that we *already* understand. This can be called "semantic understanding":

> Material models [that is, semantic interpretations] ... may assist the scientist in replacing a phenomenon in an *unfamilar* field by one in a field in which he [sic] is *more at home*.
> —Rosenblueth and Wiener (1945, p. 317, my interpolated brackets and italics)

The "semantics" of a formal system is the study of the relations *between* the markers of the system (on the one hand) and something else (on the other hand). The "something else" might be what the markers "represent", or what they "mean", or what they "refer to", or what they "name", or what they "describe". Or it might be "the world". If the formal system is a language, then semantics studies the relations between, on the one hand, the words and sentences of the language and, on the other hand, their meanings. If the formal system is a (scientific) theory, then semantics studies the relations between the markers of the theory and the world—the world that the theory is a theory *of*. (We'll come back to this theme in the next chapter, when we consider whether (some) computer programs are scientific theories.)

Semantics, in general, requires three things:

1. a syntactic domain; call it 'SYN'—typically, but not necessarily, a formal system,

2. a semantic domain; call it 'SEM'—characterized by an "ontology",

- An ontology is, roughly, a theory or description of the things in the semantic domain. It can be understood as a (syntactic!) theory of the semantic domain, in the sense that it specifies (a) the parts of the semantic domain (its members, categories, etc.) and (b) their properties and relationships (structural, as well as inferential or logical). Such an ontology is sometimes called a "model theory".

3. a semantic *interpretation* mapping from SYN to SEM. SEM is a "model" or "interpretation" of SYN; SYN is a "theory" or a "description" of SEM. Description is a semantic notion:

> Physical system P realizes/implements computational model M just in case [c]omputational model M accurately *describes* physical system P.
> —Rescorla (2014b, §2, p. 1278; my italics)

Here are several examples of semantic domains that are implementations of syntactic domains:

SYNTAX		SEMANTICS
algorithms	are implemented by	computer programs (in language L)
computer programs (in language L)	are implemented by	computational processes (on machine m)
abstract data types	are implemented by	data structures (in language L)
musical scores	are implemented by	performances (by musicians)
play scripts	are implemented by	performances (by actors)
blueprints	are implemented by	buildings (made of wood, bricks, etc.)
formal theories	are implemented by	(set-theoretic) models

Digression: Syntax, Semantics, and Puzzles:
We can illustrate the difference between syntax and semantics by means of jigsaw puzzles. The typical jigsaw puzzle that one can buy in a store consists of a (usually rectangular) piece of heavy cardboard or wood, with a picture printed on it, and which has been "jigsawed" into pieces. The object of the puzzle is to put the pieces back together again to (re-)form the picture. The pieces are usually stored in a box that has a copy of the picture on it (put together, but without the boundaries of the pieces indicated).

There are at least two distinct ways to solve the puzzle, that is, to put the pieces back together:

Syntactically:

Each piece has a distinct shape and a fragment of the original picture on it. Furthermore, the shapes of the pieces are such that they can be put together in (usually) only one way. In other words, any two pieces are completely unrelated in terms of their shape, or else they are such that they fit together to form part of the completed puzzle. A map of the US can be used as an example of this "fitting together": The boundaries (that is, the shape) of New York State and Pennsylvania fit together across New York's southern boundary and Pennsylvania's northern boundary, but New York and California are unrelated in this way. These properties (shapes and picture fragments) and relations (fitting together) constitute the syntax of the puzzle.

One way of putting the puzzle together is to pay attention only to the syntax of the pieces. In a rectangular puzzle, one strategy is to first find the outer boundary pieces, each of which has at least one straight edge, and then to fit them together to form the "frame" of the puzzle. Next, one would try to find pieces that fit either in terms of their shape or in terms of the pattern (picture fragment) on it.

This method makes no use of the completed picture as shown on the box. If that picture is understood as a semantic interpretation of the pieces, then this syntactic method of solving the puzzle makes no use of semantics.

Semantically:

But by using that semantic information, one can solve the puzzle solely by matching the patterns on the pieces with the picture on the box, and then placing the pieces together according to that external semantic information.

Of course, typically, one uses both techniques. But the point I want to make is that this is a nice example of the difference between syntax and semantics.

14.3.3 Two Modes of Understanding

Figure 14.2: https://www.gocomics.com/bloomcounty/2010/08/06, ©2010, Washington Post Co.

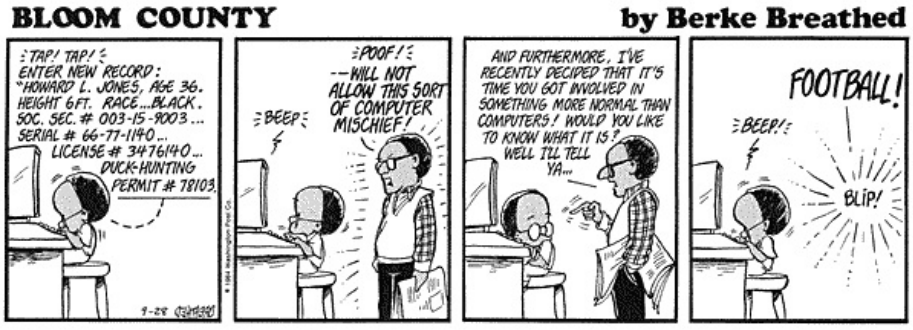


Figure 14.3: https://www.gocomics.com/bloomcounty/2010/08/09, ©2010, Washington Post Co.

Note that semantic understanding is a two-way street (or, to switch metaphors, semantic understanding is Janus faced):[4] Typically, we already understand SEM; thus, we can use SEM to help us understand SYN. For example, knowing something about the history and culture of an ancient civilization can help us understand its written language. But we can also use SYN to understand SEM. For example, language and scientific theories expressed in language enable us to describe and understand the world (as we discussed in Chapter 4). Rosenblueth and Wiener (1945, p. 317) observe that, in the 18th and 19th centuries, mechanical models were used to understand electrical problems, but that, in the 20th century, electrical models were used to understand mechanical problems! Swoyer (1991, p. 482) notes that a semantic interpretation of a language is a mapping "from the syntax to the semantics" (from SYN to SEM). But in

[4] http://en.wikipedia.org/wiki/Janus

note 27 (p. 504), he observes that in other structural representations, the mapping "runs in the opposite direction", from SEM to SYN to use our terminology.

Data types are another example: In §14.2.3, we said that an abstract data type can be implemented by a data structure:

> A data type is an abstract concept defined by a set of logical properties. Once such an abstract data type is defined and the legal operations involving that type are specified, we may implement that data type .... An implementation may be a *hardware implementation*, in which the circuitry necessary to perform the required operations is designed and constructed as part of a computer. Or it may be a *software implementation*, in which a program consisting of already existing hardware instructions is written to interpret bit strings in the desired fashion and to perform the required operations. —Tenenbaum and Augenstein (1981, p. 8)

But an abstract data type can itself be viewed as an implementation:

> A method of interpreting a bit pattern is often called a *data type*. ...
> ... a type is a method for interpreting a portion of memory. When a variable identifier is declared as being of a certain type, we are saying that the identifier refers to a certain portion of memory and that the contents of that memory are to be interpreted according to the pattern defined by the type.
> —Tenenbaum and Augenstein (1981, pp. 6, 45)

What matters is the existence of a mapping; its *direction* is a matter of which system is being used to understand the other. The crucial issue is which system (SYN or SEM) is antecedently understood. One person's antecedently understood domain is another's that needs to be understood.

Digression: A Recursive Definition of Understanding

We can combine the two kinds of understanding into a recursive definition of 'understand'. After all, if one understands a domain semantically in terms of an antecedently understood domain, we might wonder how that antecedently understood domain is understood. If it is understood in terms of yet another antecedently understood domain, we run the risk of an infinite regress, unless there is one domain that is understood in terms of *itself*, rather than in terms of another domain. But, if a domain is going to be understood in terms of itself, it would have to be understood in terms only of *its* properties and internal relations, and that means that it would have to be understood *syntactically*. So, the base case of understanding is to understand something syntactically—in terms of itself. The recursive case of understanding is to understand something semantically, in terms of something else that is already understood. (See Rapaport 1995 for further discussion.)

The antecedently understood domain can be viewed as an implementation of the domain that needs to be understood. So, (typically) SEM is an *implementation* of SYN. But, in line with our comments above, sometimes SYN is best understood as an implementation of SEM.

Consider a program written in a high-level programming language. Suppose that the program has a data structure called a "person record", containing information about

(that is, a representation of) a person, something like the record in the *Bloom County* cartoons in Figures 14.2 and 14.3. For instance, Howard L. Jones's record might look something like this:

```
(person-record:
 (name "Howard L. Jones")
 (age 36)
 (height (feet 6))
 (race Black)
 (ssn 003-15-9003)
 (serial-number 66-77-1140)
 (license-number 3476140)
 (duck-hunting-permit 78103)
)
```

This is merely a piece of syntax: a sequence of markers. You and I reading it might think that it represents a person named 'Howard L. Jones', whose age is 36, whose height is 6 feet, and so on. But as far as the computer (program) is concerned, this record might just as well have looked like this (McDermott, 1980):

```
(PR:
 (g100 n456)
 (g101 36)
 (g102 (u7 6)))
 (g103 r7)
 (g104 003159003)
 (g105 66771140)
 (g106 3476140)
 (g10778103)
)
```

And, in fact, the machine-language version of this record looks much like this (Colburn, 1999, p. 8). As long as the program 'knows" how to input new data, how to compute with these data, and how to output the results in a humanly readable form, it really doesn't matter what the data look like *to us*. That is, as long as the relationships among the symbols are well specified, it doesn't matter—as far as computing is concerned—how those symbols are related to other symbols that might be meaningful *to us*. That is why it is syntax, not semantics.

Now, there are at least two ways in which this piece of syntax could be implemented. One implementation, of course, might be Jones himself in the real world:[5] A person named 'Howard L. Jones', who is 36 years old, etc. Jones—the real person—implements that data structure; he is also a semantic interpretation of it.

Another implementation is the way in which that data structure is actually represented in the computer's machine language. That is, when the program containing that data structure is compiled into a particular computer's machine language, that data structure will be represented in some other data structure expressed in that machine

[5] Yes; I'm aware that this Jones is a cartoon character, hence not a real person!

language. That will actually be another piece of syntax. And that machine-language syntax will be an implementation of our original data structure.

But when that machine-language program is being executed by the computer, some region of the computer's memory will be allocated to that data structure (to the computer's representation of Jones, if you will), which will probably be an array of '0's and '1's, more precisely, of bits in memory. These bits will be yet another implementation of the original data structure, as well as an implementation of the machine-language data structure.

Question for Discussion:
What is the relation between the human (Jones himself) and this region of the computer's memory? Does the memory location "simulate" Jones? (Do bits *simulate* people?) Does the memory location *implement* Jones? (Do bits *implement* people?) Also: The '0's and '1's in memory can be thought of as the '0's and '1's on a Turing-machine tape, and Jones can be thought of as an interpretation of that Turing-machine tape. Now, recall from §10.4.1 what Cleland said about the difference between Turing-machine programs and mundane procedures: The former can be understood independently of any real-world interpretation (that is, they can be understood purely syntactically, to use the current terminology)—understood in the sense that we can describe the computation in purely '0'/'1' terms. (Can we? Don't we at least have to interpret the '0's and '1's in terms of a machine-language data structure, interpretable in turn in terms of a high-level programming-language data structure, which is interpretable, in turn, in terms of the real-world Jones?) Mundane procedures, on the other hand, must be understood in terms of the real world (that is, the causal world) that they are manipulating.

14.4 Chalmers's Theory of Implementation

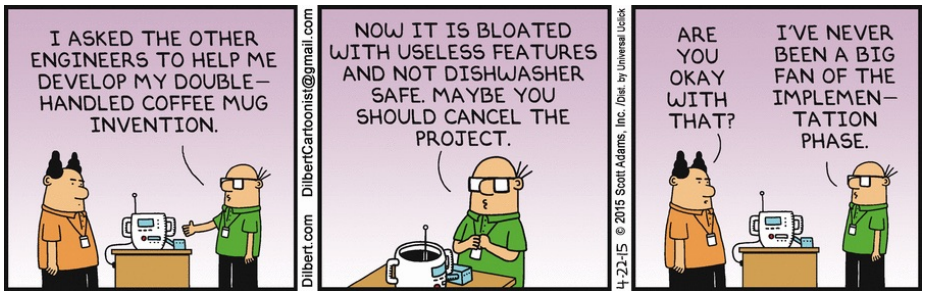


Figure 14.4: http://dilbert.com/strip/2015-04-22, ©2015, Scott Adams Inc.

14.4.1 Introduction

It is one thing to spell out the general structure of implementation as we did in §14.2, and another to suggest that the notion of semantic interpretation is a good way to understand what implementation is, as we did in §14.3.2. But we still need a more detailed

theory of implementation: What is the nature of the relation between an abstraction and one of its implementations?

One reason we need such a theory is in order to refute Searle's (1990) claim that "*any* given [physical] system can be seen to implement *any* computation *if interpreted appropriately*" (Chalmers, 2011, p. 325, my italics). David Chalmers's essay, "A Computational Foundation for the Study of Cognition" (2011; see also Chalmers 1996b) concerns both implementation and cognition, but, here, we will focus only on what he has to say about implementation.

One of his claims is that we need a "bridge" between the *abstract theory* of computation and *physical* systems that "implement" them. Besides 'bridge', other phrases that he mentions as synonyms for 'implement' are: 'realize' (that is, make real) and 'described by', as in this passage:

> Certainly, I think that when a physical system implements an a-computation [that is, a computation abstractly conceived], the a-computation can be seen as a description of it. —Chalmers (2012, p. 215, my interpolated brackets)

That is, the physical system that implements the abstract computation *is described by* that computation. (Here, Chalmers seems to agree with what we quoted Rescorla as saying; see §14.3.2.3.)

14.4.2 An Analysis of Chalmers's Theory

According to the simplest version of Chalmers's theory of implementation,

> A physical system implements a given computation when the causal structure of the physical system mirrors the formal structure of the computation.
> —Chalmers (2011, p. 326)

Almost every word of this needs clarification! For convenience, let P be a "physical system", and let C be a "computation":

- What kind of physical system is P?

  It need not be a computer, according to Chalmers.

- What kind of computation is C?

  Is it merely an abstract *algorithm*? Is it more specifically a Turing-machine *program*? Is it a program being executed—a "*process*"? In any case, it would seem to be something that is more or less abstract (because it has a "formal structure"; recall our discussion in §3.13.2 of the meaning of 'formal').

- What *is* a "formal structure"?

- What is a "causal structure"?

- What does 'when' mean?

> Is this intended to be *just* a sufficient condition ("when"), or is it supposed to be a stronger *biconditional* ("when *and only when*")?

- And the most important question: What does 'mirror' mean?

So, here is Chalmers's "more detail[ed]" version (2011, p. 326, my interpolated numerals), which begins to answer some of these questions:

> A physical system implements a given computation when there exists [1] a grouping of physical states of the system into state-types and [5a] a one-to-one mapping from formal states of the computation to physical state-types, such that [3] formal states related by an abstract state-transition relation are mapped [5b] onto physical state-types [2] related by a [4] corresponding causal state-transition relation.

Let me try to clarify this. (The numerals that I have inserted into the passage above correspond to the following list.) According to Chalmers, *P* implements *C* when (and maybe only when):

1. the *physical states* of *P* can be grouped into (physical-)state *types*,[6]
2. the physical-state *types* of *P* are related by a *causal* state-transition relation,
3. the *formal states* of *C* are related by an *abstract* state-transition relation,
4. the abstract state-transition relation of *C corresponds* to the causal state-transition relation of *P*,

   and

5. there is (a) *a 1–1 and* (b) *onto* map *from* the formal states of *C to* the physical-state types of *P*.

We still need some clarification: We have already defined "1–1" and "onto" maps in §7.6.1.2. A state is "formal" if it's part of the *abstract*—that is, non-physical—computation *C*, and a state is "causal" if it's part of the *physical* system *P*: Here, 'formal' just means "abstract", and 'causal' just means "physical". But what are abstract and causal "state-transition relations"? And what does 'correspond' mean?

Figure 14.5 might help to make some of this clear. In this figure, the physical system *P* is represented as a set of dots, each of which represents a physical state of *P*. These dots are partitioned into subsets of dots, that is, subsets containing the physical states. Each subset represents a state-type, that is, a set of states that are all of the same type. (That takes care of part 1 of Chalmers's account.)

The computation *C* is also represented as a set of dots. Here, each dot represents one of *C*'s formal states. The arrows that point *from* the dots in *C* (that is, from *C*'s

[6] A word on punctuation. Chalmers calls them 'physical state-types'. This *looks* as if he is talking about "state-types" that are "physical". But he is really talking about "types" of *P*'s "physical states". In other words, '-types' has wide scope over 'physical state', even though it doesn't look like that. So, I prefer to call them 'physical-state types'—note the position of the hyphen, which I think clarifies that, no matter how you hyphenate it, what's being discussed are "types of physical states". The types themselves are *not* physical; "types" are collections of things, and so they are abstract, even though the things that they are collections of *are* physical.

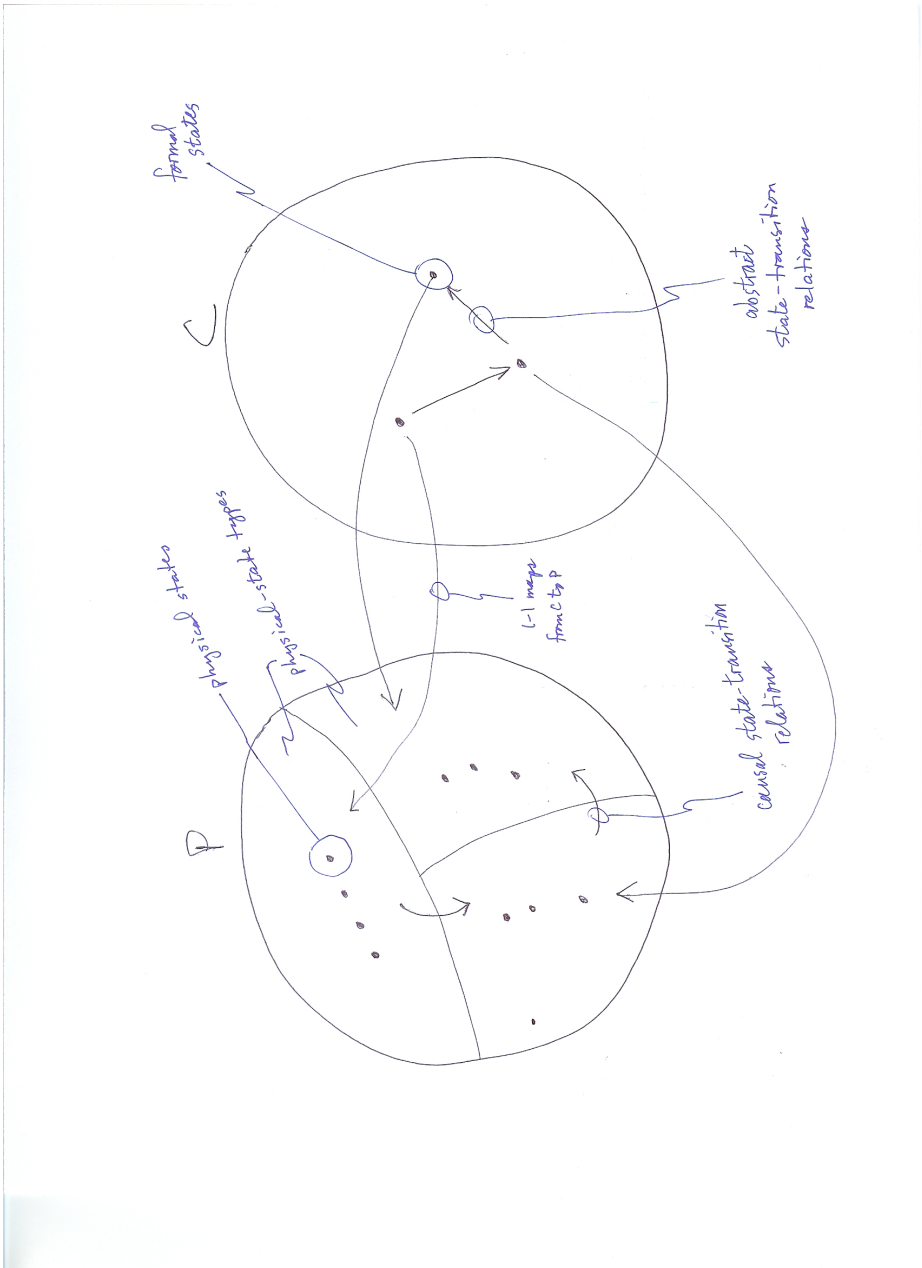


Figure 14.5: A pictorial representation of Chalmers's analysis of implementation; see text for explanation.

formal states) *to* the subsets of dots in *P* (that is, to the state-types in *P*) represent the 1–1 map from *C* to *P*. To make it a 1–1 map, each formal state of *C* must map to a *distinct* physical state-type of *P*. (That takes care of part 5a of Chalmers's account.)

The arrows *in* set *C* represent the abstract state-transition relation among *C*'s formal states (that's part 3). And the arrows *in* set *P* among *P*'s subsets represent the causal state-transition relation among *P*'s state-types (that's part 2). Finally, because *C*'s formal states are mapped *onto P*'s physical-state types, the 1–1 map is a 1–1 correspondence (this is part 5b).

Chalmers (2011, p. 326) seems to be aware of the two-sided nature of understanding; as we have seen, he says that *P* implements *C* when there is an isomorphism *from C to P*; yet, on the very next page, he says that *P* implements a finite-state automaton *M* if there is a mapping *from P to M*! (Of course, the mapping is a 1–1 correspondence, and therefore it has an inverse!)

Chalmers also says that *C*'s *abstract* state-transition relations "correspond" to *P*'s *causal* state-transition relations. I take it that this means that the 1–1 correspondence is a "homomorphism", that is, a structure-preserving map (that's part 4).

Digression: Homomorphism:
Suppose that $c_1, \ldots, c_n$ are entities that stand in relation R. Then a function f is a *homomorphism* $=_{def} f(R(c_1, \ldots, c_n)) = f(R)(f(c_1), \ldots, f(c_n))$. That is, if the c_i are related by some relation R, and if that relationship is mapped by f, then the image of $R(c_1, \ldots, c_n)$ will be the image of R applied to the images of the c_i. That's what it means to preserve structure.

Because the map is also "onto", it is an "isomorphism". (An isomorphism is a structure- or "shape"-preserving 1–1 correspondence.) So, *P* and *C* have the same *structure*.

We can then say that a physical system (perhaps a process) *P implements* an abstract computation *C* (which might be a Turing machine, or a less-powerful finite automaton (recall our discussion of these in §11.2), or a "combinatorial-state automaton" (see the Digression below) if and only if there is a "reliably causal" isomorphism $f : C \rightarrow P$. ('Reliably' probably means something like "you can always count on it".) Such an f is a relation between an abstract, computational model and something in the real, physical, causal world. This f is 1–1 and onto—a structure-preserving isomorphism such that the abstract, input-output and processing relations in *C* correspond to reliably causal processes in *P*. Michael Rescorla (2013, §1, p. 682) dubs Chalmers's view of implementation "structuralism about computational implementation". It is the fact that the *structure* of the physical system matches ("mirrors", in Chalmers's terms; more precisely, is isomorphic to) the structure of the computational system that matters. Note that *P* can be viewed as a semantic interpretation of *C*, and *C* can be viewed as a description of *P*.

Digression: Combinatorial-State Automata:

> Simple finite-state automata are unsatisfactory for many purposes, due to the monadic nature of their states. The states in most computational formalisms have a combinatorial structure: a cell pattern in a cellular automaton, a combination of tape-state and head-state in a Turing machine, variables and registers in a Pascal program, and so on. All this can be accommodated within the framework of combinatorial-state automata ..., which differ from ... [finite automata] only in that an internal state is specified not by a monadic label S, but by a vector $[S^1, S^2, S^3, \ldots]$. The elements of this vector can be thought of as the components of the overall state, such as the cells in a cellular automaton or the tape-squares in a Turing machine.
> —Chalmers (2011, p. 328); see the rest of Chalmers 2011, §2.1, for more details.

It follows from this analysis that:

- *Every* physical system implements *some* computation.

  That is, for every physical system P, there is some computation C such that P implements C. Does this make the notion of computation vacuous? No, because the fact that some P implements some C is not necessarily *the reason why* P is the *kind* of physical process that it is. (But, in the case of cognition, it *might* be the reason! We'll come back to this in Chapter 19.)

- But *not every* physical system implements *any* given computation.

  That is, it is not the case that, for every P and for every C, P implements C. That is, there is some P and there is some C such that P does *not* implement C (because there are computations that cannot be mapped isomorphically to P). For example, it is highly unlikely that the wall behind me implements Wordstar, because the computation is too complex.

- A *single* physical system *can* implement *more than one* computation.

  That is, for any P, there might be two different computations $C_1 \neq C_2$ such that P implements *both* C_1 *and* C_2. For example, my computer, right this minute as I type this, is implementing the "vi" text-processing program, a clock, Powerpoint, and several other computer programs, because each of these computations map to *different* parts of P.

14.4.3 Rescorla's Analysis of Chalmers's Theory

There is one aspect of Chalmers's analysis that we have not yet considered: "when" vs. "only when". Taken literally, Chalmers has offered only a *sufficient* condition for P being an implementation of C: *When* (that is, "if") there is a 1–1 correspondence from C to P as described above, *then* P implements C. But is this also a *necessary*

condition—is *P* an implementation of *C only when* (that is, only if) there is such a 1–1 correspondence? (That is: *When* (or *if*) *P* is an implementation of *C*, *then* there is such a 1–1 correspondence.)

Interestingly, Rescorla (2013) agrees that such structural identity is *necessary* for a physical system to implement a computation, but he *denies* that it is *sufficient*! That is, although any physical system that implements a computation must have the same structure as the computation, there are (according to Rescorla) physical systems that *have* the same structure as certain computations but that are *not* implementations of them (§1, p. 683). This is because *semantic* "relations to the social environment sometimes help determine whether a physical system realizes a computation" (Abstract, p. 681). The key word here is 'sometimes': "On my position, the implementation conditions for some but not all computational models essentially involve semantic properties" (§2, p. 684).

Roughly, the issue concerns the "intentionality" of implementation: Must *P* somehow be "intended" (by whom?) to implement *C*? Or could *P* be, so to speak, an "accidental" implementation of *C*?[7] (Recall our discussion in §3.3.3.2.1 of "chauvinism" vs. "liberalism" when trying to formalize informal notions.) In that case, Rescorla might say that *P* wasn't *really* an implementation of *C*. But Rescorla's position is rather more subtle. A year earlier, he had said:

> Mathematical models of computation, such as the Turing machine, are abstract entities. They do not exist in space or time, and they do not participate in causal relations. Under suitable circumstances, a physical system *implements* or *physically realizes* an abstract computational model. Some philosophers hold that a physical system implements a computational model only if the system has semantic or representational properties [... Ladyman 2009]. Call this *the semantic view of computational implementation.* In contrast, [Chalmers 1995; Piccinini 2006a], and others deny any essential tie between semantics and physical computation. *I agree with Chalmers and Piccinini.* —Rescorla (2012a, §2.1, p. 705, my italics)

But in Rescorla 2013, p. 684, after reciting the semantic and non-semantic passage just quoted in almost the same words, he says that he "*reject*[*s*] *both* the semantic and the non-semantic views of computational implementation" (my italics). We will investigate this issue in more detail in Chapter 17. But in Rescorla 2013, he provides a "counterexample to the non-semantic view" (§1, p. 684): an example of a physical implementation that—he claims—*requires* a representational (that is, a semantic) feature. (It is not enough to find an implementation that merely *has* a semantics; there are plenty of those, because a semantic interpretation can always be given to one.) One example that he gives is a Scheme program for Euclid's algorithm for computing GCDs (§4, p. 686):

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

[7] Perhaps in the same way that a fictional character might only "coincidentally resemble" a real person (Kripke, 2011, pp. 56, 72).

This is a recursive algorithm that we can paraphrase in English as follows:

> **To** compute the GCD of integers a and b, **do** the following:
> **If** $b = 0$,
> **then** output a
> **else** compute the GCD of b and the remainder of dividing a by b.[8]

Rescorla points out that "To do that, the machine *must* represent *numbers*. Thus, the Scheme program contains *content*-involving instructions ..." (§4, p. 687, my italics). A "content-involving instruction is [one that] is specified, at least partly, in semantic or representational terms" (§3, p. 685; he borrows this notion from Peacocke 1995). So, the Scheme program is specified in semantic terms (specifically, it is specified in terms of integers). Therefore, if a physical system is going to implement the program, that physical system must represent integers; that is, it requires semantics. Hence, "The Scheme program is a counter-example to the non-semantic view of computational implementation"
(§4, p. 687).

I can see that the machine *does* represent numbers (or can be interpreted as representing them). But why does he say that it *must* represent them? I can even see that for an agent to use such a physical computer *to compute GCDs*, the *agent must* interpret the computer as representing numbers. But surely an agent could use this computer, implementing this program, to print out interesting patterns of uninterpreted markers. (Recall the computer-in-the-desert of §3.9.5.)

To respond to this kind of worry, Rescorla asks us to consider two copies of this machine, calling them M_{10} and M_{13}. The former uses base-10 notation; the latter uses base-13. When each is given the input pair of *numerals* ('115', '20'), each outputs the *numeral* '5'. But only the former computes the GCD of the *numbers* 115 and 20. (The latter was given the integers 187 and 26 as inputs; but their GCD is 1.) So M_{10} implements the program, but M_{13} does not; yet they are identical physical computers.

One possible response to this is that the semantics lies in the user's interpretation of the inputs and outputs, not in the physical machine. Thus, one could say that both machines *do* implement the program, but that it is the user's interpretation of the inputs, outputs, and that program's symbols that makes all the difference. After all, consider the following Scheme program:

---

[8] Or, if you prefer:

> **To** compute the GCD of a and b, **do**:
> **If** $b = 0$,
> **then** output a
> **else begin**:
> divide a by b;
> let r be the remainder;
> compute the GCD of b and r
> **end**

```
(define (MYSTERY a b)
  (if (= b 0)
    a
    (MYSTERY b (remainder a b))))
```

If we are using base-10 notation, then we can interpret 'MYSTERY' as GCD; if we are using base-13 notation, then we might either be able to interpret 'MYSTERY' as some other numerical function or else not be able to interpret it at all. In either case, our two computers both implement the MYSTERY program.

One possible conclusion to draw from this is that any role that semantics has to play is not at the level of the abstract computation, but at the level of the physical implementation. Rescorla's response to this might be incorporated in these remarks:

> The program's formal structure does not even begin to fix a unique semantic interpretation. Implementing the program requires more than instantiating a causal structure that mirrors relevant formal structure. —Rescorla (2013, §4, p. 688)

I agree with the first sentence: We can interpret the MYSTERY program in many ways. I disagree with the term 'requires' in the second sentence: I would say that implementing the program *only* requires "instantiating the mirroring causal structure". But I would go on to say that if one wanted to use the physical implementation to compute GCDs, then one would, indeed, be required to do something extra, namely, to provide a base-10 interpretation of the inputs and outputs (and an interpretation of 'MYSTERY' as GCD).

In fact, Rescorla agrees that the semantic interpretation of 'MYSTERY' as GCD is required: "there is more to a program than meaningless signs. The signs have an intended interpretation ..." (§4, p. 689). But it is notoriously hard (some would say that it is logically impossible)[9] to pin down what "the intended interpretation" of any formal system is.

We will return to this debate and to Rescorla's example in Chapter 17. Till then, here are some questions to consider: Are the inputs to the Euclidean GCD algorithm *numerals* (like '10') or *numbers* (like 10 or 13)?[10] What about the inputs to a computer program written in Scheme that implements the Euclidean algorithm: Are *its* inputs numerals or numbers? (It may help to consider this analogous question: Is the input to a word-processing program the letter 'a' or an electronic signal or ASCII-code representing 'a'?)

Rescorla (2012b, p. 12; italics in original, my interpolated brackets) gives another example of semantic computation, in the sense of a computation that requires *numbers*, not (merely) *numerals*:

> A register machine contains a set of memory locations, called *registers*. A program governs the evolution of register states. The program may individuate register states syntactically. For instance, it may describe the machine as storing *numerals*

[9] In part because of something called the Löwenheim-Skolem Theorem. For discussion, see Suber 1997b.

[10] Note that the base-10 numeral '10' represents the number 10, but the base-13 numeral '10' represents the number 13.

> in registers, and it may dictate how to manipulate those syntactic items. Alternatively, the program may individuate register states representationally. Indeed, the first register machine in the published literature models computation *over natural numbers* [Shepherdson and Sturgis 1963, p. 219]. A program for this numerical register machine contains instructions to execute elementary arithmetical operations, such as *add 1* or *subtract 1*. A physical system implements the program only if [it] can execute the relevant arithmetical operations. A physical system executes arithmetical operations only if it bears appropriate representational relations to numbers. Thus, a physical system implements a numerical register machine program only if it bears appropriate representational relations to numbers. Notably, a numerical register machine program ignores *how* the physical system represent[s] numbers. It applies whether the system's numerical notation is unary, binary, decimal, etc. The program characterizes internal states representationally (e.g. *a numeral that represents the number 20 is stored in a certain memory location*) rather than syntactically (e.g. *decimal numeral "20" is stored in a certain memory location*). It individuates computational states through denotational relations to natural numbers. It contains mechanical rules (e.g. *add 1*) that characterize computational states through their numerical denotations.

I agree that this is a *semantic* computation. Note that it is *not* a Turing machine (which *would* be a purely syntactic computation). And note that there cannot be a *physical* numerical register machine,[11] only a *syntactic* one. This is not because there are no numbers, but because (if numbers do exist) they are not physical!

These are important questions, and we will return to them in Chapter 17. But there are two issues concerning the nature of computer programs that we need to look at first: In the next chapter, we'll consider whether any computer programs can be considered to be scientific theories. Then, in Chapter 16, we'll discuss whether computer programs can be "verified" (or proved "correct").

[11] *Numerical* as opposed to *numeral*!

Further Reading:
The problem of the nature of implementation is closely related to the mind-body (or the mind-brain) problem: Very roughly, (how) is the mind implemented in the brain? For a nice discussion of this, see Scheutz 2001, §1.

Eden and Turner 2007, §4, discusses "concretization ... a process during which an entity or entities of one category are synthesized (come into being) from entities of a more abstract category".

Sloman 2008 explores implementation and its relationship to the concept of a virtual machine.

Ladyman 2009, p. 379, critiques the kind of relationship that Chalmers sees between his formal and his causal mappings.

Dresner 2010 examines "the association between numbers and the physical world that is made in measurement" and argues that implementation "and (measurement-theoretic) representation" are "a single relation (or concept) viewed from different angles" (p. 276). Note that "representation" is a semantic concept.

Shagrir 2012b takes on almost everyone: He critiques Putnam (1988); Searle (1990); Chalmers (1996b); Scheutz (2001); and Piccinini (2006a), arguing that there can be "systems that simultaneously implement different complex automata" (p. 137).

The philosopher and computer scientist Matthias Scheutz has written extensively on implementation: Scheutz 1998 analyzes Searle's and Putnam's arguments, concluding that "a better notion of implementation is ... [needed] that avoids state-to-state correspondences between physical systems and abstract objects." To refute Putnam (1988), Scheutz (1999) replaces the notion of "implementation of a computation" with "realization of a *function*". His view is not inconsistent with the "semantic" theory that we presented in §14.3, CHECK THIS at least when he says (§7, p. 174): "what is the same [in the case of two realizations] is the syntactic structure of ... the function"; that is, they are different semantic interpretations of the same syntactic structure. §3 is an especially good discussion of Putnam's argument. Scheutz approaches implementation by way of its inverse, abstraction (from a physical system); see §7. Scheutz 2001 critiques Chalmers's theory of implementation and introduces a new theory of implementation based on a notion of "bisimulation". And Scheutz 2012, p. 75 argues that Chalmers's "definition of implementation still allows for unwanted implementations".

Chapter 15

Are Computer Programs Theories?

Version of 9 September 2018; DRAFT © 2004–2018 by William J. Rapaport

> . . . within ten years most theories in psychology will take the form of computer programs, or of qualitative statements about the characteristics of computer programs. —Simon and Newell (1958, pp. 7–8)

Figure 15.1: https://www.gocomics.com/nonsequitur/2014/11/07, ©2014, Wiley Ink, Inc.

15.1 Readings:

1. Required:

   (a) Simon, Herbert A.; & Newell, Allen (1956), "Models: Their Uses and Limitations", in Leonard D. White (ed.), *The State of the Social Sciences* (Chicago: University of Chicago Press): 66–83, http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=356856.

   (b) Simon, Herbert A. (1996), "Understanding the Natural and Artificial Worlds", Ch. 1, pp. 1–24 of Simon's *The Sciences of the Artificial, Third Edition* (Cambridge, MA: MIT Press).

   - Originally written in 1969; updated in 1996.
   - Chapter 1 also has interesting things to say about whether CS is a science.

   (c) Moor, James H. (1978), "Models and Theories", §4 of his "Three Myths of Computer Science", *British Journal for the Philosophy of Science* 29(3) (September): 213–222.

   (d) Thagard, Paul (1984), "Computer Programs as Psychological Theories", in O. Neumaier (ed.), *Mind, Language, and Society* (Vienna: Conceptus-Studien): 77–84.

2. Recommended:

   (a) Weizenbaum, Joseph (1976), *Computer Power and Human Reason: From Judgment to Calculation* (New York: W.H. Freeman).

   - Ch. 5 ("Theories and Models"), pp. 132–153.
   - Ch. 6 ("Computer Models in Psychology"), pp. 154–181.
     - Ch. 6 discusses computer programs as theories and the potential evils of AI, as well as presenting Weizenbaum's objections to Simon.

   (b) Wilks, Yorick (1990), "One Small Head: Models and Theories", pp. 121–134, in §4 ("Programs and Theories") of Derek Partridge & Yorick Wilks (eds.), *The Foundations of Artificial Intelligence: A Sourcebook* (Cambridge, UK: Cambridge University Press).

   - An earlier version appeared as Wilks 1974.
   - Has a useful, if sometimes confusing, overview of the many meanings of 'theory' and 'model'.

   (c) Daubert v. Merrell Dow Pharmaceuticals (92-102), 509 U.S. 579 (1993), http://openjurist.org/509/us/579

   - A Supreme Court case concerning what counts as "generally accepted" reliability by the scientific community. Has interesting observations on the nature of scientific theories and expertise.

15.2 Introduction

> I haven't formalized my theory of belief revision, but I have an algorithm that does it. — Frances L. Johnson (personal communication, February 2004).

The issue raised in this epigraph (from a former graduate student in my department) is whether an *algorithm* or a computer *program*—both of which are pretty formal, precise things—is different from a formal *theory*. Some might say that her algorithm *is* her theory. Others might say that they are distinct things and that her algorithm (merely) *expresses*—or *implements*—her theory. Does it really make sense to say that you *don't* have a formal theory of something if you *do* have a formal algorithm that implements your (perhaps informal) theory? Roger Schank, an AI researcher famous for taking a "scruffy"—that is, non-formal—approach to AI used formal algorithms to express his non-formal theories. That sounds paradoxical.

As part of our investigation into the nature of computer programs, we have seen that algorithms are said to be implemented in computer programs. If implementation is semantic interpretation (as I suggested in §14.3), then computer programs are semantic interpretations of algorithms, in the medium of some programming language. However, some philosophers have argued that computer programs are theories; yet theories are more like abstractions than they are like implementations. After all, if an algorithm (merely) *expresses* a theory, then a theory is akin to an abstract idea, as in our discussion in §13.4 of copyrights. And others have argued that computer programs are *simulations* or *models* of real-world situations, which sounds more like an implementation than an abstraction.

In §4.7, we briefly discussed the nature of scientific theories. In this chapter, we will look further into the nature of theories, models, and simulation, and whether programs are (scientific) theories. And we will begin an investigation into the relation of a program to that which it models or simulates. (We'll continue that investigation in Chapter 17.)

15.3 Simulations, Theories, and Models

15.3.1 Simulations

Simulations are sometimes contrasted with "emulations". And sometimes a simulation is taken to be an "imitation". Let's look at these distinctions.

15.3.1.1 Simulation vs. Emulation

There is no standard, agreed-upon definition of either 'simulation' or 'emulation'. This sort of situation occurs unfortunately all too frequently. Therefore, it is always important for you to try to find out how a person is using such terms before deciding whether to agree with what they say about them.

Here is one definition of 'simulate', from the *Encyclopedia of Computer Science* (R. Smith 2000):

> *x* **simulates** *y* means (roughly): *y* is a real or imagined system, and *x* is a model of *y*, and we experiment with *x* in order to understand *y*.

This is only a rough definition, because I have not said what is meant by 'system', 'model', or 'understand', not to mention 'real', 'imagined', or 'experiment'! Typically, a computer program (*x*) is said to simulate some real-world situation *y* when program *x stands in for y*, that is, when *x* is a model of situation *y*. If we want to understand the *situation*, we can do so by experimenting with the *program*. In the terminology of §14.3.3, presumably the program is antecedently understood—at least it is more understandable than the situation that it simulates, because it is *designed* by someone. Perhaps the program is easier to deal with or to manipulate than the real-world situation. In an extreme case, *x* simulates *y* if and only if *x* and *y* have the same input-output behavior, but they might differ greatly in some of the details of how they work.

And, following another *Encyclopedia of Computer Science* definition (Habib, 2000), let's say that

> *x* **emulates** *y* means (roughly)
>
> > *either*:
> > *x* and *y* are computer systems, and *x* interprets and executes *y*'s instruction set by implementing *y*'s operation codes in *x*'s hardware—that is, hardware *y* is implemented as a virtual machine on *x*,[1]
> >
> > *or*:
> > *x* is some software feature, and *y* is some hardware feature, and *x* simulates *y*, doing what *y* does "exactly" as *y* does it.

In general, *x* emulates *y* if and only if *x* and *y* not only have the same input-output behavior (*x* not only simulates *y*), but *x* also uses the same algorithms and data structures as *y*.

It is unlikely that being a simulation and being an emulation are completely distinct notions. More likely, they are the ends of a spectrum, in the middle of which are *x*s and *y*s that differ in the level of detail of the algorithms and data structures that *x* uses to do *y*'s job. At the "pure" simulation end of the spectrum, only *x*'s and *y*'s external, input-output behaviors agree; at the "pure" emulation end, all of their internal behaviors also agree. Perhaps, then, the only pure example of emulation of *y* would be *y* itself! Perhaps, even, there is no real distinction between simulation and emulation except for the degree of faithfulness to what is being simulated or emulated.

15.3.1.2 Simulation vs. Imitation

In many cases, *y* is only an imagined situation, whereas *x* will always be something real. On the other hand, it is often said that *x* is "merely" a simulation, which suggests that *y is* real but that *x* is not. That is, the word 'simulation' has a connotation of

[1] On emulation as *simulation by a virtual machine*, see Denning and Martell 2015, p. 212.

"imitation" or "unreal". For example, it is often argued that a simulation of a hurricane is not a real hurricane, or that a simulation of digestion is not real digestion.

But there are cases where a simulation *is* the real thing. (Or would such simulations be better called 'emulations'?) For example, although a scale model of the Statue of Liberty is not the real Statue of Liberty, a scale model of a scale model (of the Statue of Liberty) *is* itself a scale model (of the Statue of Liberty). A Xerox copy or PDF or faxed copy of a document *is* that document, even for legal purposes (although perhaps not for historical purposes;[2] see Korsmeyer 2012). Some philosophers and computational cognitive scientists have argued that a computational simulation of cognition really is cognition (Edelman, 2008a; Rapaport, 2012b). In general, it seems, a simulation of information *is* that information.

There are also cases where it is difficult or impossible to tell if something is a simulation or not, such as Bostrom's argument that we are living in a *Matrix*-like simulation (which we briefly mentioned in §9.8.2.2). After all, if a program could be a scientific theory, then the process that comes into being when the program is executed could be a model of what the program is a theory of, and if some models *are* the kind of thing that they model, then a simulation of the real world could be a real world.

Further Reading:

Peschl and Scheutz 2001b argues that computer programs are good simulations (and even implementations) of cognition, but only as long as they respect "the temporal metric imposed by physics". See also Peschl and Scheutz 2001a, and this passage from —Simonite 2009, my interpolated brackets:

> Despite the increasing sophistication of computer simulations, finding ways to show complex air flows visually is critical to understanding aerodynamics ... and new ways to do that in large wind tunnels are valuable. "You cannot solve everything completely in space and time on a computer," [Alex] Liberzon [of Tel Aviv University] told *New Scientist*. "Simulations do not capture the full complexity of wakes and other features, which can exhibit large changes in behaviour caused by very small changes."

Nick Bostrom (2003) argues that if "the human species is [not] likely to go extinct before reaching a [technologically advanced] posthuman stage" and if "any posthuman civilization is ... likely to run a significant number of simulations of their evolutionary history", then "we are almost certainly living in a computer simulation" (my interpolated brackets). And Donald Hoffman (2009) argues that our internal mental image of the external world need not bear any resemblance to the actual external world, any more than the graphical user interface for an operation system need bear a resemblance to the "diodes, resistors, voltages and magnetice fields in the computer", on the grounds that what is important from the standpoint of evolution is not accuracy but fitness. "The very evolutionary processes that endowed us with our interfaces might also have saddled us with the penchant to mistake their contents for objective reality" (§1.6). This seems to be consistent with Bostrom's theory. See also Papakonstantinou 2015 and Rothman 2016. We'll discuss this a bit more in Chapter 20.

Shieh and Turkle 2009 is an interview with sociologist of science Sherry Turkle: "Computer simulations have introduced some strange problems into reality."

[2] Unless a PDF *is* the original document!

Questions for the Reader:
Does a universal Turing machine that is executing a program for some algorithm *A* *simulate* (or *emulate*) a (dedicated) Turing machine for *A*?

Is that universal Turing machine "really" executing *A*, or is it "merely" simulating (or emulating) it?

15.3.2 Theories

Simulations are one thing; theories are another. Recall our discussion in §4.7 of the term 'theory'. When people say, in ordinary language, that something is a "theory", they often mean that it is mere speculation, that it isn't necessarily true. But scientists and philosophers use the word 'theory' in a more technical sense. (For a humorous illustration of this, see the cartoon in Figure 15.1 at the beginning of this chapter.)

This is one reason that people who believe in the "theory" of evolution and those who don't are often talking at cross purposes, with the former saying that evolution is a true, scientific theory:

> Referring to biological evolution as a theory for the purpose of contesting it would be counterproductive, since scientists only grant the status of theory to well-tested ideas.
> —Terry Holliday, Kentucky education commissioner, 2011;
> cited in *Science* 337 (24 August 2012): 897.

and the latter saying that, if it is only a theory—if, that is, it is mere speculation—then it might not be true:

> The theory of evolution is a theory, and essentially the theory of evolution is not science—Darwin made it up.
> —Ben Waide, Kentucky state representative, 2011;
> cited in *Science* 337 (24 August 2012): 897.

They are using the word in very different senses.

Further complicating the issue, there are at least two views within the philosophy of science about what scientific theories are:

- On the *syntactic* approach to theories (due to a group of philosophers known as the "Logical Positivists"; see Uebel 2012), a theory is an abstract description of some situation (which usually is, but need not be, a real-world situation) expressed in a formal language with an axiomatic structure. That is, a theory is a formal system (see §14.3.2.1). Such a "theory" is typically considered to be a set of sentences (linguistic expressions, well-formed formulas) that describe a situation or that codify (scientific) laws about a situation. (This is the main sense in which the theory of evolution is a "theory".) Such a description, of course, must be expressed in some language. Typically, the theory is expressed in a formal, axiomatic language that is semantically interpreted by rules linking the sentences to "observable" phenomena. These phenomena either are directly

observable—either by unaided vision or with the help of devices such as microscopes and telescopes—or are theoretical terms (such as 'electron') that are definable in terms of directly observable phenomena (such as a vapor trail in a cloud chamber).

- On the *semantic* approach to theories (due largely to the philosopher Patrick Suppes; see Frigg and Hartmann 2012), theories are the *set-theoretic models of* an axiomatic formal system. Such models are isomorphic to the real-world situation being modeled. (Weaker semantic views of theories see them as "state spaces" (http://en.wikipedia.org/wiki/State_space) or "prototypes" (http://en.wikipedia.org/wiki/Prototype), which are merely "similar" to the real-world situation.) A theory viewed semantically can clearly resemble a simulation (or an emulation).

Further Reading:
Partridge and Wilks 1990, an anthology on the foundations of AI, has two sections on the nature of theories: §3 ("Levels of Theory", pp. 95–118), contains Marr 1977, Boden 1990a, and Partridge 1990. And §4 ("Programs and Theories", pp. 119–164), contains Wilks 1974, Bundy and Ohlsson 1990, and T. Simon 1990.

15.3.3 Models

> ... computational models are better able to describe many aspects of the universe better than any other models we know. All scientific theories can, for example, be modeled by programs. —Knuth (2001, p. 168)

Both simulation and semantic theories are said to be "models". So, what is a model?

The notion of *model* is associated with what I have called "The Muddle of the Model in the Middle" (Wartofsky, 1966, 1979; Rapaport, 1995). As with theories, there are two different uses of the term 'model': It can be used to refer to a *syntactic* domain, as in the phrase 'mathematical model' of a real-world situation. And it can be used to refer to a *semantic* domain, as in the phrase 'set-theoretic model' of a mathematical theory. And, of course, there is the real-world situation that both of them refer to in some way. The "muddle" concerns the relationships among these.

We saw the dual, or Janus-faced, nature of models in §14.3.3, when we briefly considered what Brian Cantwell Smith (1987) called a "correspondence continuum" (§14.3.1): Scientists typically begin with data that they then interpret or model using a formal theory; so, the data are the syntactic domain in need of understanding, and the formal theory is its semantic domain in terms of which it can be understood. The formal theory can then be modeled set-theoretically or mathematically; so, the formal theory now becomes the syntactic domain, and the set-theoretic or mathematical model is *its* semantic domain. But that set-theoretic or mathematical model can be interpreted by some real-world phenomenon; so, the model is now the syntactic domain, and the real world is the semantic domain. To close the circle, that real-world phenomenon consists of the same kind of data that we started with! (Compare the example in §14.3.3 of person records and persons.) Hence my phrase "the muddle of the model *in the middle*".

No one seems to deny that computer programs can be simulations or models. But can they be *theories*?

15.4 Computer Programs as Theories

15.4.1 Introduction

Computational cognitive scientists such as Philip N. Johnson-Laird, Allen Newell, Zenon Pylyshyn, and Herbert Simon have all claimed that (some) computer programs *are* theories, in the sense that the programming languages in which they are written are languages for theories and that the programs are ways to express theories. First, consider these passages from their writings (my italics and bracketed interpolations throughout):

Simon and Newell (1962, p. 97):

1. Computers are quite general symbol-manipulating devices that can be programmed to perform nonnumerical as well as numerical symbol manipulation.
2. Computer programs can be written that use nonnumerical symbol manipulating processes to perform tasks which, in humans, require thinking and learning.
3. *These programs can be regarded as theories, in a completely literal sense, of the corresponding human processes.* These theories are testable in a number of ways: among them, by comparing the symbolic behavior of a computer so programmed with the symbolic behavior of a human subject when both are performing the same problem-solving or thinking tasks.

Johnson-Laird (1981, pp. 185–186):

> Computer programming is too useful to cognitive science to be left solely in the hands of the artificial intelligenzia [sic]. There is a well established list of advantages that programs bring to a theorist: they concentrate the mind marvelously; they transform mysticism into information processing, forcing the theorist to make intuitions explicit and to translate vague terminology into concrete proposals; they provide a secure test of the consistency of a theory and thereby allow complicated interactive components to be safely assembled; they are "working models" whose behavior can be directly compared with human performance. Yet, many research workers look on the idea of *developing their theories in the form of computer programs* with considerable suspicion. The reason ... [i]n part ... derives from the fact that any large-scale program intended to model cognition inevitably incorporates components that lack psychological plausibility .... The remedy ... is not to abandon computer programs, but to *make a clear distinction between a program and the theory that it is intended to model.* For a cognitive scientist, the single most important virtue of programming should come ... from the business of developing [the program]. Indeed, the aim should be neither to *simulate* human behavior ... nor to exercise artificial intelligence, but to force the theorist to think again.

Pylyshyn (1984, p. 76):

> [T]he . . . requirement—that we be able to implement [a cognitive] process in terms of an actual, running program that exhibits tokens of the behaviors in question, under the appropriate circumstances—has far-reaching consequences. One of the clearest advantages of *expressing a cognitive-process model in the form of a computer program* is, it provides a remarkable intellectual prosthetic for dealing with complexity and for exploring both the entailments of a large set of proposed principles and their interactions.

Johnson-Laird (1988, p. 52):

> *[T]heories of mind should be expressed in a form that can be modelled in a computer program.* A theory may fail to satisfy this criterion for several reasons: it may be radically incomplete; it may rely on a process that is not computable; it may be inconsistent, incoherent, or, like a mystical doctrine, take so much for granted that it is understood only by its adherents. These flaws are not always so obvious. Students of the mind do not always know that they do not know what they are talking about. The surest way to find out is to try to devise a computer program that models the theory.

Simon (1996a, p. 160):

> In the late 1950s, the hypothesis was advanced that human thinking is information processing, alias symbol manipulation. . . . [T]hese ideas [were translated] into symbolic (nonnumerical) computer programs that *simulated* human mental activity at the symbolic level. The traces of these programs could be compared in some detail with data that tracked the actual paths of human thought (especially verbal protocols) in a variety of intellectual tasks, and the *programs' veracity as theories* of human thinking could thereby be tested.

The basic idea is that a theory must be expressed in some language. As an old saying has it, "How can I know what I think till I see what I say?" (Wallas, 1926, p. 54). If you don't express a theory in a language, how do you know what it is? And if you don't write your theory down in some language, no one can evaluate it.

Scientific theories, on this view, are sets of sentences. And the sentences have to be in some language: Some theories are expressed in a natural language such as English, some in the language of mathematics, some in the language of formal logic, some in the language of statistics and probability. The claim here is that some theories can be expressed in a programming language.

One advantage of expressing a theory as a computer program is that *all* details must be filled in. That is, a computer program must be a full "implementation" of the theory. Of course, there will be implementation-dependent details. There is certainly a difference between a theory and the part of the world that it is a theory of. One such difference is this:

> Why should theories of all kinds make irrelevant statements—possess properties not shared by the situations they model? The reason is clearest in the case of electromechanical analogues. To operate at all, they have to obey electromechanical

> laws—they have to be made of something—and at a sufficiently microscopic level these laws will not mirror anything in the reality being pictured. If such analogies serve at all as theories of the phenomena, it is only at a sufficiently high level of aggregation. —Simon and Newell (1956, p. 74)

For another example, if the theory is expressed in Java, there will be details of Java that are irrelevant to the theory itself. This is an unavoidable problem arising whenever an abstraction is implemented. It is only at the more abstract levels ("sufficiently high level[s] of aggregation") that we can say that an implementation and a corresponding abstract theory are "the same". So, one must try to ensure that such details are indeed irrelevant. One way to do so is to make sure that two computer programs expressing the same theory but that are written in two different programming languages—with different implementation-dependent details—have the same input-output, algorithmic, and data-structure behavior (that is, that they fully emulate each other).[3]

Another advantage of expressing a theory as a computer program is that you can run the program to see how it behaves and what predictions it makes. So, in a sense, the theory becomes its own model and can be used to test itself. As Joseph Weizenbaum 1976, pp. 144–145 says:

> ... theories are texts. Texts are written in a language. Computer languages are languages too, and theories may be written in them. ... Theories written in the form of computer programs are ordinary theories as seen from one point of view. ... But the computer program has the advantage [over "a set of mathematical equations" or even a theory written in English] not only that it may be understood by anyone suitable trained in its language, ... but that it may also be run on a computer. ... A theory written in the form of a computer program is thus both a theory and, when placed on a computer and run, a model to which the theory applies.

Further Reading:
Apostel 1961, pp. 1–2, suggests that a *computer* can be a model of the central nervous system, and that that model might be easier to study than the system itself. If it is the *computer* that is the model, then it makes sense to say that the computer's *program* expresses the theory that the model is a model of. (Keep in mind, however, that there is an ambiguity over whether the model is a model of a *theory* or of some real-world phenomenon.)

Let's look at three explicit arguments for the conclusion that computer programs can be scientific theories—two due to Herbert Simon, and one from the Supreme Court.

[3] In alternative terminolgy, by implementing the theory in two different programming languages, you "divide out" the irrelevant implementation-dependent details. See Rapaport 1999, §3.2, and Rapaport 2005b, p. 395.

15.4.2 Simon & Newell's Argument from Analogy

Simon, along with Allen Newell (1956), argued as follows:

1. x is a *theory* of y iff x is an *analog* of y (p. 82, column 1).

   > More precisely, the *content* of a *theory* of y is identical to the *content* of an *analog* of y. The only difference between them is the way in which they are *expressed*. We could equally well say that a syntactic theory of y (for example, a verbal or mathematical theory of y) is an analog of y (p. 75, col. 2).
   >
   > In fact, we only need the right-to-left direction of this premise: All analogs are theories.[4]

2. (Some) computer programs are analogs.

   > For example, some computer programs are analogs of mathematical processes (p. 71, col. 2; pp. 79–82).

3. $\therefore$ (Some) computer programs are theories.

But what *kind* of theory is a computer program? According to Simon and Newell, a theory is a set of statements, but those statements could be:

verbal: "Consumption increases linearly with income, but less than proportionately" (p. 69, col. 2).

mathematical: "$C = a + bY;\ a > 0;\ 0 < b < 1$" (p. 70, col. 1)

or

analog:

> The idea that the flows of goods and money in an economy are somehow analogical to liquid flows is an old one. There now exists a hydraulic mechanism ... one part of which is so arranged that, when the level of the colored water in one tube is made to rise, the level in a second tube rises ... , but less than proportionately. I cannot "state" this theory here, since its statement is not in words but in water. (p. 70, col. 1)

Presumably, they would classify programming languages as being of the mathematical kind, from which it would follow that computer programs are theories expressed in that language. Alternatively, there seems to be no reason not to admit a fourth kind of theory, namely, one expressed computationally, that is, in procedural language.

Here is a reason why theories expressed as computer programs may be better than theories expressed in mathematics or in English ("verbally"): It has to do with the idea that such computational theories are analogies:

> ... what is the particular value of the computer analogy? Why not work directly toward a mathematical (or verbal) theory of human problem-solving processes without troubing about electronic computers? ... it is at least possible, and perhaps

[4] For more on analogy, see Hofstadter and Sander 2013.

> even plausible, that we are dealing here with systems of such complexity that we have a greater chance of building a theory by way of the computer program than by a direct attempt at mathematical formulation. (p. 81)

Note first that they seem to consider (some) computer programs as analogy theories, not mathematical theories! Second, computation is perhaps the best way of managing complexity (as we saw in §3.14.3).

There are two advantages of expressing a theory in a programming language. "First, we would experiment with various modifications of the ... program to see how closely we could simulate in detail the observable phenomena" (pp. 81–82). In other words, we can run the program to see how it behaves—to see how good a theory it is—and we can then modify the program (and then run the modified version) in order to make it a better theory.

Second, the program can (or, at least, should) be written in such a way that it *explains* what it is doing: "The computer, however complex its over-all program, could be programed [sic] to report, in accurate detail, a description of any part of its own computing processes in which we might be interested" (p. 82). This, of course, can make it easier not only to debug and improve the *program*, but also to correct and improve the *theory*.

Further Reading:
The ability—and the desirability—of a program to explain its own behavior is also important for the *ethical* use of computer programs; recall §3.15.2.5 and see Chapter 18. In the context of ethical computing, Neumann 1993 contains useful, real-life examples of ways in which simulations (and theories) can fail to be precise models of reality, and it discusses "the illusion that the *virtual* is *real*" (quoting Rebecca Mercuri).

15.4.3 Simon's Argument from Prediction

In a later essay, Simon said:

> These programs, which predict each successive step in behavior as a function of the current state of the memories together with the current inputs, are theories, quite analogous to the differential equation systems of the physical sciences.
> —Simon (1996a, pp. 161–162)

This is more a statement that (some) computer programs are scientific theories than an argument for that conclusion. But an argument for it can, perhaps, be constructed from it:

1. Differential equation systems of the physical sciences predict successive steps in physical processes as a function of the current state together with the current inputs.
2. Anything that allows prediction (of successive steps in some process as a function of the current state together with the current inputs) is a theory.
3. Therefore, differential equation systems are theories (in physics).

4. Cognitive computer programs predict successive steps in human cognitive behavior as a function of the current state of the memories together with the current inputs.

5. Therefore, (cognitive) computer programs are theories (in psychology).

The point is that the reason that we consider differential equation systems to be theories is the same reason that we should consider computer programs (cognitive ones in particular, but other kinds of programs as well) to be theories.

Well, maybe not *all* computer programs. Arguably, a computer program for adding two numbers or for computing income tax is not a theory. But maybe they should be considered to be theories expressed computationally: a theory of addition in the first case, a theory of taxation in the second!

Simon believes that computer programs are simultaneously both theories and simulations:

> Thus the digital computer provided both a means (program) for stating precise theories of cognition and a means (simulation, using these programs) for testing the degree of correspondence between the predictions of theory and actual human behavior. —Simon (1996a, p. 160)

Thus, for Simon, computer programs are a very special kind of theory. Not only are they statements, but they are simultaneously models—instances of the very thing that they describe. Well, perhaps not quite: They only become such instances when they are being executed.

This duality gives them the ability to be self-testing theories. And their precision gives them the ability to pay attention to details in a way that theories expressed in English (and perhaps even theories expressed in mathematics) lack.

Simon hedges a bit, however:

> ... a program was analogous to a system of differential (or difference) equations, hence could *express* a dynamic theory. —Simon (1996a, p. 161, my italics)

So, is it the case that a program *is* a theory? Or is merely the case that a program *expresses* a theory? Perhaps this distinction is unimportant. After all, it hardly seems to matter whether a system of equations *is* a theory or merely *expresses* a theory. (The distinction is roughly akin to that between a sentence and the proposition that it expresses.)

Further Reading:
Downes 1990 is a critique of Simon's views on the philosophy of science in general, and of programs as theories in particular.

15.4.4 Daubert vs. Merrell-Dow

As we have seen, there are several questions to consider:

- Is a computational theory (of X) a theory?
- Is a computational theory (of X) a *scientific* theory?
- What is a computational theory (of X)?

Daubert vs. Merrell-Dow Pharmaceuticals, Inc., was a 1993 Supreme Court case "determining the standard for admitting expert testimony in federal courts" (https://en.wikipedia.org/wiki/Daubert_v._Merrell_Dow_Pharmaceuticals,_Inc.). My colleague Sargur N. Srihari recommended Daubert to me, after his experience being called as an expert witness on handwriting analysis, on the grounds that his computer programs that could recognize handwriting were scientific theories of handwriting analysis.

Presumably, a computer scientist is an expert on CS. But is a computer scientist who writes a computer program about (or who develops a computational theory of) X (where $X \neq$ CS) thereby an expert on X? Or must that computer scientist *become*, or work with, an expert on X? (Recall question 13 in §3.17 about who counts as being a computer scientist.)

Two points that we have made about the nature of science were (1) Popper's view that a statement was scientific to the extent that it was falsifiable (§4.9.1.1) and (2) Simon's views about bounded rationality (§3.15.2.3). These are nicely summarized in three comments in Daubert:

> ... scientists do not assert that they know what is immutably 'true'—they are committed to searching for new, *temporary* theories to explain, *as best they can,* phenomena.
> —Brief for Nicolaas Bloembergen et al. as Amici Curiae 9
> (cited in Daubert at II.B.24 in the online version, my italics)

> Science is not an encyclopedic body of knowledge about the universe. Instead, it represents a process for proposing and refining theoretical explanations about the world that are subject to further testing and refinement.
> —Brief for American Association for the Advancement of Science and the National Academy of Sciences as Amici Curiae 7–8 (cited in Daubert at II.B.24)

> ... there are important differences between the quest for truth in the courtroom and the quest for truth in the laboratory. Scientific conclusions are subject to perpetual revision. Law, on the other hand, must resolve disputes finally and quickly.
> —Daubert, at III.35

Justice Harry Blackmun, writing in Daubert at II.B.24, citing the first two of these quotes, states that "in order to qualify as 'scientific knowledge,' an inference or assertion must be derived by the scientific method". So, if a computer program that can, say, identify handwriting is a good *scientific* theory of handwriting, is its creator a scientific expert on handwriting?

There are two concerns with this: First, a computer program that can identify handwriting need not be a good *scientific theory* of handwriting. It might be a "lucky guess" not based on any scientific theory, or it might not even work very well outside carefully selected samples. Second, even if it is *based on* a scientific theory of handwriting and works well on arbitrary samples, the programmer need only be a good *interpreter* of the theory, not necessarily a good handwriting scientist.

However, if a computer scientist studies the nature of handwriting and develops a scientific theory of it that is then expressed in a computer program capable of, say, identifying handwriting, then it would seem to be the case that that computer scientist is (also) a scientific expert in handwriting.

Blackmun, writing in Daubert at II.C.28, suggests four tests of "whether a theory or technique is scientific knowledge". Note that this could include a computer program, whether or not such programs are (scientific) theories:

Testability (and falsifiability) (II.C.28):

> Computer programs would seem to be scientific on these grounds, because they can be tested and possibly falsified, by simply running the program on a wide variety of data to see if it behaves as expected.

Peer review (II.C.29):

> Surely, a computer program can (and should!) be peer reviewed.

Error rate (II.C.30):

> It's not immediately clear what Blackmun might have in mind here, but perhaps it's something like this: A scientific theory's predictions should be within a reasonable margin of error. To take a perhaps overly simplistic example, a polling error of 5 ± 4 points is not a very accurate ("scientific") measurement, nor is a measurement error of 5.00000 ± 0.00001 inches if made with an ordinary wooden ruler. In any case, surely a computer program's errors should be "reasonable".

General acceptance (II.C.31):

> A computer program that is not based on a "generally accepted" scientific theory or on "generally accepted" scientific principles would not be considered scientific.

Whether or not Blackmun's four criteria are complete or adequate is not the point here. The more general point is that, whatever criteria are held to be essential to a theory's being considered scientific should also apply to computer programs that are under consideration.

Further Reading:
Tymoczko 1979 discusses whether a computer program can be (part of) a proof of a mathematical theorem. For a survey of critiques of Tymoczko's arguments, see Scherlis and Scott 1983, §3.

And Ray Turner has argued that programming languages are mathematical theories:

> That computer science is somehow a mathematical activity was a view held by many of the pioneers of the subject, especially those who were concerned with its foundations. At face value it might mean that the actual activity of programming is a mathematical one. Indeed, at least in some form, this has been held. But here we explore a different gloss on it. We explore the claim that programming languages are (semantically) mathematical theories. This will force us to discuss the normative nature of semantics, the nature of mathematical theories, the role of theoretical computer science and the relationship between semantic theory and language design. —Turner 2010, Abstract

15.5 Computer Programs *Aren't* Theories

However, philosophers James Moor (1978, §4) and Paul Thagard (1984) argue that computer programs are *not* theories, on the grounds that they are neither sets of (declarative) sentences nor set-theoretic models of axiom systems.

15.5.1 Moor's Objections

Moor (1978, pp. 219–220) says that computer models simulate phenomena in the real world and that models "help [us] understand and test theories". He also warns that

> computer scientists often speak as if there is no distinction among programs, models, and theories; and discussions slide easily from programs to models and from models to theories. (p. 220)

There is no question that this is the case, as might be clear from our earlier discussion, but what picture does Moor himself provide? Presumably, a theory is a kind of description of part of the real world. A model helps us understand the *theory*, hence it only *indirectly* helps us understand the *world*. Yet a (computer) model is said to simulate the *world*. Here is one way to make sense of this: A theory is a syntactic domain that has *two* semantic interpretations: One semantic domain is the real world; the other is a (computer) model of the real world. Presumably, the computer model is easier to understand (and to manipulate) than the real world, which is why it can help us test the *theory*.

This is consistent with what Moor says next:

> One can have a theory, *i.e.*, a set of laws used to explain and predict a set of events, without having a model except for the subject matter itself. Also, one can have a model of a given subject matter, *i.e.*, a set of objects or processes which have an isomorphism with some portion of the subject matter, without having a theory about the subject matter. (p. 220)

So, for Moor, there are three independent things: a portion of *the real world* (the "subject matter"); a *theory* about the real world, which offers explanations, predictions, and (presumably) descriptions of (a portion of) the real world; and a *model* of the real world, which could be a set-theoretic object that is isomorphic to (a portion of) the real world. But, if the theory describes the real world, then it *also* describes a model that is isomorphic to the real world. That's why I said, above, that the syntactic domain that is the theory has *both* the real world *and* the model of the real world as two semantic interpretations of it.

Now, what about computer programs as *models*? Moor says that a computer *model* "is ... more than just a computer *program*" (p. 220, my italics): To turn a computer program into a computer model, he says, you need a semantic interpretation function between the program and the portion of the real world that it is modeling. I think that makes sense: The program is merely a syntactic object; the parts of the program need not "wear their meanings on their sleeve", so to speak. We saw this in §14.3.3, when we noted that a "person record" that had slots for things like "name" and "age" (hence "obviously" modeling a real person, at least "obviously" to a user of the program) would work just as well as a "PR" record that had slots *unobviously* labeled 'g100' or 'g101'. To know that such a computer program modeled a *person*, one would have to know that 'g100' was to be semantically interpreted as a *name* and that its value (n456, in our example) was to be semantically intepreted as the name 'Howard L. Jones'.

So far, so good. But Moor goes on to say that it is a "myth" that a computer program that is a model of a real-world phenomenon is therefore a *theory* of that phenomenon:

> The model/theory myth occurs in computer science when the model/theory distinction is blurred so that programming a computer to generate a model of a given subject matter is taken as tantamount to producing a theory about the subject matter or at the very least an embodiment of a theory. (pp. 220–221)

One of his reasons for this conclusion that programs are not theories is that "The theory must be statable independently of the computer model" (p. 221). He has already marked the distinction between a theory, a program, and a model. And he allows that programs can be *models* (of the world) if they have a semantic interpretation in terms of the world.

But why couldn't a program also be a *theory*? Theories, for Moor, must explain and predict. Wouldn't a computer program be able to do that? Suppose that we want to have a scientific understanding of some portion of the real world in which we observe that certain causes always have certain effects. Suppose that we have a computer program in which computer analogs of those causes always computationally yield computer analogs of those effects. Would not the program itself explain how this occurs? And would we not be able to use the program to make predictions about future effects from future causes? More to the point, why would we need a *separate* ("independent") "set of laws" (presumably expressed in declarative sentences)?

Another reason that Moor offers for why a computer program that successfully models the real world is not thereby a theory of the world is that the program might be "*ad hoc*" (p. 221). He gives as an example Joseph Weizenbaum's Eliza program that simulates a Rogerian psychotherapist, not by embodying a theory of Rogerian psychotherapy, but by "superficial analysis of semantic and syntactic cues" (p. 221).

But all that that shows is that not all computer programs are theories. The question of whether a computer program *can* be a theory remains open.

There is still one more reason that he provides:

> The program will be a collection of *instructions* which are not true or false, but the theory will be a collection of *statements* which are true or false.
> (pp. 221–222, my italics)

And this, I think, is his real reason for arguing that programs are not theories: They are procedural, not declarative. They tell you how to *do* things, not how things *are*. But why must theories be declarative? Recall our earlier discussion in §3.14.4 of procedural vs. declarative language. There, we saw not only that those two kinds of language can (often) be intertranslatable, but—more to the point—that the statements of programming languages such as Prolog can be interpreted both procedurally *and* declaratively. Arguably, a declarative theory expressed in Prolog would also be a computer program.

15.5.2 Thagard's Objections

Thagard (1984, p. 77) argues that, on both the "syntactic" and the "semantic" conceptions of what a theory is, computer programs are not theories. He also argues that programs are not "models". Rather, "a program *is* a simulation *of* a model which approximates *to* a theory". So, on Thagard's view, we have:

R	=	some aspect of the real word
$T(R)$	=	a theory about R
$M(T(R))$	=	a model of T
$P(M(T(R)))$	=	a program that simulates M

On the syntactic theory of theories, a theory is a set of sentences (perhaps expressed as a formal system with axioms). On the semantic theory of theories, a theory is a "definition of a kind of system". Presumably, he will argue that a program is neither a set of sentences nor a definition (of a kind of system). He has not yet said what a "model" is.

Along with Moor, Thagard argues that, because programs are sets of *instructions*, which do not have truth values and hence are not sentences, programs cannot be theories in the syntactic sense (pp. 77–78). As we just saw, one question that can be raised about this is whether programs written in a programming language such as Prolog, whose statements can be interpreted as declarative sentences with truth values, could be considered to be theories. If so, then why couldn't any program that was equivalent in some sense to such a Prolog program also be considered a theory?

Another question that can be raised is this: Suppose that we are looking for a buried treasure. I might say, "I have a theory about where the treasure is buried: Walk three paces north, turn left, walk 5 paces, and then dig. I'll bet that you find the treasure." Is this not a theory? I suppose that Thagard might say that it isn't. But isn't there a sentential theory that is associated with it—perhaps something like "The treasure is buried at a location that is three paces north and 5 paces west of our current location". Doesn't my original algorithm for finding the treasure carry the same information as this theory, merely expressing it differently?

The argument that Thagard makes that programs can't be theories because they are not sets of declarative sentences just seems parochial. They are surely sets of (imperative) statements that have the additional benefit that they can become an instance of what they describe (alternatively: that they can control a device that becomes an instance of what they describe).

Thagard (1984, p. 78) considers a *model* to be "a set-theoretic interpretation of the sentences in a [syntactic] theory .... a system of things ... which provide an interpretation of the sentences". But "a program is not ... a system of things, nor does it ... provide an interpretation for anything". Hence, a program is not a model.

But a program being executed—a process—can be considered to be a system of (virtual) things that are interpretations of data structures in the program. If a *process* might be a *model*, then why couldn't the *program* be a *theory*?

On the semantic or "structuralist" (p. 78) view of theories, a theory "is a definition of a kind of natural system" (p. 79). Given some scientific laws (which, presumably, are declarative, truth-functional sentences, perhaps expressed in the language of mathematics), we would say that something is a certain kind of natural system "if and only if it is a system of objects satisfying" those laws. (The system is *defined* as being something that satisfies those laws.)

But this seems very close to what a model is. In fact, Thagard says that a "real system R is a system of the kind defined by the theory T" (p. 79). But how is that different from saying that R is an implementation of (that is, a model of) T?

Thagard's response is that, first, "a program *simulates* a system: it does not define a system" and that, second, programs contain "a host of characteristics which we know to be extraneous" to the real system that they are supposed to be like (p. 79). He says this because simulations aren't definitions: A *simulation* of the solar system, to use his example (recall our discussion of this in §9.8.2), doesn't *define* the solar system. This seems reasonable, but it also seems to support the idea that a process (not necessarily a program) is a model (and hence that a program would be a theory).

As for the problem of implementation-dependent details, Thagard says that

> if our program [for some aspect of human cognition] is written in LISP, it consists of a series of definitions of functions. The purpose of writing those functions is not to suggest that the brain actually uses them, but to simulate at a higher level the operation of more complex processes such as image ... processing.
> (p. 80, my interpolated brackets)

In other words, the real system that is being simulated (modeled?) by the program (process?) need not have Lisp functions. But, as he notes, it will have "complex processes" that do the same thing as the Lisp functions. But isn't this also true of any theory compared to the real system that it is a theory of? A theory of cognitive behavior expressed in declarative sentences will have, say, English words in it, but the brain doesn't. A similar point is made by Humphreys (1990, p. 501):

> Inasmuch as the simulation has abstracted from the material content of the system being simulated, has employed various simplifications in the model, and uses only the mathematical form, it obviously and trivially differs from the 'real thing', but in the respect, there is no difference between simulations and any other kind of mathematical model ....

Thagard does admit that he "shall take models to be like theories (on the semantic conception) as being definitions of kinds of systems" (p. 80). And he notes that "a model contains specifications which are known to be false of the target real system"—that is, implementation-dependent details! According to Thagard, the problem with this is that, if you try to make a prediction about the real system based on the model, then you might erroneously make it based on one of these implementation-dependent details (pp. 80–81). But that seems to be a problem endemic to any model (or any theory, for that matter).

If you make a prediction that turns out to be false, you may have to change your theory or your model. Perhaps you have to eliminate that implementation-dependent detail. But others will always crop up; otherwise, your theory or model will not merely describe or simulate the real system; it will *be* the real system. But there are well-known reasons why a life-sized map of a country is not a very good map! (We'll return to this in §17.3.2.)

However, Thagard's final summary is not really the wholesale rejection of programs as theories that it might first appear to be. It is more subtle:

> a program P, when executed on a computer, provides a simulation of a system of a kind defined by a model M, where M defines systems which are crude versions of the systems defined by a theory T, and the set of systems defined by T is intended to include the real system R. (p. 82)

I can live with this: The process and R are both implementations of T.

Further Reading:
Paul Humphreys (1990, 2002) discusses computer simulations and computer models.

Green 2001 analyzes the use of connectionist (or neural-network) computer programs as models of cognition, and argues that "Just because two things share some properties in common does not mean that one models the other. Indeed, if it did, it would mean that everything models everything else. There must be at least a plausible claim of some similarity in the *ways* in which such properties are realized in the model and the thing being modeled" (§IV, final paragraph).

Coward and Sun 2004 discusses a computational theory of consciousness, that is, a theory of consciousness that is implemented as a computer program.

Frigg et al. 2009 is a special issue of the philosophy journal *Synthese* on models and simulations.

15.6 Questions for the Reader

1. Must a syntactic theory be expressed in declarative sentences? (A declaritive sentence is a sentence that is either true or else false.)

2. Must a computer program be expressed in imperative language? (An imperative sentence is a sentence that says "Do this!"; it has no truth value.)

3. Must a semantic theory be a *set*-theoretic model of a real-world situation?

4. Could a computer *process*—that is, a program being executed—be a model of a real-world situation?

Chapter 16

Can Computer Programs Be Verified?

Version of 31 March 2019; DRAFT © 2004–2019 by William J. Rapaport

> Mechanical computers should, Babbage thought, offer a means to eliminate at a stroke all the sources of mistakes in mathematical tables. ... A printed record could ... be generated ..., thereby eliminating every opportunity for the genesis of errors. ... Babbage boasted that his machines would produce the correct result or would jam but that they would never deceive.
> —Doron D. Swade (1993, pp. 86–87)

> We talk as if these parts [of a machine] could only move in this way, as if they could not do anything else. How is this—do we forget the possibility of their bending, breaking off, melting, and so on? —Ludwig Wittgenstein (1958, §193, p. 77e)

> Present-day computers are amazing pieces of equipment, but most amazing of all are the uncertain grounds on account of which we attach any validity to their output. It starts already with our belief that the hardware functions properly.
> —Edsger W. Dijkstra (1972, p. 3)

> The history of program verification ... has now expanded to be about nothing less than the nature of the relationship between abstract logical systems and the physical world. —Selmer Bringsjord (2015, p. 265)

16.1 Readings:

1. Required:

   - Fetzer, James H. (1988), "Program Verification: The Very Idea", *Communications of the ACM* 31(9) (September): 1048–1063, https://pdfs.semanticscholar.org/712f/8ed1ba1ecec1c7f548fb094eacbb62cf5b40.pdf; reprinted in Colburn et al. 1993, pp. 321–358.
     - A highly controversial essay!

2. Very Strongly Recommended:

   (a) De Millo, Richard A.; Lipton, Richard J.; & Perlis, Alan J. (1979), "Social Processes and Proofs of Theorems and Programs", *Communications of the ACM* 22(5): 271–280, https://www.cs.umd.edu/~gasarch/BLOGPAPERS/social.pdf; reprinted in Colburn et al. 1993, pp. 297–319.

   - The equally controversial "prequel" to Fetzer 1988.

   (b) Ardis, Mark; Basili, Victor; Gerhart, Susan; Good, Donald; Gries, David; Kemmerer, Richard; Leveson, Nancy; Musser, David; Neumann, Peter; & von Henke, Friedrich (1989), "Editorial Process Verification" (letter to the editor, with replies by James H. Fetzer and Peter J. Denning), ACM Forum, *Communications of the ACM* 32(3) (March): 287–290.

   - The first of many sequels to Fetzer 1988. This one includes (1) a strongly worded letter to the editor of *CACM*, signed by 10 computer scientists, protesting the publication of Fetzer 1988; (2) a reply by Fetzer; and (3) a self-defense by the editor.

3. Strongly Recommended:

   (a) Any of the following 4 essays that started the current field of program verification. Hoare's is the most important.

   i. McCarthy, John (1963), "Towards a Mathematical Science of Computation", in C.M. Popplewell (ed.), *Information Processing 1962: Proceedings of DFIP Congress 62* (Amsterdam: North–Holland): 21–28, http://www-formal.stanford.edu/jmc/towards.html; reprinted in Colburn et al. 1993, pp. 35–56.

   ii. Naur, Peter (1966), "Proof of Algorithms by General Snapshots", *BIT* 6: 310–316; reprinted in Colburn et al. 1993, pp. 57–64.

   iii. Floyd, Robert W. (1967), "Assigning Meanings to Programs", in *Mathematical Aspects of Computer Science: Proceedings of Symposia in Applied Mathematics*, Vol. 19 (American Mathematical Society): 19–32, http://www.cs.tau.ac.il/~nachumd/term/FloydMeaning.pdf; reprinted in Colburn et al. 1993, pp. 65–81.

   iv. **Hoare, C.A.R. (1969), "An Axiomatic Basis for Computer Programming", *Communications of the ACM* 12: 576–580, 583**, https://www.cs.cmu.edu/~crary/819-f09/Hoare69.pdf; reprinted in Colburn et al. 1993, pp. 83–96.

(b) Dijkstra, Edsgar W. (1975), "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Communications of the ACM* 18(8): 453–457, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.97&rep=rep1&type=pdf

- An example of program verification in action.

(c) Gries, David (1981), *The Science of Programming* (New York: Springer-Verlag).

- A classic textbook. Contrast its title with that of Knuth's (1973) *The **Art** of Computer Programming*.

4. Recommended:

(a) Any of the following articles, which are representative of the battle that resulted from Fetzer 1988 and Ardis et al. 1989.

i. Pleasant, James C.; AND Paulson, Lawrence; Cohen, Avra; & Gordon, Michael; AND Bevier, William R.; Smith, Michael K.; & Young, William D.; AND Clune, Thomas R.; AND Savitzky, Stephen (1989), "The Very Idea" (5 letters to the editor), Technical Correspondence, *Communications of the ACM* 32(3) (March): 374–377.

ii. Fetzer, James H. (1989), "Program Verification Reprise: The Author's Response" (to the above 5 letters), Technical Correspondence, *Communications of the ACM* 32(3) (March): 377–381.

iii. Dobson, John; & Randell, Brian (1989), "Program Verification: Public Image and Private Reality", *Communications of the ACM* 32(4) (April): 420–422.

iv. Müller, Harald M.; AND Holt, Christopher M.; AND Watters, Aaron (1989), "More on the Very Idea" (3 letters to the editor, with reply by James H. Fetzer), Technical Correspondence, *Communications of the ACM* 32(4) (April): 506–512.

v. Hill, Richard; AND Conte, Paul T.; AND Parsons, Thomas W.; AND Nelson, David A. (1989), "More on Verification" (4 letters to the editor), ACM Forum, *Communications of the ACM* 32(7) (July): 790–792.

vi. Tompkins, Howard E. (1989), "Verifying Feature-Bugs" (letter to the editor), Technical Correspondence, *Communications of the ACM* 32: 1130–1131.

(b) **Barwise, Jon (1989), "Mathematical Proofs of Computer System Correctness", *Notices of the American Mathematical Society* 36: 844–851.**

- A cooler head prevails. An admirably clear and calm summary of the Fetzer debate.
- See also:
  Dudley, Richard (1990), "Program Verification" (letter to Jon Barwise (ed.), Computers and Mathematics column, with a reply by Barwise), *Notices of the American Mathematical Society* 37: 123–124.

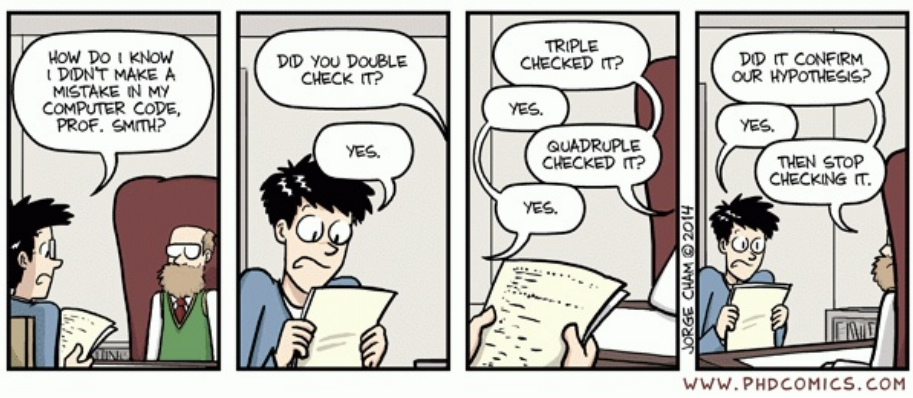


Figure 16.1: http://phdcomics.com/comics/archive.php?comicid=1693, ©2014, Jorge Cham

16.2 Introduction

The Halting Problem (§7.7) tells us that it is not possible to have a *single* computer program that can tell us in advance whether any given computer program will halt. However, given a *specific* computer program, there might be ways of determining whether that particular program will halt.

What about other problems that computer programs might have? It would be useful to be able to know in advance whether a given computer program will work. But what does it mean to say that a program "works"? It could mean that it successfully transforms its input into output in the sense that, when you start it up, it finishes. It could mean that it not only finishes, but has no logical "bugs"[1] (such as dividing by 0 or having an infinite loop) that would cause it to "crash". It could mean that it not only finishes without bugs, but yields the *correct* output. It could mean that it not only finishes without bugs or incorrect output, but also does what it was intended to do.

[1] On the history of the term 'bug', see Hopper 1981; Krebs and Thomas 1981. The idea that the first computer bug was really a bug (actually, a moth) is an urban legend, because the term was used in the non-entomological sense as early as 1875; see F.R. Shapiro 1985 and the *OED* online at http://www.oed.com/view/Entry/24352. For a photo of the allegedly first "bug", see http://www.catb.org/jargon/html/B/bug.html, which traces the term back to Shakespeare!

Digression, Questions for the Reader, and Further Reading:
On the TV show "Chopped" (a cooking competition), a chef is sometimes "chopped" (is eliminated from competition) because the chef tells the judges that the dish they just prepared is, say, a puttanesca, but the judge says that it isn't, on the grounds that it includes some ingredient that it shouldn't (or vice versa). Yet the dish might be delicious. Had the chef not said that it was a puttanesca, the chef might not have been chopped. Does it matter what the chef calls the dish? Is a delicious dish unsuccessful because it has a misleading name?

A similar problem can occur with computer programs Jeff Buechner argues that

> we cannot, by observing its output behavior, acquire the knowledge that a physical computer is operating normally, that it is correctly computing the values of a function F, that it is executing program P, and that it is using data structure D. ... [P]hysical computers can break down in various ways, and when they do, they might not physically realize the ... function which the computer would correctly compute if it did not break down. ... When a physical computer is functioning normally in the computation of the values of some function F, it will output the correct range value for F when it is given as input a domain value for F. However, there might be another function G which the same physical computer might be computing. When the physical computer is operating normally *in the computation of F* it is suffering a breakdown *in the computation of G.* By examining only its output behavior, one cannot determine whether it is operating normally (in the computation of F) or suffering a breakdown (in the computation of G). Similarly, by examining only its output behavior, one cannot determine whether it is operating normally (in the computation of G) or suffering a breakdown (in the computation of F). The problem is that what is breakdown behavior in the computation of F is normal behavior in the computation of G. Whether this physical computer is operating normally or suffering a breakdown is relative to which it is actually computing. —Buechner (2018, pp. 496–497)

Does this mean that we might not ever be able to decide if a computer is doing what it is "supposed" to be doing? Is this an even more serious problem than merely determining whether a program "works" (in any of the senses of 'works' that we just mentioned)? What if we are not limited to examining only the output? On the other hand, what if that is the only thing that *can* be examined (as might be the case with some "black box" machine-learning algorithms, as discussed in §3.9.5)?

Buechner (2011) also discusses this problem in connection with human cognition, arguing that the argument above, which he attributes to the philosopher Saul Kripke, also "rules out *any* scientific study of the mind that envisions it as an information-processing device, which is the core idea underlying cognitive science" (p. 362).

Many of these issues concern the relationship of mathematics to the real world (see §4.11, above, and Wigner 1960; Hamming 1980a) and also touch on ethical problems:

> 1. Many, if not most, errors in software engineering occur when bridging the gap between the informal, real world and the formal world of mathematical specifications. ...
> 2. Even if software engineers have a clear-cut specification of how they intend their software to behave, they will at best be able to prove that a mathematical model of their software satisfies this specification, not that the software will have the desired effects in the real world. —Daylight (2016, p. viii)

Is there a way to logically *prove* that a computer program "works"?

Recall our discussions in §§3.9.1 and 3.10 of the dual mathematical and engineering natures of CS: Is computer programming like mathematics? Or is it more like engineering? Many people identify computer programming with "software *engineering*". Yet many others think of a program as being like a *mathematical* proof: a formal structure, expressed in a formal language. For example, Peter Suber (1997a) compares programs to proofs this way: A program's *input* is analogous to the *axioms* used in a proof; the program's *output* is analogous to the *theorem* being proved; and the *program* itself is like the *rules of inference* that transform axioms into theorems, with the program transforming the input into the output.[2] Or perhaps a *program* is more like the endpoint of a proof, namely, a mathematical *theorem*. In that case, just as theorems can be proved (and, indeed, *must* be proved before they are *accepted*), perhaps programs can be proved (and, perhaps, *should* be proved before they are *used*). (We'll explore these analogies further in §16.3.2.)

Can we prove things about programs? What *kinds* of things might be provable about them? Two answers have been given to the first of these questions: yes and no. (Did you expect anything else?)

One of the most influential proponents of the view that programs *can* be the subjects of mathematical proofs is Turing Award winner C.A.R. (Tony) Hoare (the developer of the Quicksort sorting algorithm):

> Computer programming is an exact science in that all the properties of a program and all the consequences of executing it *in any given environment* can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. —Hoare (1969, p. 576, my italics)

> When the correctness of a program, its compiler, *and the hardware of the computer* have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with *a confidence limited only by the reliability of the electronics*.
> —Hoare (1969, p. 579, my italics)

> I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived *from their specifications* through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic.
> —Hoare (1986, p. 115, my italics)

Among those arguing that programs are *not* like mathematical proofs are the computer scientists Richard De Millo, Richard Lipton, and Alan Perlis:

> ... formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. —De Millo et al. (1979, p. 271)

[2] Suber makes a stronger claim, that computers are physical implementations of formal systems. This would require the rules of inference to satisfy the constraints of being an algorithm, but not all formal systems require that.

One reason that they give for this is their view that formal proofs are long and tedious, and don't always yield acceptance or belief. (My college friend Alan Arkawy always advised that you had to *believe* a mathematical proposition before you should try to *prove* it.) Hence, they argue, it is not worthwhile trying to formally verify programs.

The epigraph to their essay (slightly incorrectly cited) is the following quotation from J. Barkley Rosser's logic textbook:

> "I should like to ask the same question that Descartes asked. You are proposing to give a precise definition of logical correctness which is to be the same as my vague intuitive feeling for logical correctness. How do you intend to show that they are the same?" ... [T]he average mathematician ... should not forget that intuition is the final authority .... —Rosser (1978, pp. 4, 11)

Note the similarity to the Church-Turing Computability Thesis, which also states an equivalence between a "precise" notion and a "vague intuitive" one. This is not accidental: Recall from §7.4.2 that Rosser was one of Church's Ph.D. students and a pioneer in computability theory.

The "Descartes" mentioned in the passage is not the real Descartes, but a fictional version visited by a time-traveling mathematician who tries to convince him that the modern and formally "precise", ε-δ definition of a continuous curve is equivalent to the fictional Descartes's "vague intuitive" definition as something able to be drawn without lifting pencil from paper. Rosser observes that "the value of the ε-δ definition lies mainly in *proving* things about continuity and only slightly in *deciding* things about continuity" (p. 2, my italics). "Descartes" then says this to the time-traveling mathematician:

> "I have here an important concept which I call continuity. At present my notion of it is rather vague, not sufficiently vague that I cannot decide which curves are continuous, but too vague to permit of careful proofs. You are proposing a precise definition of this same notion. However, since my definition is too vague to be the basis for a careful proof, how are we going to verify that my vague definition and your precise definition are definitions of the same thing?" —Rosser (1978, p. 2)

The time traveler and "Descartes" then agree that, despite the informality of one definition and the formality of the other, the two definitions can be "verified"—but not "proved"—to be equivalent by seeing that they agree on a wide variety of cases. When the mathematician returns to the present, a logician points out that the mathematician's intuitive notion of proof bears the same relation to the logician's formal notion of proof as the fictional Descartes's intuitive notion of continuity bears to the mathematician's formal definition of continuity. The passage that De Millo et al. (1979, p. 271) quote is the mathematician's response to the logician in the story. (It is the real logician Rosser, in his own voice, who comments that "intuition is the final authority"!)

So, Hoare says that programs *can and should* be formally verified. DeMillo et al. (and Rosser, perhaps) suggest that they can, but *need not* be. Along comes the philosopher James Fetzer, who argues that they *cannot*. More precisely, he argues that the things that we *can* prove about programs are not what we think they are:

> … there are reasons for doubting whether program verification can succeed as a generally applicable and completely reliable method for guaranteeing the performance of a program. —Fetzer (1988, p. 1049)

We'll come back to DeMillo et al. in §16.4.3. But, first, what does it mean to formally verify a program? Before we can answer that, we need to be clear about what it means to verify—that is, to prove—a *theorem*.

Further Reading:
Dijkstra 1972, pp. 9–11 has a discussion about the nature and value of formal proofs of "obvious" theorems that is relevant to the issues raised by De Millo et al. (1979) and Rosser's time-traveler.

Devlin 1992 (which serves as an introduction to MacKenzie 1992) contains a dialogue between a math professor who defends De Millo et al. (1979)'s notion of "social" proof and a philosophy student who defends the more formal, logical notion of proof.

For an interesting contrast to Suber (1997a) on the relationship between computer programs and mathematical proofs, see Thurston 1994.

Davis and Hersh 1998, ("The Ideal Mathematician", pp. 34–44, http://babel.ls.fi.upm.es/~pablo/Jumble/ideal.pdf) is a partly facetious description of the behavior of (some) mathematicians, including a discussion of the nature of proof as carried out by mathematicians. For an antidote to their characterization of mathematicians, read Frenkel 2013.

16.3 Theorem Verification

16.3.1 Theorems and Proofs

A formal proof in logic or mathematics can be thought of as a sequence of propositions. A **proposition** is what computer scientists call a "Boolean statement", that is, a statement that is either true or else false. The proofs themselves (the *sequences* of propositions beginning with axioms and ending with a theorem) are not Boolean-valued: They are neither "true" nor "false"; rather, they are either "correct" (the technical terms are 'valid' and 'sound') or "incorrect" (technically, 'invalid' or 'unsound').

16.3.1.1 Syntax

However, the actual truth values of the propositions in a proof are irrelevant to the structure of the proof. From a purely syntactic point of view, a **proof** of a theorem T has the general form:

$$\langle A_1, \ldots, A_n, P_1, \ldots, P_m, C_1, \ldots, C_l, T \rangle$$

where:

- T—the last item in the sequence of propositions that constitutes the proof—is the **theorem** to be proved.

- The A_i are **axioms**, that is, propositions that are "given" or "assumed without argument". They are the starting points—the "basic" or "primitive" propositions—of any proof, no matter what the subject matter is. From a strictly syntactic point of view, the axioms of a formal system need not be (semantically) "true". (For example of "axioms" that are not "true", see some of the formal systems that we mentioned in the Further Reading box in §14.3.2.2.)

- The P_j are **premises**, that is, propositions about some particular subject matter that is being formalized. They are also "starting points", though only for the current topic of the proof.

- The C_k are propositions that logically follow from previous propositions in the sequence by a "rule of inference". A rule of inference can be thought of as a "primitive proof" in the same sense as the "primitive" operations of a Turing machine or the "basic" functions in the definition of recursive functions. A **rule of inference** has the form:

  From propositions $Q_1, \ldots, Q_r$
  you may infer proposition R

  (See §2.6.1.1 for an example.) Just as with axioms, the rules of inference are given by fiat. The rules of inference are *syntactically valid* by definition. Note that if $n = 0$ (that is, if there are *no* axioms, and, especially, if $m = 0$ also—that is, if there are no premises), then there will typically have to be *lots* of rules of inference, and the demonstration is then said to be done by "natural deduction" (because it is the way that logicians "naturally" prove things—see Pelletier 1999 for a history of natural deduction).

A more complex proof is then recursively defined in terms of successive applications of rules of inference. Then, to say that the sequence

$$\langle A_1, \ldots, A_n, P_1, \ldots, P_m, C_1, \ldots, C_l, T \rangle$$

is a proof of T from the axioms and premises means (by definition) that both each C_k and T follow from previous propositions in the sequence by a (syntactically valid) rule of inference. A proof is **syntactically valid** iff the final conclusion T and every intermediate conclusion C_k results from a correct application of a rule of inference to preceding propositions in the proof.

16.3.1.2 Semantics

What about truth? Surely, we want our theorems to be true! From a semantic point of view, the axioms are typically considered to be *necessarily* true by virtue of their meanings (or assumed to be true for the time being); they are usually logical tautologies. Premises, on this account, are *contingently* or *empirically assumed* to be true (but they would normally require some justification). For example, often you need to justify a premise P_j by providing a *proof* of P_j or at least some empirical evidence in its favor. Just as, normally, we want our axioms and premises to be (semantically) *true*,

so, normally, we want our rules of inference to be (semantically) *truth-preserving*. A rule of inference of the form

> From propositions $Q_1, \ldots, Q_r$
> you may infer proposition R

is **truth-preserving** $=_{def}$ *if* each of $Q_1, \ldots, Q_r$ is true, then R is true. This does not mean that R *is* true; all it means is that R is true *relative to* $Q_1, \ldots, Q_r$. As we saw with axioms, from a purely syntactic point of view, rules of inference do not *have* to be truth-preserving; again, see some of the formal systems cited in §14.3.2.2. A truth-preserving rule of inference is said to be **semantically valid**. In order for theorem T to be *true*, each rule of inference (and therefore the entire proof) must be semantically valid, that is, it must be truth-preserving: Its conclusion *must* be true *if* its axioms and premises are true. But, of course, the axioms and premises of an argument might *not* be true. If they *are* true, *and if* the argument is semantically valid, then its conclusion *must* be true. Such a truth-preserving argument with true premises is said to be **sound**; it is *unsound* iff either one or more of its axioms or premises is false or it is *syntactically* invalid. Roughly, a syntactically valid argument that is unsound because of a false axiom or premise is like a correct program whose input is "garbage"; the output of such a program is also "garbage" (this is the famous saying: "garbage in, garbage out"). And the final conclusion of a syntactically valid but semantically invalid proof need not be true.

But, for a syntactically valid proof to also be *semantically valid*, the rules of inference must be truth-preserving. And, for it to be *sound*, the axioms must be true. (For more on this, recall §2.6.1.1, see Appendix A.)

It is strictly speaking incorrect to say that a theorem is "*proved* to be *true*". 'Theorem' is a syntactic notion, while 'truth' is a semantic notion. Theorems do not have to be true: A syntactically valid proof that began with a false axiom or a false premise might end with a theorem that is also false (or it might end with a theorem that *is* true!). And a truth need not be a theorem: Gödel's Incompleteness Theorem shows that there are true propositions of arithmetic that are not provable. The conclusion of any formal proof—that is, any theorem—is only true *relative to* the *axioms* (and premises) of the formal theory (Rapaport, 1984b, p. 613). Of course, *if* all of the axioms (and premises) *are* true, *and if* all of the rules of inference *are* truth-*preserving*, then the theorem that has been (syntactically) proved will be (semantically) true. (We'll come back to this point in §16.5.1.)

16.3.2 Programs and Proofs

How is all of this applicable to *programs*? That depends on how the *logical* paraphernalia (axioms, rules of inference, proofs, theorems, etc.) line up with the *computational* paraphernalia (specifications, input, programs, output, etc.). Several different analogies can be made:

A1: Fetzer's and Suber's Analogy (Fetzer 1988, p. 1056, col. 1; Suber 1997a)

Logic		Computation
axioms, premises	≈	input
rules of inference	≈	program
theorem	≈	output

On this analogy, verifying a *program* would be like proving the *inference rules*! But inference rules are not propositions, so they can't be proved. So, a slight modification of this is analogy A2:

A2

Logic		Computation
axioms, premises	≈	input
intermediate conclusions	≈	program
theorem	≈	output

Both intermediate conclusions and program are a sequence (or at least a set) of expressions that begin with something given and that end with a desired result. But you are trying to *prove* the theorem; you are not trying to prove the intermediate conclusions. And you *verify* an entire program, not just its output. So, let's consider analogy A3:

A3: Scherlis & Scott's Analogy (Scherlis and Scott, 1983, p. 207)

Logic		Computation
problem	≈	specification
proof	≈	program derivation (or verification)
theorem	≈	program

On this analogy, proving a theorem is like verifying or deriving a program. But the role of axioms (and premises) and rules of inference are not made clear. So, let's try a slightly different modification of A2:

A4

Logic		Computation
axioms, premises	≈	input
proof	≈	program
theorem	≈	output

Just as, when you prove (or derive) a theorem, you transform the axioms into the theorem, so a program transforms input into output. To verify a program is to prove that it will, indeed, transform the input into the expected output—that is, that it will satisfy its specification. This analogy seems closer to what program verification is all about.

A5 There is one more analogy: The idea behind this one is that a theorem usually has the form "if antecedent A, then consequent C". And most programs can be put into the form "if you execute program P, then you will accomplish goal G". So, proving a theorem is analogous to verifying that P accomplishes G. One issue that this analogy highlights is whether the *goal* of a program is an essential part of it. We'll return to this issue in Chapter 17.

Further Reading:
Scherlis and Scott (1983, p. 207) say that all such analogies must "be taken with a big grain of salt, since all these words can mean many things".

Avigad and Harrison 2014 discusses the history and nature of formal proofs in math, and then turns the relationship between theorem proving and program verification around: "With the help of computational proof assistants, formal verification could become the new standard for rigor in mathematics" (from the introductory blurb, p. 66).

16.3.3 Programs, Proofs, and Formal Systems

There is another way to think about rules of inference that clarifies the relationship between programs, proofs, and formal systems.

First, consider Gödel's observation about the importance of Turing's analysis of computation. (Although here I only want to focus on Gödel's first sentence, it is worth quoting the rest because of his observations on other topics that we have discussed.)

> *[D]ue to A.M. Turing's work, a precise and unquestionably adequate definition of the general concept of formal system can now be given* .... Turing's work gives an analysis of the concept of "mechanical procedure" (alias "algorithm" or "computation procedure" ...). this concept is show to be equivalent with that of a "Turing machine". A formal system can simply be defined to be any mechanical procedure for producing formulas, called provable formulas. For any formal system in this sense there exists one in the sense ... [of "a system of symbols with rules for employing them"] that has the same provable formulas (and likewise vice versa) ... . [The "essence" of] the concept of formal system ... is that reasoning is completely replaced by mechanical operations on formulas. (Note that the question of whether there exist finite non-mechanical procedures not equivalent with any algorithm, has nothing whatsoever to do with the adequacy of the defiition of "formal system" and of "mechanical procedure".) —Gödel (1964, pp. 71–72)

In what sense is a Turing machine the same as a formal system?

As we have seen, a rule of inference tells you what kind of proposition can be inferred from other kinds. So, for instance, the rule *modus ponens* (also sometimes called "$\rightarrow$ elimination") tells us that a proposition of the form Q (that is, any proposition whatsoever) can be inferred from propositions of the forms $(P \rightarrow Q)$ and P. And the rule of *addition* (sometimes also called "and-introduction" or "$\wedge$ introduction") tells us that a proposition of the form $(P \wedge Q)$ can be inferred from any two propositions P and Q.

Each of these can also be thought of as functions: Modus ponens is the function $\text{MP}((P \rightarrow Q),\ P) = Q$; addition is the function $\text{ADD}(P,\ Q) = (P \wedge Q)$. A proof of a

theorem T from axioms (and premises) $A_1, \ldots, A_n$ can then be thought of as successive applications of such inference-rule functions to the axioms and to the previous outputs of such applications. Are these functions (these rules of inference) computable? If so, then a proof can be thought of as a kind of program.

Further Reading:
Haugeland 1981b is a good description of the syntax and semantics of formal systems and their relationship to Turing Machines.

Exercise for the Reader:
Show that a typical rule of inference is, indeed, computable. Hint: Can you write a Turing-machine program that has the inputs to a rule of inference encoded on its tape, and that, when executed, has the output of the rule encoded on the tape?

But this raises an interesting question: Do any of our analogies capture this relationship? Is verifying a program really like proving a theorem? The relationship I have just outlined suggests that, if a program is like a proof, then verifying a program is actually more like checking a proof to show that it is syntactically valid. To check a proof for validity is to check whether each proposition in the sequence of propositions that constitutes the proof is either a basic proposition (an axiom) or else follows from previous propositions in the sequence by a rule of inference (by the application of an inference-rule function). In verifying a program of the form

$$\{P_1\}\ S_1\ \{Q_1\}, \ldots, \{P_n\}\ S_n\ \{Q_n\}$$

P_1 plays the role of an axiom, and each proposition of the form

> If P_i is the case, and if S_i is executed, then Q_i is the case

plays the role of an application of a rule of inference to the "inputs" P_i and S_i. The final state of the computation—Q_n— plays the role of the theorem to be proved. Any theorem is really of the form $(A_i \rightarrow T)$ (if the axioms are the case, then the theorem is the case). Similarly, a program can be thought of as taking the form "If P_1 is input, then Q_n is output", which is a high-level specification of the program. To verify the program is to check that it satisfies the specification.

This gives us a refinement of analogy **A4**, above:

A4′

Logic		**Computation**
axioms	$\approx$	input
rules	$\approx$	computable functions
theorem	$\approx$	output
proof	$\approx$	program
valid proof	$\approx$	verified program

16.4 Program Verification

16.4.1 Introduction and Some History

"Program verification" is a subdiscipline of CS. It can be thought of as theoretical software engineering, or the study of the logic of software. It is also a subissue of the question concerning the relation of software to hardware that we looked at in §12.4.

But it is not a new idea: Nowadays, we think of Euclidean geometry as a formal axiomatic system in which geometric *theorems* are stated (in *declarative* language) and proved to follow logically from the axioms. However, as we saw in §3.14.4, each proposition of Euclid's original *Elements* actually consisted of an *algorithm* (expressed in a *procedural* language for constructing a geometric figure using only compass and straightedge) and a *proof of correctness* of the algorithm—that is, a "verification" that the compass-and-straightedge "program" actually resulted in a geometic figure with the desired properties. (See, for example, the statement of Euclid's Proposition 1 at http://tinyurl.com/kta4aqh.[3])

Similarly, a program verification typically consists of taking an algorithm expressed in a (procedural) language for computing a function using only primitive computable (that is, recursive) operations (as in §7.6), and then providing a proof of correctness of the algorithm—that is, a verification that the algorithm satisfies the input-output specification of the function (as in analogies A3 and A4).

In practice, there is a preliminary step, which will occupy us for much of Chapter 17: Typically, one begins with a *problem* (or "goal"), perhaps informally stated, that is then formally modeled by a function. So, another possible goal of program verification might be to show that the program that implements the *function* actually solves the *problem* (as in analogy A5).

Another historical antecedent—perhaps the earliest example of program verification—is due to Turing himself. His 1949 essay ("Checking a Large Routine") "is remarkable in many respects. The three ... pages of text contain an excellent motivation by analogy, a proof of a program with two nested loops, and an indication of a general proof method very like that of Floyd [1967]" (Morris and Jones, 1984, p. 139).

[3] http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3Atext%3A1999.01.0086%3Abook%3D1%3Atype%3DProp%3Anumber%3D1

16.4.2 Program Verification by Pre- and Post-Conditions

The idea behind program verification is to augment, or annotate, each statement S of a program with:

1. a proposition P expressing a "pre-condition" of executing S, and
2. a proposition Q expressing a "post-condition" of executing S.

A **pre-condition** P of a program statement S is a description (of a situation, either in the world in which the program is being executed or in the computer that is executing the program) that must be true in order for S to be able to be executed; that is, P must be true *before* S can be executed. (And, according to the correspondence theory of truth (§2.4.1), P will be true iff the situation that it describes "exists", that is, really is the case.)

A **post-condition** Q of S is a description (of a situation) that will necessarily be true *after* S is executed. That is, the situation that Q describes will come into "existence" (come to be the case) after S is executed: S changes the computer (or the world) such that Q becomes true.

So, such annotations describe both how things *must* be if *S is to be* executed successfully and how things *should* be if *S has been* executed successfully. They are typically written as *comments* preceding and following S in the program. Letting comments be signaled by braces, the annotation would be written as follows:

$$\{P\}\ S\ \{Q\}$$

Such an annnotation is semantically interpreted as saying:

> **If** P correctly describes the state of the computer (or the state of the world) *before* S is executed,
>
> **and if** S is executed,
>
> **then** Q correctly describes the state of the computer (or the state of the world) *after* S is executed.

The "state of the computer" includes such things as the values of all registers (that is, the values of all variables).

So, if we think of a program as being expressed by a sequence of executable statements:

> **begin** $S_1; S_2; \ldots; S_n$ **end.**

then the program annotated for program verification will look like this:

> **begin** $\{I \ \&\ P_1\}S_1\{Q_1\};\ \{P_2\}S_2\{Q_2\}; \ldots; \{P_n\}S_n\{Q_n\ \&\ O\}$ **end.**

where:

- I is a proposition describing the input,
- P_1 is a proposition describing the initial state of the computer (or the world),

- For each i, Q_i logically implies P_{i+1}. (Often, $Q_i = P_{i+1}$.)
- Q_n is a proposition describing the final state of the computer (or the world), and
- O is a proposition describing the output.

The claim of those who believe in the possibility of program verification is that we can then logically prove whether the program does what it's supposed to do *without having to run the program*. We would construct a proof of the program as follows:

premise:	The input of the program is I.
premise:	The initial state of the computer is P_1.
premise:	If the input is I and the initial state is P_1, and if S_1 is executed, then the subsequent state will be Q_1.
premise:	S_1 is executed.
conclusion:	$\therefore$ The subsequent state is Q_1.
premise:	If Q_1, then P_2.
conclusion:	$\therefore P_2$.
premise:	If the current state is P_2, and if S_2 is executed, then the subsequent state will be Q_2.
conclusion:	$\therefore$ The subsequent state is Q_2.
...	...
conclusion:	$\therefore$ The final state is Q_n.
premise:	If Q_n, then O.
conclusion:	$\therefore O$.

The heart of the proof consists in verifying each premise. If the program isn't a "straight-line" program such as this, but is a "structured" program with separate modules, then it can be recursively verified by verifying each module (Dijkstra, 1972).

16.4.3 The Value of Program Verification

If debugging a program by running it, and then finding and fixing the bugs, is part of *practical* software engineering, then you can see why program verification can be thought of as *theoretical* software engineering.

One reason why program verification is argued to be an important part of software engineering is that this annotation technique can also be used to help *develop* programs that would thereby be guaranteed to be correct. Dijkstra (1975b) shows how to "formally derive" a program that satisfies a certain specification. And Gries 1981 is a textbook that shows how to use logic to "develop" programs simultaneously with a proof of their correctness.

Scherlis and Scott (1983) argue that program verification and development should go hand-in-hand, rather than verification coming *after* a program is complete. Their notion—"inferential programming"—differs from "program derivation": Whereas "program derivations [are] highly structured justifications for programs[,] inferential programming [is] the process of building, manipulating, and reasoning *about* program derivations" (p. 200, my interpolated brackets and italics). A " 'correctness' proof [shows] that a program is consistent with its specifications" (p. 201), where "*Specifications* differ from programs in that they describe aspects or restrictions on the functionality of a desired algorithm without imposing constraints on *how* that functionality is to be achieved" (p. 202). For instance, a specification might just be an input-output description of a function. To prove that a program for computing that function is "correct"—that is, to "verify" the program—is to prove that the program has the same input-output behavior as the function.

Scherlis and Scott (1983, p. 204) take issue with De Millo et al. (1979). First, they observe that the claim that "Mathematicians do not really build formal proofs in practice; why should programmers?" is fallacious, because "formalization plays an even more important rôle in computer science than in mathematics", and this, in turn, because "computers do not run 'informal' programs". Moreover, formalization in mathematics has made possible much advancement independent of whether "there is any sense in looking at a *complete* formalization of a whole proof. Often there is not."

They advocate, not for a complete proof of correctness of a completed program, but for proofs of correctness of stages of development, together with a justification that "derivation steps *preserve* correctness". This is exactly the way in which proofs of theorems are justified: If the axioms and premises are true, and if the rules of inference are truth-preserving, then the conclusions (theorems) will be true (relative to the truth of the axioms and premises).

Further Reading:

In Chapter 3, we considered some of Dijkstra's positions on the nature of CS. For some other things he has had to say about program verification, see Dijkstra 1974, in which he argues "that the correctness of programs could and should be established by proof", that structured programs are simpler to prove than unstructured ones (Dijkstra, 1968), that theorems about programs make program proofs easier, and that "to prove the correctness of a given program was ... putting the cart before the horse. A much more promising approach turned out to be letting correctness proof and program grow hand in hand" (pp. 609–610). See also Dijkstra 1983.

Verity 1985 contains interviews with Dijkstra, Hoare, and Gries, and includes a copy of Dijkstra 1983.

Mili et al. 1986, §1, contains a formal presentation of program correctness.

Tam 1992 and Arnow 1994 show how to use program-verification techniques to develop programs.

16.5 The Fetzer Controversy

> In many creative activities the medium of execution is intractable. Lumber splits; paint smears; electrical circuits ring. These physical limitations of the medium constrain the ideas that may be expressed, and they also create unexpected difficulties in the implementation. —Frederick P. Brooks (1975, p. 15)

> The transition function for a finite-state automaton specifies everything there is to know about it. From this it does not follow that we know everything about the behavior of a PCM [physical computing machine] that physically realizes the abstract diagram of a finite-state automaton, since the physical realization may be imperfect. —Jeff Buechner (2011, p. 349)

16.5.1 Fetzer's Argument against Program Verification

Nonsense!, said philosopher James H. Fetzer (1988), thus initiating a lengthy controversy in the pages of the *Communications of the ACM* and elsewhere. Several strongly worded letters to the editor chastised the editor for publishing Fetzer's paper; supportive letters to the editor praised the decision to publish; and articles in other journals attempted to referee the publish-or-not-to-publish controversy as well as the more substantive controversy over whether programs can, or should, be verified.

What did Fetzer say that was so controversial? Here is the abstract of his essay:

> The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.
> —Fetzer (1988, p. 1048)

Despite the analogies between proofs of theorems and verifications of programs, Fetzer focuses on one significant *dis*analogy, which he expresses in terms of a difference between "algorithms" and "programs": *Algorithms*, for Fetzer, are abstract, formal (mathematical or logical) entities; *programs*, for Fetzer, are physical ("causal") entities (Fetzer, 1988, pp. 1052, note 6; 1056, col. 2; and §"Abstract Machines versus Target Machines" (pp. 1058–1059)).

A "program", for Fetzer is a "causal model of" an algorithm (p. 1048), an "implementation of an algorithm in a form that is suitable for execution by a machine" (p. 1057, col. 2). In other worlds, whereas an "algorithm" (in Fetzer's terminology) is a formal entity susceptible to logical investigation, a "program" is a real-world, physical object that is *not* susceptible to *logical*—but only *empirical*—investigation. The analogies we discussed in §16.3.2 hold for "algorithms", but not for "programs" in Fetzer's senses. (Fetzer prefers A1 to A3; Fetzer 1988, p. 1056, col. 2.)

The computer historian Edgar G. Daylight (2016, p. 97), makes a similar distinction between a "mathematical program" and a "computer program": The former is an algorithm expressed in a formal language; the latter "resides electronically in a specific computer and is what most of us would like to get 'correct' ". (You should also

keep in mind the difference between a static "computer program [that] resides electronically in a computer"—perhaps as a specific arrangement of switch-settings—and the dynamic *process*, that is, the actually running program.) Even the very "same" program as implemented in *text* or as implemented in a *computer* might have different behaviors depending on how its numerical-valued variables are interpreted: A "mathematical program" for computing the square root of an integer can be "correct" to any decimal place, whereas the program implemented in a computer can only have a finite accuracy; yet, in a perfectly reasonable sense, they are the "same" program (Dijkstra 1972, §6; Daylight 2016, p. 102).

As Fetzer (1988, p. 1059, col. 1) observes, algorithms (Fetzer's terminology) or mathematical programs (Daylight's terminology) "can be conclusively verified, but ... [this] possesses no significance at all for the performance of any physical system", whereas "the performance of" programs (Fetzer) or computer programs (Daylight) "possesses significance for the performance of a physical system, but it cannot be conclusively verified". Fetzer (1988, p. 1060, col. 1) quotes Einstein (1921):

> As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.

Recall Chomsky's competence-performance distinction from §10.4.1: Even if program-verification techniques can prove that a program is correct ("competent"), there may still be *performance* limitations. The point, according to Fetzer, is that we must distinguish between the *program* and the *algorithm* that it implements. A program is a *causal* model of a *logical* structure, and, although *algorithms* might be capable of being absolutely verified, *programs* cannot.

Consider program statements that specify physical output behaviors. For example, some programming languages have a command BEEP whose intended behavior is to ring a bell. Or suppose that you have a graphical programming language one of whose legal instructions is DRAW_CIRCLE(x, y, r), whose intended behavior is to draw a circle at point (x, y) with radius r. How can you prove or verify that the program will ring the bell or draw the circle? How can you mathematically or *logically* prove that the (physical) bell will (actually) ring or that a (physical) circle will (actually) be drawn? How can you *logically* prove that the (*physical*) bell *works* or that the pen has ink in it? Fetzer's point is that you can't. And the controversy largely focused on whether that's what's meant by program verification. Recall our discussion in Chapter 10 about Cleland's interpretation of the Church-Turing Computability Thesis: Is preparing hollandaise sauce, or physically ringing a bell, or physically drawing a circle a computable task?

But, according to Fetzer, it's not just real-world output behaviors like ringing bells, drawing circles, or, for that matter, cooking that's at issue. What about the mundane PRINT command? According to Fetzer, it's not just a matter of causal output, because you can replace every PRINT(x) command with an assignment statement: $p := x$. *But even this is a causal statement*, because it instructs the computer to *physically* change the values of bits in a *physical* register p, and so Fetzer's argument goes through: How can you *logically* prove that the *physical* computer will actually work? Indeed, the history of early modern computers was largely concerned with ensuring that the vacuum

tubes would be reliable (Dyson, 2012b). Recall Babbage's boast and Wittgenstein's warning, cited in the epigraph to this chapter.

In Fetzer's terminology, a theorem T is "absolutely verifiable" $=_{def}$ T follows only from (logical) axioms (and not from empirical premises), and T is "relatively verifiable" $=_{def}$ T follows from (logical) axioms *together with* (empirical) premises. That is, T is "relatively" verifiable iff it is a logical consequence of some of the premises about the particular subject matter; it is "verifiable relative to" the premises. As Donald MacKenzie puts it,

> ... mathematical reasoning alone can never establish the "correctness" of a program or hardware design in an absolute sense, but only relative to some formal specification of its desired behavior. —MacKenzie (1992, p. 1066, col. 2)

Although an "absolutely verifiable" theorem T is not relative to the *premises*, even what Fetzer calls '*absolute* verifiability' is still a kind of *relative* verifiability, except that the verifiability is relative to the *axioms* (not to the premises), as we saw in §16.3.1.

Given all of this terminology, Fetzer phrased the fundamental question of program verification this way: *Are programs absolutely verifiable?* That is, can programs be verified directly from axioms, with no empirical premises? (One question that you should keep in mind as you read the papers involved in this controversy is this: *Do* the pro-verificationists claim that programs are absolutely verifiable, in Fetzer's terminology?)

To be "absolutely verifiable" requires there to be *program* rules of inference that are truth-preserving, or it requires there to be *program* axioms that are necessarily true about "the *performance* that a machine will display when such a program is executed" (Fetzer, 1988, p. 1052, my italics). Verification that requires axioms about *performance* is different from program verification in the Hoare-Dijkstra-Gries tradition, because of a difference between *logical* relations and *causal* relations, according to Fetzer. The former are abstract; the latter are part of the real world. It might be replied, on behalf of the pro-verificationists, that we can still do *relative* verification: verification relative to "causal axioms" that relate these commands to causal behaviors. So, we can say that, *if* the computer executing program P is in good working order, and *if* the world (the environment surrounding the computer) is "normal", *then* P is verified to behave in accordance with its specifications.

No, says Fetzer: Algorithms and programs that are only intended for *abstract* machines *can* be *absolutely* verified (because there is nothing physical about such machines; they are purely formal). But programs that can be compiled and executed on *physical* machines can only be *relatively* verified.

Further Reading:
MacKenzie 1992 discusses a legal challenge to a claim that a certain computer program had been verified. The claim was that the verification was of the relatively informal, De Millo et al. 1979-style variety of proof; the challenge was that a formal, mathematical proof was necessary.

16.5.2 The Controversy

The reaction to Fetzer's paper was explosive, beginning with a letter to the editor signed by 10 distinguished computer scientists arguing that it should never have been published, because it was "ill-informed, irresponsible, and dangerous" (Ardis et al., 1989, p. 287, col. 3)! The general tone of the responses to Fetzer also included these objections:

- So what else is new? We program verificiationists never claimed that you could logically prove that a physical computer would not break down.
- Verification techniques *can* find logical faults; it is logically possible to match a program or algorithm to its specifications.
- You can minimize the number of rules of the form "input *I causes* output *O*" such that they only apply to descriptions of logic gates and the physics of silicon.
- Many programs are incorrect because, for example, of the limits of accuracy in using real numbers.
- Verifiably *incorrect* programs can be better than verifiably correct programs *if* they have better average performance. (Moor 1979 makes a similar argument in the context of whether we should trust decisions made by computers; we'll discuss this in Chapter 18.

Let's look at some of these.

Ardis et al. (1989) claimed that program verification was *not* supposed to "provide an *absolute* guarantee of correctness with respect to the execution of a program on computer hardware" (p. 287, col. 1, my italics). This is interestingly ambiguous: On one reading, they might have been claiming that program verification only provides a *relative* guarantee of correctness; if so, they are actually in agreement with Fetzer! On another reading, they might have been claiming that program verification *does* provide an absolute guarantee, but not of *hardware* execution; if so, that is also consistent with Fetzer's arguments!

They also claimed that it was not the case that "verification can be applied only to abstract programs written in high level languages" (Ardis et al., 1989, p. 287, col. 2). For example, they said, it *can* be applied to assembly languages, contrary to what Fetzer (1988, p. 1062, col. 2) claimed. But Fetzer didn't have to claim that: There can be abstract, formal assembly languages. What Fetzer perhaps should have said was that program verification cannot be applied to assembly-language programs that "reside electronically in a computer" (to use Daylight's characterization). As Parsons (1989, p. 791, col. 1) later observed, their "rage" might have been indicative of a lack of evidence for their belief.

Other critics responded to Fetzer's paper by saying "So what else is new?": Pleasant (1989, p. 374, col. 1, my italics) observed that Fetzer's complaint "*belabor[s] the rather obvious fact* that programs which are run on real machines cannot be completely reliable, *as though advocates of verification thought otherwise.*"

And Paulson et al. (1989, p. 375, col. 1, my italics) said that "Fetzer makes one important *but elementary* observation and takes it to an absurd conclusion. ... [M]ost

systems ... do not need to work perfectly. ... A physical fault can usually be repaired quickly, replacing the damaged part; then the job can be run again." It is interesting to note, especially in connection with topics that we will look into in Chapter 17, that the passage that I omitted after "most systems" concerned one major exception: SDI—the Strategic Defense Initiative—a program to defend the US using a computer-controlled missile defense system; that system, of course, needed to "work perfectly"! (On program verification of SDI, see Myers 1986.)

Or this from Holt (1989):

> No one expects a computer to work properly if someone pulls the plug out [p. 508, col. 2]. ... Errors in programs due to inaccurate scientific theories, omissions in specifications, and implementation failures are inevitable; those due to programming mistakes should not be [p. 509, cols. 1–2].

An interesting variation on this came from Conte (1989), who called "Fetzer's article ... an over-inflated treatment of a principle most children learn by the age of ten—no matter how perfect your cookie recipe is, if the oven thermostat fails, you may burn the cookies". How do you think Carol Cleland, whose objections to the Computability Thesis we examined in Chapter 10, would respond if she were told that, no matter how perfect her Hollandaise-sauce recipe is, if it is prepared on the Moon, it may not work?

16.5.3 Barwise's Attempt at Mediation

The logician Jon Barwise (1989b) attempted to mediate the controversy. In doing so, he also discussed many other issues that we have looked into (or will in future chapters), including the relation between algorithms and programs, the possibility of finding fault with an argument yet believing its conclusion (see Appendix A of this book), the nature of "philosophy of *X*" (see §2.8 of this book), and the difference between the truth of a premise and agreeing with it (see Appendix A of this book).

Barwise saw the issue between Fetzer and his opponents as being a special case of the more general question of how mathematics can be applied to the real world, given that the former is abstract and purely logical, whereas the latter is concrete and empirical (p. 846, col. 2). (See also Wigner 1960 and §§4.11 and 17.9.) But there is another aspect to that issue in the philosophy of math, namely, the relation between the syntax of a formal mathematical expression and its semantic interpretation in the real world:

> The axiomatic method says that our theorems are true *if* our axioms are. The modeling method says that our theorems model facts in the domain modeled *if* there is a close enough fit between the model and the domain modeled. The sad fact of the matter is that **there is usually no way to prove—at least in the sense of mathematical proof—the antecedent of a conditional of either of these types.**
> —Barwise (1989b, p. 847, col. 2, italics in original, my boldface)

This is a point made by Brian Cantwell Smith (1985), which we'll look at in the next chapter. Barwise cites Smith, noting that

> Computer systems are not just physical objects that compute abstract algorithms. They are also embedded in the physical world and they interact with users. ... Thus, ... our mathematical models need to include not just a reliable model of the computer, but also a reliable model of the environment in which it is to be placed, including the user. —Barwise (1989b, p. 850, col. 1)

Barwise noted that Fetzer was only willing to talk about the causal (that is, physical) role of computers, which is not susceptible of mathematical verification, whereas the field of program verification only concerns abstract programs (p. 848, col. 2). So it really seems that both sides are not only talking past each other, but might actually be consistent with each other!

Question for the Reader:
In remarks given at the 40th Anniversary celebration of the founding of the SUNY Buffalo Department of Computer Science & Engineering (April 2007), Bruce Shriver, former president of the IEEE Computer Society, said

> Hardware does not have flaws; only software does.

What do you think he might have meant by this?

Further Reading:
For Fetzer's later writings on the controversy, see Fetzer 1991 (a summary, reply to objections, and further discussion of the relation of software to hardware), Fetzer 1993 (an encyclopedia-style article on program verification, but written from Fetzer's perspective), Fetzer 1996 ("... in the real world, the operation of computer systems is inherently uncertain, which raises profound problems for public policy" (from the Abstract)), and Fetzer 1998 ("The reliability of computer systems ... depends on the ... interaction of hardware and software ... and the accuracy and completeness of the knowledge upon which they are based. ... The reliability of computer-based systems is necessarily uncertain ... [and] must not be taken for granted" (from the Abstract)).

Burkholder 1999 uses Fetzer 1988 and Barwise 1989b to argue that AI is an empirical science.

MacKenzie 2001, Ch. 6, "Social Processes and Category Mistakes", concerns the De Millo et al. 1979-Fetzer 1988 controversy. For a review, see B. Hayes 2002.

Glass 2002 is a historical survey, with some useful references, arguing that the controversies over program verification are "extremely healthy for the field" of computing, roughly for the same reasons that (as we argued in Chapter 2) philosophy is important: the challenging of assumptions.

16.6 Summary

Recall from §9.4.1 that Kleene claimed that Turing machines, unlike physical computers, were "error free" (Kleene, 1995, p. 27). As noted in that section, if the Turing machine were poorly programmed, it *wouldn't* be error free! Indeed, fifty years earlier, von Neumann said:

> The remarks . . . on the desired automatic functioning of the device [that is, von Neumann's definition of a computer, as quoted in Ch. 9, §9.3.2] must, of course, assume that it functions faultlessly. Malfunctioning of any device has, however, always a finite probability—and for a complicated device and a long sequence of operations it may not be possible to keep this probability negligible. Any error may vitiate the entire output of the device. For the recognition and correction of such malfunctions intelligent human intervention will in general be necessary.
>
> However, it may be possible to avoid even these phenomena to some extent. The device may recognize the most frequent malfunctions automatically, indicate their presence and location by externally visible signs, and then stop. Under certain conditions it might even carry out the necessary correction automatically and continue. —von Neumann (1945, §1.4, p. 1, my interpolated brackets).

One way to read this is as a recognition or anticipation of Fetzer's point. Given this inevitability, the focus presumably has to be on the elimination of *logical* errors, so that program verification still has a role to play. The second paragraph suggests that some *machine* "verification" might be automated, but that just leads to an endless regress: Even if the logical structure of that automation is guaranteed, the physical device that carries it out will itself be subject to some residual malfunction possibilities (Bringsjord, 2015).

Of course, another way to read von Neumann's remarks (as well as the entire program verification debate) is to recognize that no one, and no thing, is perfect. There's always the chance of error or malfunction: Complete elimination of error is physically impossible, so the point is, at least, to minimize it.

Thus, the entire issue of program verification might be considered as a subset of the more general engineering issue of reliability. Allen Newell (1980, p. 159), for example, assumes that a symbol system should be "totally reliable, nothing in its organization reflecting that its operators, control or memory could be erroful" [sic!]. He goes on to say that "universality is always relative to physical limits, of which reliability is one" (p. 160), where 'universality' is defined as the ability to "produce an arbitrary input-output function" (p. 147). This suggest that, even if a *program* could be proved mathematically to be correct, the *process* that executes it would still be limited by *physical* correctness, so to speak, and that, presumably, cannot be *mathematically* proved.

> From a methodological point of view, it might be said that programs are conjectures, while executions are attempted—and all too frequently successful—refutations (in the spirit of Popper . . .). —Fetzer (1988, p. 1062, col. 2)

And, in addition to the slippages between the real world, models of it, algorithms that simulate the models, programs that implement the algorithms, and physical processes that implement the programs (the real world again), Carhart (1956, p. 149) calls for "a total systems approach" that would include not only the physical components (hardware) but also the *people* who operate it (whom he called 'software'—recall our discussion in §12.4).

Note how the issue that we discussed in Chapter 12 about the nature of software vs. hardware is relevant to the issue of program verification. Does a formal proof of a

program's "correctness" apply to the program as software or to the program as hardware (perhaps to the process that comes into existence when the program is executed)? This is also relevant to a discussion we will have in the next chapter about the relation between computers and the world.

The bottom line is that programs as hardware need causal rules of inference of the form: input *I causes* output *O*. Perhaps the BEEP command would have to be annotated something like this:

{The bell is in working order.} BEEP {A sound is emitted.}

If such causal rules are part of the definition of an abstract machine, then we *can* have "absolute" verification of the program. But if they are merely empirical claims, then we can only have "relative" verification of the program.

Even so, absolute verification is often thought to be too tedious to perform and can lure us into overconfidence. The problem of tediousness seems to me not to be overly serious: It's tedious to prove theorems in mathematics, too. In any case, techniques are being devised to automate program verification. The problem of overconfidence is more important, for precisely the reasons that Fetzer adduces. Just because you've proved that a program is correct is no reason to expect that the computer executing it will not break down.

But, in addition to the relativity to axioms (logical relativity) and to premises (subject-matter relativity), there is another "level" of relativity:

> Mathematical argument can establish that a program or design is a correct implementation of that specification, but not that implementation of the specification means a computer system that is "safe", "secure", or whatever.
> —MacKenzie (1992, p. 1066, col. 2)

There are two points to notice here. First, a mathematical argument can establish the correctness of a program relative to its specification, that is, whether the program satisfies the specification. But, second, not only does this not necessarily mean that the computer system is safe (or whatever), it also does not mean that the *specification* is itself "correct":

> Human fallibility means some of the more subtle, dangerous bugs turn out to be errors in design; the code faithfully implements the intended design, but the design fails to correctly handle a particular "rare" scenario.
> —Newcombe et al. (2015, p. 67)

Presumably, a specification is a relatively abstract outline of the solution to a problem. Proving that a computer program is correct relative to—that is, satisfies—the specification does not guarantee that the specification actually solves the problem!

This is the case for reasons that Smith (1985) discusses and that we will look at in the next chapter. Once more, it is not unrelated to the Computability Thesis and to Rosser's debate between "Descartes" and the time-traveling mathematician: You can't show that two systems are the same, in some sense, unless you can talk about both systems in the same language. In the case of the Computability Thesis, the problem concerns the informality of the language of algorithms versus the formality of the

language of Turing machines or recursive functions. In the present case, the problem concerns the mathematical language of programs versus the non-linguistic, physical nature of the hardware. Only by *describing* the hardware in a (formal, mathematical) language, can a proof of equivalence be attempted. But then we also need a proof that that formal description is correct; and that can't be had (as Barwise noted):

> ... what can be proven correct is not a physical piece of hardware, or program running on a physical machine, but only a mathematical model of that hardware or program. —MacKenzie (1992, p. 1066, col. 2).

The abstraction that produced the design (the specification) can be in error, and that is precisely the kind of error that Smith warns about. It is time to look into this possibility. How *do* programs relate to the real world?

Further Reading:
Colburn et al. 1993 is an anthology containing many of the papers discussed in this chapter, as well as an extensive bibliography on program verification. And Colburn himself, some of whose views on the philosophy of CS we have examined in earlier chapters, has two papers on program verification that are worth reading (Colburn, 1991, 1993).

Frenkel 1993, which was cited in Chapter 11, is an interview with Turing Award winner Robin Milner that discusses program verification, among other topics.

Long after the Fetzer controversy, articles for and against program verification continue to be published. Neumann 1996 discusses the use of formal methods to reduce risks. Henzinger 1996 and Hinchey et al. 2008 give arguments in favor of program verification. Tedre 2007, p. 108, says that "one cannot formally prove either that an engineered product has the intended qualities or that an engineered product will not fail ...". Dewar and Astrachan 2009 is a debate over the teaching of formal reasoning in computer science, including program-verification techniques. For Hoare's later views, see Hoare 2009 and an interview with him in Shustek 2009. Leroy 2009 offers a "Formal Verification of a Realistic Compiler"; see also the introductory editorial (Morrisett, 2009), containing philosophical remarks on the value of program verification. Lamport (2015) argues that "the main reason for writing a formal spec[ification] is to apply tools to check it", that "the math needed for most specifications is quite simple: predicate logic and elementary set theory", that "a specification can and should be embedded as a comment within the code it is specifying", and—perhaps most importantly—that "thinking does not guarantee that you will not make mistakes. But not thinking guarantees that you will."

Digression: Program Verification and Argument Analysis:
Fetzer's paper makes some comments about the nature of logical reasoning and about knowledge and belief that are relevant to what you've been doing in your argument analyses.

1. On p. 1050, column 1, he says:

   > [W]hat makes … a *proof* a proof is its validity rather than its acceptance (by us) as valid, just as what makes a sentence true is [that] what it asserts to be the case is the case, no[t] merely that it is believed (by us) and therefore referred to as *true*.

   Note that I've been allowing you to evaluate the truth-value of the premises of an argument, *not* by trying to demonstrate whether they *are* true, but by trying to say whether and why *you believe* them. According to Fetzer's quote, it would follow that you are not *really* evaluating the premises.

   He's correct! Whether a statement (or premise) *is* true or not does *not* depend on whether you (or anyone) *believes* it to be true. It is (or isn't) true iff what it states corresponds (or fails to correspond) to reality.

   Nevertheless, that's very hard (if not impossible) to prove. And that's why I'm allowing you to do something that is a bit easier, and a bit more realistic, and—for our purposes—just as useful, namely, to try to explain whether and why you *believe* the statement.

2. In the same column, at the beginning of the next section, Fetzer says:

   > Confidence in the truth of a theorem (or in the validity of an argument) … appears to be a psychological property of a person-at-a-time ….

   It's that "confidence" that I've been asking you to examine, explain, and defend. Because it's a "psychological property of" *you now*, I only grade you on how well you explain and defend your confidence, not on what you are confident about.

3. Finally, in column 2 on the same page, he says:

   > [A]n individual z who is in a state of belief with respect to a certain formula f … cannot be properly qualified as possessing knowledge that f is a theorem unless his belief can be supported by means of reasons, evidence, or warrants ….

   This is a complicated way of making the following important point: If you *believe* a statement f, that belief doesn't count as *knowing* that f is the case *unless* you have a *reason* for your belief. In other words, knowledge is belief plus (at least) a reason. (Actually, most philosophers agree that knowledge also requires a third thing: knowledge is belief, plus a reason, *plus f being true*: You can't "know" something that's false.)

   This need to justify your beliefs is what turns a mere opinion, or an expression of feeling, into a claim that is worthy of holding and of convincing others of. It's why we have arguments to justify conclusions, and why we have to also justify all the premises of the arguments. (And it's why we have to justify, recursively, all the justifications, until we reach some starting point that is a self-justifying belief. But it's not clear that there really are any, which means that our investigations may never end!)

Chapter 17

How Do Programs Relate to the World?

Version of 5 September 2019;[1] DRAFT © 2004–2019 by William J. Rapaport

> Today, computing scientists face their own version of the mind-body problem:
> how can virtual software interact with the real world?
> —Philip Wadler (1997, p. 240)

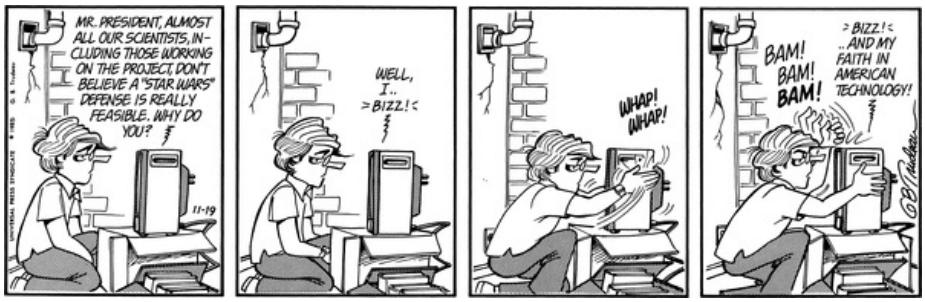


Figure 17.1: https://www.gocomics.com/doonesbury/1985/11/19;
©1985, Universal Press Syndicate

[1] Portions of this chapter are adapted from Rapaport 2017a.

17.1 Readings:

1. Required:

   - Smith, Brian Cantwell (1985), "Limits of Correctness in Computers", *ACM SIGCAS Computers and Society* 14-15 (1-4) (January): 18–26, https://www.student.cs.uwaterloo.ca/~cs492/11public_html/p18-smith.pdf
     - Reprints:
       * *Technical Report CSLI-85-36* (Stanford, CA: Center for the Study of Language and Information)
       * Dunlop, Charles; & Kling, Rob (eds.), *Computerization and Controversy* (San Diego: Academic Press, 1991): 632–646.
       * Colburn, Timothy R.; Fetzer, James H.; & Rankin, Terry L. (eds.) (1993), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers): 275–293.

2. Highly Recommended:

   (a) Piccinini, Gualtiero (2006), "Computation without Representation", *Philosophical Studies* 137(2): 204–241, http://www.umsl.edu/~piccininig/Computation_without_Representation.pdf

   (b) Rescorla, Michael (2007), "Church's Thesis and the Conceptual Analysis of Computability", *Notre Dame Journal of Formal Logic* 48(2): 253–280, http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/church2.pdf

   (c) Egan, Frances (2010), "Computational Models: A Modest Role for Content", *Studies in History and Philosophy of Science* 41(3) (September): 253–259.

   (d) Hill, Robin K. (2016) "What an Algorithm Is", *Philosophy and Technology* 29: 35–59.

3. Recommended:

   (a) Egan, Frances (1995), "Computation and Content", *Philosophical Review* 104(2) (April): 181–203.

   (b) Peacocke, Christopher (1999), "Computation as Involving Content: A Response to Egan", *Mind & Language* 14(2) (June): 195–202.

   (c) Piccinini, Gualtiero (2004), "Functionalism, Computationalism, and Mental Contents", *Canadian Journal of Philosophy* 34(3) (September): 375–410, http://www.umsl.edu/~piccininig/Functionalism_Computationalism_and_Mental_Contents.pdf

   (d) Piccinini, Gualtiero (2008), "Computers", *Pacific Philosophical Quarterly* 89: 32–73, http://www.umsl.edu/~piccininig/Computers.pdf

   (e) Rescorla, Michael (2012), "Are Computational Transitions Sensitive to Semantics?", *Australian Journal of Philosophy* 90(4) (December): 703–721, http://www.philosophy.ucsb.edu/docs/faculty/papers/formal.pdf

   (f) Rescorla, Michael (2013), "Against Structuralist Theories of Computational Implementation", *British Journal for the Philosophy of Science* 64(4) (December): 681–707, http://philosophy.ucsb.edu/docs/faculty/papers/against.pdf

(g) Rescorla, Michael (2014), "The Causal Relevance of Content to Computation", *Philosophy and Phenomenological Research* 88(1) (January): 173–208, http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/causalfinal.pdf

(h) Rescorla, Michael (2015), "The Representational Foundations of Computation", *Philosophia Mathematica* 23(3): 338–366, http://www.philosophy.ucsb.edu/docs/faculty/michael-rescorla/representational-foundations.pdf

(i) Shagrir, Oron; & Bechtel, William (2015), "Marr's Computational-Level Theories and Delineating Phenomena", in D.M. Kaplan (ed.), *Integrating Psychology and Neuroscience: Prospects and Problems* (Oxford: Oxford University Press), http://philsci-archive.pitt.edu/11224/1/shagrir_and_bechtel.Marr's_Computational_Level_and_Delineating_Phenomena.pdf

(j) Shagrir, Oron (2018), "In Defense of the Semantic View of Computation", *Synthese*, https://doi.org/10.1007/s11229-018-01921-z

17.2 Introduction

In the previous chapter, we looked at arguments to the effect that, roughly, a computer program might succeed in theory but fail in practice. In this chapter, we continue our examination of the relationship between a program and the world in which it is executed.

Science, no matter how conceived, is generally agreed to be a way of *understanding* the world (as we saw in Chapter 4). So, CS as a science should be a way of understanding the world *computationally*. And engineering, no matter how conceived, is generally agreed to be a way of *changing* the world, preferably by improving it (as we saw in Chapter 5). So, CS as an engineering discipline should be a way of changing (improving?) the world via computer programs that have physical effects. One of CS's central questions is "What can be computed physically?", which merges the (theoretical) scientific and (real-world practical) engineering aspects of CS (§3.15.2). Thus, CS deals with the real world by trying to understand the world computationally, and to change the world by building computational artifacts.

In this chapter, we will focus on two questions:

1. **Is computing directly concerned with the *world*?**

2. **Or is computing only indirectly concerned with the world by directly dealing only with *descriptions* or *models* of the world?**

In other words, is computation primarily concerned with the internal workings of a computer, both abstractly in terms of the theory of computation—for example, the way in which a Turing machine works—as well as more concretely in terms of the internal physical workings of a physical computer? This is an aspect of question 2.

Or is computation primarily concerned with how those internal workings can reach out to the world in which they are embedded? As Philip Wadler noted (see the epigraph for this chapter), this is related to the question of how the mind (or, more materialistically, the brain) reaches out to "harpoon" the world (Castañeda, 1989, p. 114). This is an aspect of question 1.

Those who say that computing is directly concerned with the *world* sometimes describe computing as being "external", "global", "wide", or "semantic". And those who say that computing is only directly concerned with *descriptions* or *models* of the world sometimes describe computing as being "internal", "local", "narrow", or "syntactic". As you should expect by now, of course, it might be both! After all, even if computing is "narrow", it is embedded in—and interacts with—the "wider" world. In that case, the question is how these two positions are related.

We have been considering these issues throughout the book. So, we'll continue where we left off in the last chapter, with program verification. We'll then revisit other topics, and suggest some conclusions that might be drawn from our survey so far.

17.3 Program Verification, Models, and the World

> The goal of software development is to model a portion of the real world on the computer. ... That involves an understanding not of computers but of the real-world situation in question. ...That is not what one learns in studying computer science; that is not what computer science is about.
> —Michael Mahoney (2011, p. 117)

17.3.1 "Being Correct" vs. "Doing What's Intended"

Recall that one objection to program verification is that a program can be "proven correct" yet not "do what you intend". One reason, as we saw in the previous chapter, might be that the *computer* on which the program is run might fail physically. That is, the computer system might fail at the hardware level (for a humorous take, see Figure 17.1).

A second reason might be that the *world* is inhospitable. There are two ways in which this latter problem might arise. There might be a physical problem with the connection between the computer and the environment: At a simple level, the cables connecting the computer to the world (say, to a printer) might be flawed. Or the world itself—the environment—might not provide the correct conditions for the intended outcome, as in Cleland's hollandaise-sauce case (§10.4.1).

A third reason is related to the possible "hyper"-computability of interactive programs, which might depend on the unpredictable and non-verifiable behavior of an "oracle" or human user (§§11.4.3–11.4.4).

What does 'correct' mean in this context? Does it mean that the program has been logically verified? Does it mean that it "does what was intended"? Perhaps a better way of looking at things is to say that there are two different notions of "verification": an internal one (logical verification) and an external one (doing what was intended) (Tedre and Sutinen, 2008, pp. 163–164).

But, to the extent that doing what was intended is important, then we need to ask *whose* intent counts? Here is computer scientist and philosopher Brian Cantwell Smith on this question:

> What does *correct* mean, anyway? Suppose the people want peace, and the President thinks that means having a strong defense, and the Defense department thinks that means having nuclear weapons systems, and the weapons designers request control systems to monitor radar signals, and the computer companies are asked to respond to six particular kinds of radar pattern, and the engineers are told to build signal amplifiers with certain circuit characteristics, and the technician is told to write a program to respond to the difference between a two-volt and a four-volt signal on a particular incoming wire. If being correct means *doing what was intended*, whose intent matters? The technician's? Or what, with twenty years of historical detachment, we would say *should have been intended*?
> (B.C. Smith 1985, §2, p. 20, col. 1)

According to Smith, the cause of these problems lies not in the relation of *programs* to the world, but in the relation of *models* to the world. Let's see what he means.

17.3.2 Models: Putting the World into Computers

> What the conference [on the history of software] missed was software as model, ... software as medium of thought and action, software as environment within which people work and live. It did not consider the question of *how we have put the world into computers*.
> —Michael Mahoney (2011, pp. 65–66; my interpolated brackets and italics)

According to Smith (1985, §3, p. 20, col. 1), to design a computer system to solve a real-world problem, we *don't* directly "put the world into computers". Rather, we must do two things:

1. Create a *model* of the real-world problem.
2. Create a *representation of the* ***model*** in the computer.

17.3.2.1 Creating a Model of the World

Moreover, "to build a model is to conceive of the world in a certain delimited way" (Smith, 1985, §3, p. 20, col. 1). The model that we create has no choice but to be "delimited", that is, it must be abstract—it must omit some details of the real-world situation. Abstraction, as we saw in §14.2.1, is the opposite of implementation: It is the removal of "irrelevant" implementation details.

Why must any real-world information be removed? Why are models necessarily partial? One reason is that it is methodologically easier to study a phenomenon by simplifying it, coming to understand the simplified version, and then adding some complexities back in, little by little. If models weren't partial, there would be too much complexity, and we would be unable to use them as a basis for action. You can't use, much less have, a map of Florida that is the size of Florida and that therefore can show everything in Florida. Such a map might be thought to be more useful than a smaller, more manageable one, in that it would be able to show all the detail of Florida itself. But the life-sized version's lack of manageability is precisely its problem.

Further Reading:
The earliest discussion of a map that is the same size as what it maps is due to Lewis Carroll (of *Alice in Wonderland* fame); see Carroll 1893, Ch. 11, http://etc.usf.edu/lit2go/211/sylvie-and-bruno-concluded/4652/chapter-11-the-man-in-the-moon/. Other discussions of the idea can be found in Royce 1900, pp. 502–507ff; Rosenblueth and Wiener 1945, p. 320; Rapaport 1978, §5; Borges 1981, p. 234; and Eco 1982. All of these concern space, but there is an analogous problem for time: Weather prediction is based on models of the world that have to be "capable of scrolling ahead to the future faster than time can progress" (Fry, 2019).

Can we eat our cake but keep it, too? Perhaps we *can* use the real world as a representation of itself. The computer scientist Rodney Brooks (1991, §1) suggested that we should "use the world as its own model": A Roomba robotic vacuum cleaner doesn't *need* a map showing where there is a wall; if it bumps into one, it will know that it's there. But even this is only a *part* of the real world.

In any case, the usual first step in solving a problem is to create a "delimited" (abstract, simplified) model of it. For example, Figure 17.2 is a picture showing (in a 2-dimensional way, of course!) a 3-dimensional, real-world house and a 2-dimensional model of it.

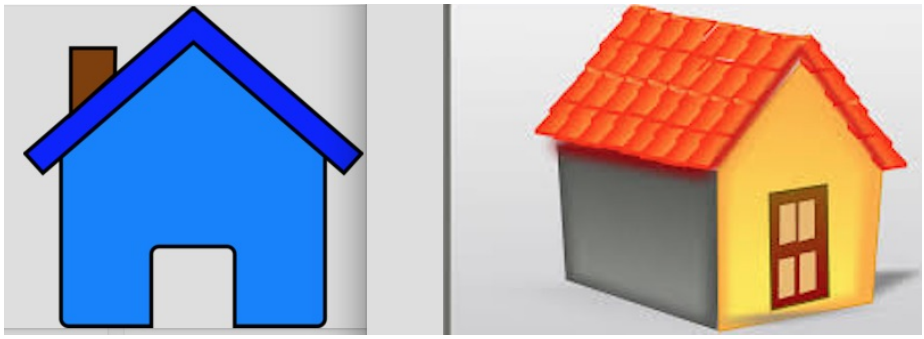


Figure 17.2: Right: 3D, real-world house; Left: 2D model of it

17.3.2.2 Creating a Computer Representation of the Model

A second step is to use logical propositions or programming-language data structures to represent, *not* the real-world situation, but the *model*. Figure 17.3 adds to the house and the house's model a computer representation *of the model*.[2]

Figure 17.3: Right: 3D, real-world house;
Middle: 2D model of 3D real-world house;
Left: computer representation of 2D model

Smith's point is that computers only deal with *their representations of* these *abstract models of* the real world. As Paul Thagard (1984, p. 82, citing Zeigler 1976) notes, computers are twice removed from reality, because "a computer simulates a model which models a real system".

Is that necessarily the case? Can't we skip the intermediate, abstract model, and directly represent the real-world situation in the computer? Perhaps, but this won't help

[2]Note to readers of the draft of this chapter: I am not an expert on illustrations. The intention of these figures is to show three things: a 3D picture of a house; a 2D drawing of it in the same colors (please imagine the 2D drawing as being yellow and red, *not* blue!); and a computer representation of the 2D drawing. Please use your imagination to create a better set of figures. Smith's original is in Figure 17.4.

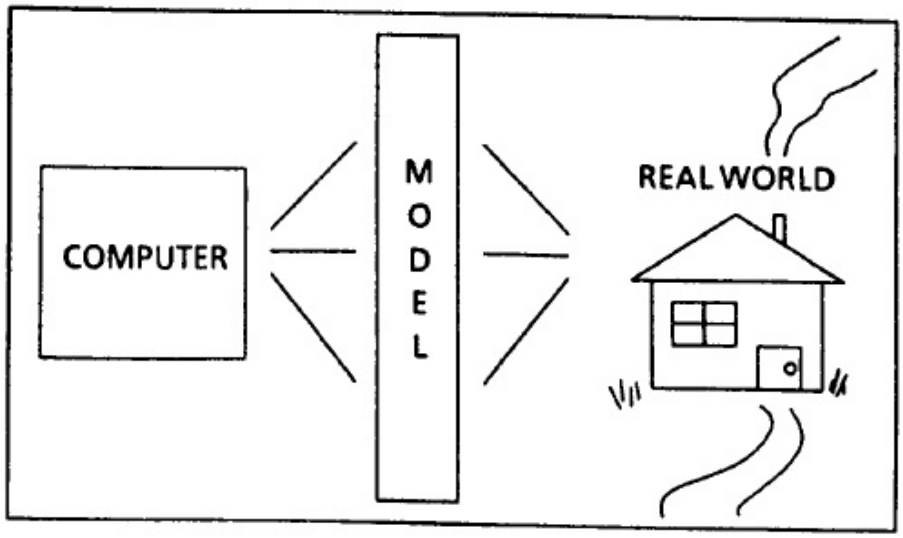


Fig. 1. Computers, models, and the embedding world.

Figure 17.4: From Colburn et al. 1993, p. 283

us avoid the problem of *partiality* (or abstraction, or idealization, or simplification). The only rational way to deal with (real-world) complexity is to analyze it, that is, to simplify it, that is, to deal with a partial (abstract) representation or model of it.

We are condemned to do this whenever we act or make decisions: If we were to hold off on acting or making a decision until we had complete and fully accurate information about whatever situation we were in, we would either be paralyzed into inaction or else the real world might change before we had a chance to complete our reasoning. (As Alan Saunders—and, later, John Lennon—said, "Life is what happens to us while we are making other plans.")[3] This is the problem that Herbert Simon recognized when he said that we must always reason with uncertain and incomplete (even noisy) information: Our rationality is "bounded", and we must "satisfice" (Simon, 1996b, p. 27). And this holds for computation as well as thinking.

Further Reading:
We discussed the role of CS in managing complexity in §§3.14.3, 4.5.2, and 15.4.2. See also Simon 1962, 1996b; Dijkstra 1972.

But action is *not* abstract: You *and* the computer must act *in* the complex, real world, even though such real-world action must be based on *partial models* of the real world, that is, on incomplete and noisy information. Moreover, there is no guarantee that the *models* are correct.

[3] http://quoteinvestigator.com/2012/05/06/other-plans/

Action can help: It can provide feedback to the computer system, so that the system won't be isolated from the real world. Recall the blocks-world program that didn't "know" that it had dropped a block, but "blindly" continued to faithfully execute its program to put the block on another (§10.4.1). If it had had some sensory device that would have let it know that it no longer was holding the block that it was supposed to move, and if the program had had some kind of error-handling procedure in it, then it might have worked much better (it might have worked "as intended"). *Did* the blocks-world program behave as intended?

17.3.2.3 Model vs. World

The problem, as Smith sees it, is that mathematical model theory only discusses the relation between the model and a *description* of the model. It does not discuss the relation between the model and the *world*. A model is like eyeglasses for the computer, through which it sees the world. The model *is* the world as the computer sees it. The problem is that computers have to act in the *real* world on the basis of a *model* of it.

Philosophical Digression:
Immanuel Kant said that the same thing is true about us: Our concepts are like eyeglasses that distort reality; our only knowledge of reality is filtered through our concepts, and we have no way of knowing how things "really" are "in themselves", unfiltered by our concepts (as illustrated in Figure 17.5). (Recall our earlier discussions of Kant in §§3.12 and 4.5.1.)

Similarly, to prove a program correct, we need both (a) a *specification* (a model of the real-world problem) that says (declaratively) *what* the computer systems should do and (b) a *program* (a computer model of the specification model) that says (usually procedurally) *how* to accomplish this. A correctness proof, then, is a proof that any system that obeys the program will satisfy the specification. But this is a proof that two *descriptions* are compatible. The program is proved correct *relative to* the specification:

> ... what can be proven correct is not a physical piece of hardware, or program running on a physical machine, but only a mathematical model of that hardware or program. (MacKenzie 1992, p. 1066; see also Turner 2018, §4.4)

Suppose the proof fails to show "correctness"; what does this mean? It means *either* that the program is wrong, *or* that the specification is wrong (or both). And, indeed, often we need to adjust both specification and program.

The real problems lie in the model-world relation, which correctness does not address. This is one of the morals of Cleland's and Fetzer's claims. That is, programs can fail because the models can fail to correspond to the real world *in "appropriate" ways*. But that italicized clause is crucial, because all models abstract from the real world, but each of them does so in different ways.

This is the case for reasons that are related to the Church-Turing Computability Thesis: You can't show that two systems are the same, in some sense, unless you can talk about both systems in the same language. (Recall Rosser's time-traveling mathematician, from §16.2.) In the case of the Computability Thesis, the problem concerns the *informality* of the language of algorithms versus the *formality* of the language of

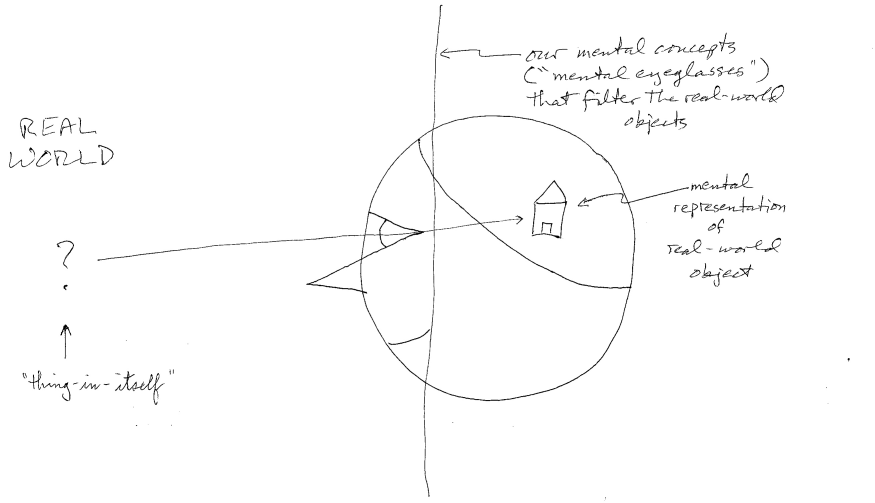


Figure 17.5: A cognitive agent looking at a real-world object that the agent categorizes as a house. Light reflecting off the house (the "thing-in-itself") enters the agent's eyes and the resulting neural signals are "filtered" through the agent's mental concepts, producing a mental image of the house. The mental image may, or may not, be a "perfect" representation of the house, but the agent has no way to directly compare the mental image with the real house, independent of the agent's concepts. Figure 17.4 is Smith's version of both Figure 17.3 and Figure 17.5.

Turing machines. In the present case, the problem concerns the mathematical *language* of programs versus the non-linguistic, *physical* nature of the hardware. Only by *describing* the hardware in a (formal, mathematical) *language*, can a proof of equivalence be attempted. But then we also need a proof that that formal description of the hardware is correct; and that can't be had. It can't be had, because, to have it, we would need *another* formal description of the hardware to compare with the formal description that we were trying to verify. And that leads to a Zeno-like infinite regress. (We can come "close, but no cigar".)[4]

Both Smith and Fetzer agree that the program-verification project fails, but for slightly different reasons: For Fetzer (and Cleland), computing is about the *world*; it is external and wide. Thus, computer programs can't be ("absolutely" or "externally") verified, because the world may not be conducive to "correct" behavior: A physical part

[4]Note to philosophers: It's actually closer to a Bradley-like regress (Perovic, 2017), which was one of Royce's points in his discussion of maps; see also Rapaport 1995, §2.5.2, p. 64.

might break; the environment might prevent an otherwise-perfectly-running, "correct" program from accomplishing its task (such as making hollandaise sauce on the Moon using an Earth recipe); and so on.

For Smith, computing is done on a *model* of the world; it is internal and narrow. Thus, computer programs can't be verified, because the model might not match the world. Smith also notes that computers must act *in* the real world. But their abstract narrowness *isolates* them from the concrete, real world *at the same time* that they must act *in* it. Smith's gap between model and world is due, in part, to the fact that specifications are abstract:

> A specification is an abstraction. It should describe the important aspects and omit the unimportant ones. *Abstraction is an art* that is learned only through practice. ... [A] specification of what a piece of code does should describe everything one needs to know to use the code. It should never be necessary to read the code to find out what it does. (Lamport, 2015, p. 39, my italics)

How does one know if s omething that has been omitted from the specification is important or not? This is why "abstraction is an art" and why there's no guarantee that the model is correct (in the sense that it matches reality).

Further Reading:
On what I am calling "Smith's gap" between the model and the world, see:

1. M. Jackson 2003, which discusses "the interplay between the formal world of the computer and its programming language with the informal world where the problem to be solved is located" (from the abstract, p. 13).
2. van Fraassen 2006, which discusses the relation between formal, mathematical models of reality and the reality of which they are models, arguing that what I call Smith's gap presents difficulties for scientific realism (which we discussed in §4.6).
3. Rescorla 2015b, which identifies "a gap between the domain of items manipulated by the Turing machine and our desired domain of computation" (§1). From the existence of this gap—which seems clearly akin to Smith's—Rescorla argues "that computability theory is an *intensional* enterprise ... : it studies entities as represented in certain ways, rather than entities detached from any means of representing them" (§1). (On "intenSionality" with an 's', see our §3.4, above, and Rapaport 2012a.) We'll come back to this in §17.5.

Smith's gap is also related to the issues surrounding attempts to prove the Computability Thesis from axioms that are intended to capture the informal notion of computability (such as Dershowitz and Gurevich 2008; Sieg 2008):

> ... how [can] we tell whether a given piece of live mathematical reasoning corresponds to a given actual or envisioned formal proof ... How does one guarantee that the stated axioms or premises of the formal proof are in fact necessary for the intuitive, pre-theoretic notions invoked in the informal text? That is, how do we assure ourselves that the formalization is faithful? This question cannot be settled by a *formal* derivation. That would start a regress, at least potentially. We would push our problem to the axioms or premises of *that* derivation. Moreover, any formalization of a real-life argument reveals missing steps, or gaps, and plugging those would sometimes require theses much like Church's thesis. (Stewart Shapiro 2013, pp. 158–159.)

Fetzer 1999 is a clearly written summary and critique of Smith's essay, arguing that there are *more* than merely the two models that Smith considers (a specification as a model of the world and a program as a model of the specification).

Smith's essay begins with a discussion of a real event, in which the Moon was mistaken for a Soviet missile attack. The Associated Press report on the moon-missile mistake is "Canadian Is Praised over Missile Scare", *New York Times* (23 December 1960), p. 6, https://timesmachine.nytimes.com/timesmachine/1960/12/23/99831694.pdf. For more examples like this, only with more dire consequences, see Neumann 1993. B. Hayes 2007a is another interesting article on computational modeling (and what can go wrong). And Cerf 2013 suggests "that we should treat as robots any programs that can have real-world ... effect. ... [T]hose of us who live in and participate in the creating of software-based 'universes' might wisely give thought to the potential impact that our software might have on the real world", a thought that echoes Smith's essay.

17.4 Internal vs. External Behavior: Some Examples

So, internal models diverge from the external world. In earlier chapters, we saw several examples of programs whose internal (local, narrow, syntactic) behavior differed from their external (global, wide, semantic) behavior. Let's briefly review these, plus a few new ones.

17.4.1 Successful Internal Behavior but Unsuccesful External Behavior

The first two examples concern the kinds of situations that Fetzer and Smith were concerned with, in which the computer program behaves exactly as was expected—there are no logical program bugs, and the program does not crash—yet it fails to accomplish its stated task. That is, it exhibits "successful" *internal* behavior but "unsuccessful" *external* behavior.

17.4.1.1 The Blocks-World Robot

The blocks-world program of §10.4.1 worked "correctly", in the sense that it performed each step without crashing. Yet it did not do what was intended, because it accidentally dropped a block, and was therefore unable to put it where it was supposed to go. "Narrowly", perhaps, it did what was intended; "widely", however, it didn't: After all, it didn't actually manipulate the blocks.

17.4.1.2 Cleland's Recipe for Hollandaise Sauce

Recall Cleland's recipe for hollandaise sauce, which we also discussed in §10.4.1. Suppose that we have an algorithm (a recipe) that tells us to mix eggs, butter, and lemon juice, and that is supposed to output hollandaise sauce. On Earth, the recipe results in an emulsion that is, in fact, hollandaise sauce. But, on the Moon, it does not result in an emulsion, so that no hollandaise sauce is output; instead, the output is a messy mixture of eggs, butter, and lemon juice. (On whether recipes really are algorithms, recall §§3.9.3 and 10.4.2.) In Cleland's case, is making hollandaise sauce computable (on the "narrow" view) or not (on the "wide" view)? *Can* a Turing machine or a physical computer make hollandaise sauce?

17.4.2 Same Internal Behavior but Different External Behavior

The second set of examples concern situations in which a single internal behavior generates different external behaviors, depending on context.

17.4.2.1 Fodor's Chess and War Programs

Recall also from §10.4.1 that the philosopher Jerry Fodor (1978, p. 232) asked us to consider two computer programs: one that simulated the Six Day War and another that simulated (or actually plays?) a game of chess, but which were such that "the internal

career of a machine running one program would be identical, step by step, to that of a machine running the other". In programs like this war-chess case, do we have one algorithm (the "narrow" view), or two (the "wide" view)?

Question for the Reader:
Is there a difference between *simulating* playing a chess game and *really* playing one? (Recall our discussion of simulation vs. "the real thing" in §15.3.1.2. We'll return to this in Chapter 19.)

A real example along the same lines is "a method for analyzing x-ray diffraction data that, with a few modifications, also solves Sudoku puzzles" (Elser, 2012). Or consider a computer version of the murder-mystery game Clue that exclusively uses the Resolution rule of inference, and so could be a general-purpose, propositional theorem prover instead.[5] A more recent version is the program AlphaZero, "a single algorithm [that] can learn to play three hard board games" (Campbell, 2018): When supplied with the rules of chess, it becomes the world's best chess player; when supplied with the rules of shogi (Japanese chess), it becomes the world's best shogi player; and when supplied with the rules of Go, it becomes the world's best Go player (Silver et al., 2018; Kasparov, 2018).

Similar examples abound, notably in applications of mathematics to science, and these can be suitably "computationalized" by imagining computer programs for each. For example,

> Nicolaas de Bruijn once told me roughly the following anecdote: Some chemists were talking about a certain molecular structure, expressing difficulty in understanding it. De Bruijn, overhearing them, thought they were talking about mathematical lattice theory, since everything they said could be—and was—interpreted by him as being about the mathematical, rather than the chemical, domain. He told them the solution of their problem in terms of lattice theory. They, of course, understood it in terms of chemistry. Were de Bruijn and the chemists talking about the same thing? (Rapaport, 1995, §2.5.1, p. 63)

A related issue is that a single action in the real world can be described in different ways:

> Recovering motives and intentions is a principal job of the historian. For without some attribution of mental attitudes, actions cannot be characterized and decisions assessed. *The same overt behavior, after all, might be described as "mailing a letter" or "fomenting a revolution"*. (Richards, 2009, p. 415)

And Figure 17.6 presents a humorous one.

[5] Robin Hill, personal communication.

Figure 17.6: https://www.gocomics.com/luann/2015/04/08; ©2015, GEC Inc.

Further Reading:
Other examples, concerning, for example, the applicability of group theory to physics are discussed in Frenkel 2013. And the mathematician Paul Halmos (1973, p. 384) points out that, once aspects of Hilbert spaces are seen to be structurally identical to aspects of quantum-mechanical systems, "the difference between a quantum physicist and a mathematical operator-theorist becomes one of language and emphasis only". Eugene Wigner's classic essay on "The Unreasonable Effectiveness of Mathematics in the Natural Sciences" (1960) is also relevant; computer scientists should be sure to read R.W. Hamming's response (1980a).

17.4.2.2 Rescorla's GCD Computers

Rescorla's GCD computers (see §14.4.3, above) fall under this category but also offer an example reminiscent of Cleland's hollandaise sauce, but less "physical". A Scheme program for computing GCDs of two numbers is implemented on two computers, one (M_{10}) using base-10 notation and one (M_{13}) using base-13 notation. Rescorla argued that only M_{10} executes the Scheme program *for computing GCDs*, even though, in a "narrow" sense, both computers are executing the "same" program. When the *numerals* '115' and '20' are input to M_{10}, it outputs the numeral '5'; "it thereby calculates the GCD of the corresponding *numbers*" (Rescorla, 2013, p. 688). But the *numbers* expressed in base-13 by '115' and '20' are 187_{10} and 26_{10}, respectively, and their GCD is 1_{10}, not 5_{10}. So, in a "wide" sense, the two machines are doing "different things", in one case behaving "correctly", in the other, behaving "incorrectly". *Are* the two GCD computers doing different things?

17.4.2.3 AND-Gates or OR-Gates?

In the Digression on Conjunction and Disjunction in §10.4.1, we saw that the truth table for *conjunction* could also be used as the truth table for *disjunction* by reinterpreting '0's and '1's. Another version uses a single truth table:

A	A	A
A	B	A
B	A	A
B	B	B

This could be interpreted as the truth table for conjunction if A is interpreted as 'false', and B as 'true'. And it could be interpreted as the truth table for disjunction if A is interpreted as 'true', and B as 'false'.

Oron Shagrir (2018b, §3.1) offers a third version. Consider the following function:

Input 1	**Input 2**	**Output**
H	H	H
H	M	M
H	L	M
M	H	M
M	M	M
M	L	M
L	H	M
L	M	M
L	L	L

(Actually, he uses physical voltages, labeled H(igh), M(edium), and L(ow). I am using the more abstract version.) He then gives two different (external semantic) interpretations of these symbols. On the first interpretation, H is mapped to 1 (or "true"), and both L and M are mapped to 0 (or "false"), thus implementing conjunction. (In Shagrir's original version, the physical device that inputs and outputs certain voltages becomes an AND-gate). On the second interpretation, both H and M are mapped to 1, and L is mapped to 0, thus implementing disjunction (or an OR-gate).

Question for the Reader:
In Shagrir's example, we have two different computers (and AND-gate and an OR-gate) that are implemented in the same way. Would two AND-gate computers be the same even if they were implemented differently?

Thus, there are at least three possible answers to the question of what this device computes: conjunction, disjunction, and the function given by the table above. One way of phrasing the puzzle here is this: What is the basic computational structure of this system (or these systems)? Is it conjunction? Disjunction? The H-M-L function? (Or something else? After all, one of the points made by Buechner (2011, 2018, see §16.2, above) is that there might not be any fact of the matter!)

Alternatively, one could say that, in order to compute conjunction, execute an algorithm for the table above, but use the first intepretation, and, in order to computer disjunction, execute that very same algorithm, but use the second interpretation. How crucial is that external interpretation to the computation? Arguably, the difference concerns, not the computation, but how that computation is "plugged in" to the external environment. But, arguably, such external relations don't change the computation any more than the external fact that a person's sibling has had a child (thus making the person an aunt or uncle) changes that person. As in Figure 17.6, the girl held up 10 fingers, irrespective of whether she intended (or the boy understood) "ten yeses or five noes".

Further Reading:
Dennett (2013a, pp. 159–164) discusses a similar example with a vending machine that, when used in the US, works on US quarters, but when used in Panama, works equally well with Panamanian quarter-balboas.

17.5 Two Views of Computation

> A computer program is a message from a man [sic] to a machine. The rigidly marshaled syntax and the scrupulous definitions all exist to make intention clear to the dumb engine. —Frederick P. Brooks (1975, p. 164)

If computation is "narrow", "local", "internal", or "syntactic", then it is concerned only (or at least primarily) with such things as the operations of a Turing machine (print, move) or the basic recursive functions (successor, predecessor, projection). On the other hand, if computation is "wide", "global", "external", or "semantic", then it must involve things like chess pieces and a chess board (for a chess program), or soldiers and a battlefield (for a war simulator). Is computation narrow and independent of the world, or is it wide and world-involving?

A related question asks whether programs are purely logical, or whether they are "intentional" (Hill, 2016b) or "teleological" (Anderson, 2015). Recall from §7.4.3.2 that something can be said to be "intentional" if it is related to goals or purposes. 'Teleological' is another adjective with roughly the same meaning. So we can ask whether programs are goal-oriented.

Philosophical Digression:
Rescorla's argument that computation is "intensional" (with an 's', not a 't')—recall item 3 of the Further Reading at the end of §17.3.2—is related, but slightly different. In earlier chapters, we distinguished being "extensional" from being "intensional" (with an 's'). There is also a difference between being "intensional" (with an 's') and being "intentional" (with a 't'). Very briefly, in a technical philosophical sense, a phenomenon is said to be "intentional" (with a 't') if it is directed to a goal or an object. The 19th-century philosopher Franz Brentano claimed that all mental phenomena were "intentional" in the sense that, when you think, there is always an object that you are thinking *about*. In another, but non-technical, sense, an act is performed "intentionally" if the actor *meant* to do it—that is, it wasn't an accident; it was done on *purpose*. Finally, there is also Dennett's technical notion of the "intentional stance", which we introduced in §12.4.4.1.1. See also Jacob 2019.

Let's distinguish between an *algorithm A* and a *goal* (or *purpose*) *G*. Let *A* be either a primitive computation (such as "print" and "move", or such as the "successor", "predecessor", and "projection" functions), or else a set of computations recursively structured by sequence, selection, and repetition (as in Chapter 7). And let *G* be a goal (or "purpose", or "intended use") of *A*. Then our question can be formulated more precisely as follows: Which of the following two forms does a computer program take?:

Do *A*

or

In order to accomplish goal *G*, do *A*

If the former, then computation is "narrow"; if the latter, then it is "wide".

This enables us to reformulate some earlier issues: For example, does the "correctness" of a computer program refer to algorithm *A* or to goal *G*? The goal of a program can be expressed in its specification. This is why you wouldn't *have* to read the code to find out what it does. Of course, if the specification has been internalized into the code, you might be able to (see §17.8.2.4, below). But it's also why you can have a chess program that is also a war simulator: They might have different specifications but the same code. So, perhaps a correctness proof is a proof that algorithm *A* satisfies specification *G*.

As another example, Cleland argued that "Make hollandaise sauce" was not a Turing-computable function. Can we view "Make hollandaise sauce" as a high-level procedure call that can be substituted for *A*? Or should it be viewed as a goal that can be substituted for *G*? In the latter case, only its expansion into a set of (more basic) computations structured by sequence, selection, and repetition would be a suitable substitute for *A*. And, in that latter case, *A* might be computable on the Moon even if *G* fails there.

We'll refer to these two formulations throughout the rest of the chapter.

Further Reading:
The two formulations "Do *A*" and "In order to accomplish goal *G*, do *A*" echo something that Herbert Simon said three quarters of a century ago. According to Reva Brown (2004, p. 1247, my italics),

> In his exposition of science, Simon ... divides it into two kinds: practical and theoretical. Scientific propositions are *practical* if they are stated in some such form as *"In order to produce such and such a state of affairs, such and such must be done"* (Simon, 1947, p. 248). The equivalent *theoretical* proposition with the same conditions of verification can be stated in a purely descriptive form: *"Such and such a state of affairs is invariably accompanied by such and such conditions"* (Simon, 1947, p. 248).

17.6 Inputs, Turing Machines, and Outputs

> Any machine is a prisoner of its input and output domains.
> —Allen Newell (1980, p. 148)

17.6.1 Introduction

Aaron Sloman notes that, for almost any machine,

> we can, to a first approximation, divide the processes produced by the machine into two main categories: *internal* and *external*. Internal physical processes include manipulation of cogs, levers, pulleys, strings, etc. The external processes include movements or rearrangements of various kinds of physical objects, e.g. strands of wool or cotton used in weaving, ... (Sloman, 2002, §3.2, p. 9)

As we noted in §3.9.3, both Shapiro (2001) and Sloman consider links to sensors and effectors as central to what a computer is. A computer without one or the other of these would be solipsistic.[6] Can computation be understood separately from interaction with the world? It is not that the latter is unimportant or secondary, but that it is a separate thing.

One obvious place where a computer program seems to necessarily interact with the real world is its inputs and output. In §§7.4.3.3 and 11.4.3.1, we considered whether programs *needed* inputs and outputs. Let's review some of this.

17.6.2 The Turing-Machine Tape as Input-Output Device

The tape of a Turing machine records symbols (usually '0' or '1') in its squares. Is the tape the input-output device of the Turing machine? Or is it (merely?) the machine's internal memory device?

Given a Turing machine for computing a certain mathematical function, it is certainly true that the function's inputs will be inscribed on the tape at the beginning of the computation, and the function's results—its outputs—will be inscribed on the tape by the time that the computation halts: So, it certainly looks like the tape is an external input-output device.

However,

> A terminating computation is one in which all the processes terminate; *its output is **the values left in the shared memory***.
> (Denning and Martell, 2015, p. 155, my emphases)

Note that this output need not be *reported* to the external world (such as a user); it's just left there on the tape. Moreover, the inscriptions on the tape will be used and modified by the machine during the computation, in the same way that a physical computer uses its internal memory for storing intermediate results of a computation. So, it looks like the tape is merely an internal memory device. In other words, it also looks like the answer to our questions is: both.

[6] On "solipsism", see §7.4.3.3.

Although Turing's *a*-machines were designed to simulate human computers—that is, humans who compute, thus constituting the first AI program!—Turing didn't talk about the humans who would *use* them. A Turing machine *doesn't* accept user-supplied input from the external world! (Recall our discussion of interactive computing in §11.4.3.) It begins with all data *pre-stored* on its tape and then simply does its own thing, computing the output of a function and leaving the result on the tape. Turing machines don't "tell" anyone in the external world what the answers are, though the answers are there for anyone to read, because the "internal memory" of the machine is visible to the external world. Of course, a user has to be able to *interpret* the symbols on the tape; we'll return to this point in §17.6.6.

Perhaps it would be better to refer to the initial symbols on the tape as "set-up conditions" and the final symbols as "terminal conditions", rather than as "inputs" and "outputs" (as suggested by Machamer et al. 2000, p. 11). So, are the symbols on the tape really inputs and outputs in the sense of coming from, and being reported to, the external world? Are such inputs and outputs an essential part of an algorithm? It may seem outrageous to deny that they are essential, but (as we saw in Chapter 7) it's been done! After all, the input-output interface "merely" *connects* the algorithm with the world. Let's consider whether inputs and outputs are needed.

Further Reading:
For more on whether a Turing-machine tape is an external input-output device or an internal memory, see Dresner 2003, 2012.

17.6.3 Are Inputs Needed?

One reason that it's outrageous that inputs or outputs might *not* be needed is that algorithms are supposed to be ways of computing mathematical functions, and mathematical functions, by definition, have both inputs and outputs—members of their domain and range. Functions are, after all, certain sets of ordered *pairs* (of inputs and outputs), and you can't very well have an ordered *pair* that is missing one or both of those.

We looked at this issue in §§7.4.3.1 and 7.4.3.3. There, we saw that:

- Markov's informal characterization of algorithm had an "applicability" condition stating that algorithms must have "The possibility of starting from original given objects which can vary within known limits" (Markov, 1954, p. 1). Those "original given objects" are, presumably, the input.
- But Hartmanis and Stearns's classic paper on computational complexity (1965, p. 288) allowed their multi-tape Turing machines to have at most one tape—an output-only tape—with no input tapes.
- And we also saw that Knuth's informal characterization of the notion of algorithm had an "input" condition stating that "An algorithm has *zero or more* inputs" (Knuth, 1973, p. 5; my italics). He not only didn't explain this, but he went on to characterize outputs as "quantities which have a specified relation to the inputs" (Knuth, 1973, p. 5). But what kind of relation would an output have to a non-existent input?

Digression and Further Reading:
Besides the basic philosophical "theorem" that, for any X, there is a philosophy of X (§2.8), there is another that says, "For any possibility you can name, there exists a philosopher who turned it into a theory" (Casati, 2000, p. 65). On some philosophical theories about such relations to non-existent entities, see Grossmann 1974, p. 109, and Rapaport 1986b, §4.5.

One way to understand having outputs without inputs is that some programs, such as prime-number generators, merely output information. In cases such as this, although there may not be any *explicit* input, there is an *implicit* input (roughly, ordinals: the algorithm outputs the *n*th prime, without explicitly requesting an *n* to be input). Another kind of function that might seem not to have any explicit inputs is a constant function, but, again, its implicit input could be anything (or anything of a certain type—"varying within known limits", as Markov might have said).

So, what constitutes input? Is it simply the initial data for a computation—that is, is it internal and syntactic? Or is it information supplied to the computer from the external world (and interpreted or translated into a representation of that information that the computer can "understand" and manipulate)—that is, is it external and semantic?

17.6.4 Are Outputs Needed?

Markov, Knuth, and Hartmanis and Stearns all require at least one output. Markov, for example, has an "effectiveness" condition stating that an algorithm must "obtain a certain result".

But Copeland and Shagrir (2011, pp. 230–231) suggest that a Turing machine's output might be unreadable. Imagine, not a Turing machine with a tape, but a physical computer that literally prints out its results. Suppose that the printer is broken or that it has run out of ink. Or suppose that the programmer failed to include a 'print' command in the program. The computer's program would compute a result but not be able to tell the user what it is, as we saw in this algorithm from §7.3.1.4 (Chater and Oaksford, 2013, p. 1172, citing an example from Pearl 2000):

1. input P
2. multiply P by 2; store in Y
3. add 1 to Y; store in Z

This algorithm has an explicit input, but does not appear to have an explicit output. The computer has computed $2X + 1$ and stored it away in Z for safekeeping, but doesn't tell you its answer. There *is* an answer, but it isn't output. ("I know something that you don't!"?)

So, what constitutes "output"? Is it simply the final result of a computation—that is, is it internal and syntactic? Or is it some kind of translation or interpretation of the final result that is physically output and implemented in the real world—that is, is it external and semantic? In the former case, wouldn't both of Rescorla's base-10 and

base-13 GCD computers be doing the same thing? A problem would arise only if they told us what results they got, and we—reading those results—would interpret them, possibly incorrectly.

17.6.5 When Are Inputs and Outputs Needed?

> Machines live in the real world and have only a limited contact with it. Any machine, no matter how universal, that has no ears (so to speak) will not hear; that has no wings, will not fly. —Allen Newell (1980, p. 148)

Narrowly conceived, algorithms might not need inputs and outputs. Widely conceived, they do. Any input from the external world has to be *encoded* by a user into a language "understandable" by the Turing machine (or else the Turing machine needs to be able to *decode* such external-world input). And any output *from* the Turing machine to be reported *to* the external world (for example, a user) has to be *encoded* by the Turing machine (or *decoded* by the user). Such codings would, themselves, have to be algorithmic.

In fact, one key to determining which real-world tasks are computable—one of CS's main questions (§3.15.2)—is finding coding schemes that allow a sequence of '0's and '1's (that is, a natural number in binary notation) on a Turing machine's tape to be *interpreted* as a symbol, a pixel, a sound, etc. According to the Computability Thesis, a mathematical function on the natural numbers is computable iff it is computable by a Turing machine. Thus, a real-world problem is computable iff it can be encoded as such a computable mathematical function.

But it's that wide conception, requiring algorithmic, semantic interpretations of the inputs and outputs, that leads to various debates. Let's look at the (semantic) coding issue more closely.

17.6.6 Must Inputs and Outputs Be Interpreted Alike?

Letting the symbol '$\underline{x}$' represent a sequence of x strokes (where x is a natural number), Rescorla (2007, p. 254) notes that

> Different textbooks employ different correlations between Turing machine syntax and the natural numbers. The following three correlations are among the most popular:
>
> $$d_1(\underline{n}) = n.$$
>
> $$d_2(\underline{n+1}) = n.$$
>
> $$d_3(\underline{n+1}) = n, \text{as an input.}$$
>
> $$d_3(\underline{n}) = n, \text{as an output.}$$
>
> A machine that doubles the number of strokes computes $f(n) = 2n$ under d_1, $g(n) = 2n+1$ under d_2, and $h(n) = 2n+2$ under d_3. Thus, the same Turing machine computes different numerical functions relative to different correlations between symbols and numbers.

Let's focus on interpretation d_3. First, having different input and output interpretations of a single internal formalism occurs elsewhere. Machine-translation systems that use an "interlingua" work this way: Chinese input, for example, can be encoded into an "interlingual" representation language (often thought of as an internal, "meaning"-representation language that encodes the "proposition" expressed by the Chinese input), and English output can then be generated from that interlingua (re-expressing in English the same proposition that was originally expressed in Chinese). Cognition (assuming that it is computable!) also works this way: Perceptual encodings (such as Newell's example of hearing) into the "interlingua" of the biological neural network of our brain surely differ from motor decodings (such as Newell's example of flying).

Further Reading:
For an example using the SNePS knowledge-representation and reasoning system as an interlingua between Chinese and English, see Liao 1998. For more on interlinguas, see Slocum 1985 and Daylight 2013, §2.

Second, using our formulation from §17.5, the idea that a single, internal representation scheme can have different external interpretations suggests that the internal *A* can be considered separately from external *G*s and that it is the internal *A* that is central to computation.

This offers a way out of Rescorla's puzzle about the two GCD computers.[7] Consider a Common Lisp version of Rescorla's GCD program. The Common Lisp version will look identical to the Scheme version shown in §14.4.3 (the languages share most of their syntax), but the Common Lisp version has two global variables—`*read-base*` and `*print-base*`—that tell the computer how to interpret input and how to display output. These are implementations of the coding algorithms mentioned in §17.6.5. By default, `*read-base*` is set to 10. So the Common Lisp read-procedure does the following:

(a) It sees the three-character string '115' (for example);

(b) it decides that the string satisfies the syntax of an integer;

(c) it converts that string of characters to an internal ("interlingual") representation of type `integer`—*which is represented internally as a binary numeral implemented as bits or switch-settings*

(d) it does the same with (say) '20'; and

(e) it computes their GCD using the algorithm from §14.4.3 *on the binary representation.*

If the physical computer had been an old IBM machine, the computation might have used binary-coded *decimal* numerals instead, thus computing in base 10. If `*read-base*` had been set to 13, the input characters would have been interpreted as base-13 numerals, and the very *same* Common Lisp (or Scheme) code would have correctly computed

[7] I am indebted to Stuart C. Shapiro, personal communication, for the ideas in this paragraph.

the GCD of 187_{10} and 26_{10}. One could either say that the algorithm computes with *numbers*—not numerals—or that it computes with *base-2 numerals as an interlingual (or "canonical") representation of numbers*. But that choice depends on one's view about the nature of *mathematics—not* about the nature of *computation*.

Digression and Further Reading about the Nature of Mathematics:
"Platonists" believe that mathematics deals with *numbers*—abstract entities that exist in a "Platonic" realm that is more perfect than, and independent of, the real world (see §12.4.2). "Nominalists", on the other hand, deny the existence of abstract numbers, and hold that mathematics deals only with *numerals*—real marks on paper, for instance. For more on this, see Bueno 2014 and Linnebo 2018.

And similarly for output: The switch-settings containing the GCD of the input are then output as base-10 or base-13 numerals appearing as pixels on a screen or ink on paper, depending on the value of such things as `*print-base*`. With respect to Rescorla's example, the point is that a *single* Common Lisp (or Scheme) *algorithm* is being executed correctly by *both* M_{10} *and* M_{13}. Those *machines are* different; they do *not* "have the same local, intrinsic, physical properties" (Rescorla, 2013, p. 687), because M_{10} has `*read-base*` and `*print-base*` set to 10, whereas M_{13} has `*read-base*` and `*print-base*` set to 13.

For a purely mathematical, Turing-computable example, recall Aizawa's program from §10.4.1, repeated here:

> ... if we represent the natural number n by a string of n consecutive 1s, and start the program with the read-write head scanning the leftmost 1 of the string, then the program,
>
> q_0 1 1 R q_0
> q_0 0 1 R q_1,
>
> will scroll the head to the right across the input string, then add a single '1' to the end. It can, therefore, be taken to compute the successor function.
> (Aizawa, 2010, p. 229)

I can describe this program semantically (or "widely") as one that generates natural numbers. Speaking purely syntactically (or "narrowly"), I'd like to describe it as one that appends a '1' to the (right) end of the sequence of '1's encoded on its (input) tape. Using our formulation from §17.5, the semantic or wide (or "intentional") description would be:

> In order to generate the natural numbers, do **begin** $q_0 11 R q_0$; $q_0 01 R q_1$ **end.**

The syntactic (or narrow) description would just be the "do" clause.

Now, a Turing machine that does merely that does not really generate the natural numbers; at best, it could be described semantically as a one-trick pony that generates the successor of the number encoded on the tape. To generate "all" natural numbers, this Turing machine would have to be embedded as the body of a loop in another, "larger" Turing machine. The idea is that, beginning with a tape "seeded" with the first

natural number (either a blank tape or one with a single stroke), it executes the first Turing machine, thus generating the input's successor, then loops back to the beginning, considers the current tape as the input tape, and generates its successor, *ad infinitum*.

But what if it uses Rescorla's d_3 interpretation scheme? Then our larger Turing machine, while still appending a '1' to the end of the current sequence of '1's on the tape, is no longer generating the natural numbers. (It is certainly generating *a* natural-number sequence, but not the one written in the same notation as the inputs.) Rather than computing $S(n) = n+1$, it is computing $S'(n) = n+2$.

The aspect of this situation that I want to remind you of is whether the tape is the *external* input and output device, or is, rather, the machine's *internal* memory. If it is the machine's internal memory, then, in some sense, there is no (visible or user-accessible) input or output (§17.6.2). If it is an external input-output device, then the marks on it are for *our* convenience only. In the former case, the only accurate description of the Turing machine's behavior is syntactically in terms of '1'-appending. In the latter case, we can use that syntactic description but we can also embellish it with one in terms of our interpretation of what it is doing. (We'll return to this in §17.8.1.)

17.6.7 Linking the Tape to the External World

If a Turing machine's tape is really just its internal memory, then, even though Turing machines compute mathematical functions; they "contemplate their navel", so to speak, and don't tell us what their results are. If we want to *use* a Turing machine to find out the result of a computation, we need to look at its internal storage. Conveniently, it's visible on the machine's tape. But it's in code. So we have to decode it into something that we can understand and use. And we have to do that algorithmically. Our examples above suggest that this can be done in many ways and that it can go wrong.

Does that decoding belong to *A*? Or does it belong to *G*? Let's now turn to this question.

17.7 Are Programs Intentional?

We have discussed two ways to view a computation: The first way is purely syntactically (or narrowly, internally, or locally), as expressed in a computer program of the form "Do *A*", where *A* is an algorithm expressed in the language of Turing machines or the language of recursive functions (etc.). The second way is semantically (or widely, externally, or globally), as expressed in a computer program of the form "In order to accomplish goal *G*, do *A*" (which we'll shorten to "To *G*, do *A*"). That preface ("To *G*") makes explicit a *goal G* of the algorithm *A*, thus indicating that the program is *intended* to have a *purpose*: It is viewed as "teleological" or "intentional". Let's now consider the question of whether the proper way to characterize a program *must* include the intentional or teleological preface "To *G*".

17.7.1 What Is an Algorithm?

As we saw in Chapter 7, the history of computation theory is, in part, an attempt to make the informal notion of an algorithm mathematically precise. In §7.4.3.4, we summarized this as follows:

> An algorithm (for executor *E*) [to accomplish goal *G*] is:
>
> 1. a procedure *A*, that is, a finite set (or sequence) of statements (or rules, or instructions), such that each statement *S* is:
>   (a) composed of a finite number of symbols (better: uninterpreted marks) from a finite alphabet
>   (b) and unambiguous (for *E*—that is,
>   i. *E* "knows how" to do *S*,
>   ii. *E* can do *S*,
>   iii. *S* can be done in a finite amount of time
>   iv. and, after doing *S*, *E* "knows" what to do next—),
> 2. *A* takes a finite amount of time (that is, it halts),
> 3. [and *A* ends with *G* accomplished].

I have put some of these clauses in (parentheses) and [brackets] for a reason.

17.7.2 Do Algorithms Need a Purpose?

> Algorithms, **in the popular imagination**, are algorithms *for* producing a particular result. ... [E]volution can be an algorithm, and evolution can have produced us by an algorithmic process, without its being true that evolution is an algorithm *for* producing us. —Daniel C. Dennett (1995, p. 308, my boldface, original italics)[8]

The notion of an algorithm is most easily understood with respect to an executor: a human or a machine that (dynamically) executes the (static) instructions. We might be able to rephrase the above characterization of an algorithm without reference to *E*, albeit awkwardly, hence the *parentheses* around the *E*-clauses.

Exercise for the Reader:
Try to eliminate the executor from this (or any other) characterization of an algorithm. Can it be eliminated? If not, why not?

But the present issue is whether the *bracketed G*-clauses are essential. As we saw in §§12.4.5 and 16.2, one executor's algorithm might be another's ungrammatical input (Suber, 1988; Buechner, 2011, 2018). Recall from the digression in §16.2 that a bad puttanesca might still be a delicious pasta dish. Does the chef's intention (or the diner's expectation) matter *more* than the actual food preparation? Is *G* more important than *A*?

Both Peter Suber and Robin K. Hill argue in favor of the importance of *G*:

[8] For hints as to what evolution's algorithms look like, see Dawkins 2016.

> To distinguish crashes and bad executions from good executions, it appears that we must introduce the element of the programmer's purpose. Software executes the programmer's will, while semantically flawed, random, and crashing code do not. This suggests that to understand software we must understand intentions, purposes, goals, or will, which enlarges the problem far more than we originally anticipated.
>
> Perhaps we must live with this enlargement. We should not be surprised if human compositions that are meant to make machines do useful work should require us to posit and understand human purposiveness. After all, to distinguish *literature* from noise requires a similar undertaking. (Suber, 1988, p. 97)

And Hill (2016b, §5) says that a "prospective user" needs "some understanding of the task in question" over and above the mere instructions. Algorithms, according to Hill, must be expressed in the form "To *G*, do *A*", not merely "Do *A*".

Question for the Reader:
Is the *executor* of an algorithm the same as a *user*? Typically, a (human) *uses* a computer, but it is the *computer* that executes the algorithm. In what follows, ask yourself if it is the user or the executor who "needs some *understanding* of the task" (as Hill says).

17.7.3 Marr's Analysis of an Algorithm's Purpose

Suber and Hill are not alone in this. The cognitive scientist and computational vision researcher David Marr also held that (at least some) computations were purposeful. He analyzed information processing into three levels (Marr, 1982):

- computational (*what* a system does),
- algorithmic (*how* it does it), and
- physical (how it is *implemented*).

In our terminology, these levels would be called 'functional', 'computational', and 'implementational', respectively: Certainly, when one is doing mathematical computation (the kind that Turing was concerned with), one begins with a mathematical *function* (that is, a certain set of ordered pairs), asks for an algorithm to *compute* it, and then seeks an *implementation* of the algorithm, usually in a physical system such as a computer or the brain.

In *non*-mathematical fields (for example, cognition in general, and—for Marr—vision in particular), the set of ordered pairs of input-output *behavior* is expressed in goal-oriented, problem-specific language, and the algorithmic level will also be expressed in that language. (The implementation level might be the brain or a computer.) A recipe for hollandaise sauce developed in this way would have to say more than just something along the lines of "mix these ingredients in this way"; it would have to take the external environment into account. (We will return to this in §17.7.6, and we will see how the external world can be taken into account in §17.8.2.4.)

Marr was trying to counter the then-prevailing methodology of trying to *describe* what neurons were doing (a "narrow", internal, implementation-level description) without having a "wide", external, "computational"-level *purpose* (a "function" in the teleological, not mathematical, sense). Such a teleological description would tell us "why" neurons behave as they do:

> As one reflected on these sorts of issues in the early 1970s, it gradually became clear that something important was missing that was not present in either of the disciplines of neurophysiology or psychophysics. The key observation is that neurophysiology and psychophysics have as their business to *describe* the behavior of cells or of subjects but not to *explain* such behavior. What are the visual areas of the cerebral cortex actually doing? What are the problems in doing it that need explaining, and at what level of description should such explanations be sought? (Marr, 1982, p. 15; for discussion of this point, see Bickle 2015)

On this view, Marr's "computational" level is *teleological*. In the formulation "To *G*, do *A*", the "To *G*" preface expresses the teleological aspect of Marr's "computational" level; the "do *A*" seems to express Marr's "algorithm" level.

The philosopher Frances Egan (Egan 1991, pp. 196–197; Egan 1995, p. 185) takes the mathematical functional view just outlined, focusing (in our terminology) on *A*, not *G*. On that view, Marr's "computational" level is *not* intentional (Egan, 1991, p. 201). Barton L. Anderson (2015, §1), on the other hand, says that Marr's "computational" level

> concern[s] the presumed *goal* or *purpose* of a mapping,[9] that is, the specification of the 'task' that a particular computation 'solved.' Algorithmic level questions involve specifying how this mapping was achieved computationally, that is, the formal procedure that transforms an input representation into an output representation.

Shagrir and Bechtel (2015, §2.2) suggest that Marr's "computational" level conflates two separate, albeit related, questions: not only "why", but also "what". On this view, Egan is focusing on the "what", whereas Anderson is focusing on the "why". (We will return to Marr in §§17.7.6 and 17.9.)

17.7.4 Are Purposes Eliminable?

Certainly, knowing what the goal of an algorithm is makes it easier for *cognitive-agent* executors (who are also users?) to follow the algorithm and to have a fuller understanding of what they are doing. But is such understanding *necessary*? Consider the following two (real-life!) personal stories:

Story 1

> I vividly remember the first semester that I taught a "Great Ideas in Computer Science" course aimed at computer-phobic students. We were going to teach the students how to use a spreadsheet program, something that, at the time, I had

[9] Presumably, a mathematical function.

never used! So, with respect to this, I was as naive as any of my students. My TA, who had used spreadsheets before, gave me something like the following instructions:

> enter a number in cell_1;
> enter a number in cell_2;
> enter '=⟨click on cell_1⟩⟨click on cell_2⟩' in cell_3

Now, some current implementations of Excel require a plus-sign between the two clicks in the third instruction. But the version I was using at the time did not, making the operation that much more mysterious! Indeed, I had no idea what I was doing. I was blindly following her instructions *and had no idea that I was adding two integers*. Once she told me that that was what I was doing, my initial reaction was "Why didn't you tell me that before we began?".

When I entered those data into the spreadsheet, was I adding two numbers? I didn't understand that I was adding when my TA told me to enter certain data into the cells of the spreadsheet. It was only when she told me that that was how I could add two numbers with a spreadsheet that I understood. Now, (I like to think that) I am a cognitive agent who can come to understand that entering data into a spreadsheet can be a way of adding. But a Turing machine that adds or a Mac running Excel is not such a cognitive agent. It does not understand what addition is or that that is what it is doing. *And it does not have to.*

Further Reading:
However, an AI program running on a robot that passes the Turing test would be a very different matter. Such an AI program could, would, and should (come to) understand what it was doing. We'll explore this further in Chapter 19. See also Rapaport 1988a, 2012b, and Albert Goldfain's work on how to get AI computer systems to *understand* mathematics in addition to merely *doing* it (Goldfain, 2006, 2008).

Story 2 Years later, I had yet another experience along these lines:

> My wife recently opened a restaurant and asked me to handle the paperwork and banking that needs to be done in the morning before opening (based on the previous day's activities). She wrote out a detailed set of instructions, and one morning I went in with her to see if I could follow them, with her looking over my shoulder. As might be expected, there were gaps in her instructions, so even though they were detailed, they needed even more detail. Part of the reason for this was that she knew what had to be done, how to do it, and why it had to be done, but I didn't. This actually disturbed me, because I tend to think that algorithms should really be just "Do A," not 'To G, do A." Yet I felt that I needed to understand G in order to figure out how to do A. But I think the reason for that was simply that she hadn't given me an algorithm, but a sketch of one, and, in order for me to fill in the gaps, knowing why I was doing A would help me fill in those gaps. But I firmly believe that if it made practical sense to fill in all those gaps (as it would if we were writing

> a computer program), then I wouldn't have to ask why I was doing it. No "intelligence" should be needed for this task if the instructions were a full-fledged algorithm. If a procedure (a sequence of instructions, including vague ones like recipes) is not an algorithm (a procedure that is fully specified down to the last detail), then it can require "intelligence" to carry it out (to be able to fill in the gaps, based, perhaps on knowing why things are being done). If intelligence is not available (i.e., if the executor lacks relevant knowledge about the goal of the procedure), then the procedure had better be a full-fledged algorithm. There is a difference between a human trying to follow instructions and a machine that is designed to execute an algorithm. The machine cannot ask why, so its algorithm has to be completely detailed. But a computer (or a robot, because one of the tasks is going to the bank and talking to a teller!) that could really do the job would almost certainly be considered to be "intelligent."
> (Rapaport, quoted in Hill and Rapaport 2018, p. 35)

Despite the fact that understanding what task *G* that an algorithm *A* is accomplishing makes it easier to understand *A* itself, the important point is that "blind" following of *A* is all that is necessary in order to *accomplish G*. The fact that computation can be "blind" in this way is what Dennett has called

> Turing's ... strange inversion of reasoning. The Pre-Turing world was one in which computers were people, who had to understand mathematics in order to do their jobs. Turing realised that this was just not necessary: you could take the tasks they performed and squeeze out the last tiny smidgens of understanding, leaving nothing but brute, mechanical actions. IN ORDER TO BE A PERFECT AND BEAUTIFUL COMPUTING MACHINE IT IS NOT REQUISITE TO KNOW WHAT ARITHMETIC IS. (Dennett, 2013b, p. 570, capitalization in original)[10]

The point is that a Turing machine need not "know" that it is adding. But agents who do understand adding can use that machine to add.

Or can they? In order to do so, the machine's inputs and outputs have to be interpreted—understood—by the user as representing the numbers to be added. And that seems to require an appropriate relationship with the external world. It seems to require a "user manual" that tells the user what the algorithm does in the way that Hill prescribes, not in the way that my TA explained how to use a spreadsheet. And such a "user manual"—an intention or a purpose for the algorithm—in turn requires an interpretation of the machine's inputs and outputs.

But before pursuing this line of thought, let's take a few more minutes to consider "Turing's inversion", the idea that a Turing machine can be doing something very particular by executing an algorithm without any specification of what that algorithm is "doing" in terms of the external world. (We'll return to "Turing's strange inversion" in Chapter 19.) Algorithms, on this view, seem not to have to be intentional or teleological, yet they remain algorithms.

Brian Hayes (2004) offers two versions of an algorithm that ants execute:

[10] See also the more easily accessible Dennett 2009a, p. 10061.

non-teleological version:

1. "If you see a dead ant[11] and you're not already carrying one, pick it up;
2. "if you see a dead ant, and you *are* carrying one, put yours down near the other."

teleological version:

To create an ant graveyard, "gather all ... [your] dead in one place.[12]

As Hayes notes, the teleological version requires planning and organization skills far beyond those that an ant might have, not to mention conceptual understanding that we might very well be unwilling to ascribe to ants. The point, however, is that the ant needs none of that. The teleological description helps *us* describe and perhaps understand the ant's behavior; it doesn't help the ant.

The same is true in my spreadsheet example. Knowing that I am adding helps *me* understand what I am doing when I fill the spreadsheet cells with certain values or formulas. But the spreadsheet does its thing without needing that knowledge.

These examples suggest that the user-manual (or external-world) interpretation is not necessary. Algorithms *can* be teleological, and their being so can help users and cognitive agents who execute them to more fully understand what they are doing. But they don't *have to* be teleological.

17.7.5 Can Algorithms Have More than One Purpose?

In addition to being teleological, algorithms seem to be able to be *multiply* teleological, as in the chess-war example and its kin. That is, there can be algorithms of the form:

To G_1, do A

and algorithms of the form:

To G_2, do A

where $G_1 \neq G_2$, and where neither G_1 nor G_2 subsumes the other, although the A is the same. In the cartoon of Figure 17.6, depending on the semantic interpretation of the syntactic finger movements, we have two Gs with one A: *either* "In order to say 'yes' ten times, raise 10 fingers" *or* "In order to say 'no' five times, raise 10 fingers".

In other words, what if doing A can accomplish two distinct goals? Do we have two algorithms in that case? (One that accomplishes G_1, and another that accomplishes G_2, counting teleologically, or "widely".) Or just one? (A single algorithm that does A, counting more narrowly.)

Were de Bruijn and the chemists talking about the same thing? On the teleological (or wide) view, they weren't; on the narrow view, they were. Multiple teleologies are multiple implementations of an algorithm narrowly construed: 'Do A' can

[11] Note that testing this condition does not require the ant to have a concept of death; it is sufficient for the ant to sense—either visibly or perhaps chemically—what *we* would describe as a dead ant.

[12] This is a "fully" teleological version, with a high-level, teleologically formulated execution statement. A "partially" teleological version would simply prefix "To create an ant graveyard" to the non-teleological version.

be seen as a way to algorithmically implement the higher-level "function" (mathematical *or* teleological) of accomplishing G_1 *as well as* accomplishing G_2. For example, executing a particular subroutine in a given program might result in checkmate or winning a battle. Viewing multiple teleologies as multiple implementations can also account for hollandaise-sauce failures on the Moon, which could be the result of an "implementation-level detail" (§14.2.1) that is irrelevant to the abstract, underlying computation.

17.7.6 What If *G* and *A* Come Apart?

What if "successfully" executing *A fails* to accomplish goal *G*? This could happen for external, environmental reasons. Does this mean that *G* might not be a computable task even though *A* is? We have seen several examples of this kind of failure:

- The blocks-world computer's model of the world was an incomplete, partial model; it assumed that its actions were always successful. This program lacked feedback from the external world. There was nothing wrong with the environment; rather, there was incomplete information about the environment.
- In the case of Cleland's hollandaise-sauce recipe, the environment was at fault. Her recipe (*A*) was executed flawlessly on the Moon, but failed to produce hollandaise sauce. Her diagnosis was that making hollandaise sauce (*G*) is not computable. Yet *A* was!
- Rescorla's GCD computers do "different things" *by* doing the "same thing". The difference is not in *how* they are doing what they are doing, but in the interpretations that *we* users of the machines give to their inputs and outputs. Would Hill (2016b) say that the procedure encoded in that Scheme program was therefore not an algorithm?

Question for the Reader:
How does this relate to the trial-and-error machines that we discussed in §11.4.5? After all, they also differ from Turing machines only in terms of our *interpretations* of *what* they are doing, not in *how* they do it.

Further Reading:
Rescorla 2015b, §2.2, considers the opposite case, in which *G is* computable even when *A* is *not*: "There exist 'deviant' notations relative to which intuitively non-computable functions become Turing-computable".

What is more central to the notion of "algorithm": all of parts 1–3 in our informal characterization in §17.7.1 ("To *G*, do *A*"), or just parts 1–2, that is, without the bracketed goals (just "Do *A*")? Is the algorithm the narrow, non-teleological, "purposeless" (or non-purposed) entity? Or is the algorithm the wide, intentional, teleological (that is, goal-directed) entity?

On the narrow view, the war and chess algorithms are just *one* algorithm, the hollandaise-sauce recipe *does* work on the Moon (its computer program might be logically verifiable even if it fails to make hollandaise sauce), and Rescorla's "two" GCD

programs are also just *one* algorithm that does its thing correctly (but only we base-10 folks can *use* it to compute GCDs).

On the wide view, the war and chess programs are *two*, distinct algorithms, the hollandaise-sauce recipe *fails* on the Moon (despite the fact that the program might have been verified—shades of the Fetzer controversy that we discussed in §16.5.1!), and the Scheme program when fed base-13 numerals (as Rescorla describes it) is doing something *wrong* (in particular, its "remainder" subroutine is incorrect). It does the *right* thing on the interpretation discussed in §17.6.6.

These examples suggest that the wide, goal-directed nature of algorithms that are teleologically conceived is due to the interpretation of their input and output. As Shagrir and Bechtel (2015, §2.3) put it (echoing Sloman's distinction from §17.6.1), Marr's "algorithmic level ... is directed to the *inner working* of the mechanism .... The computational level looks *outside*, to identifying the function computed and relating it to the environment in which the mechanism operates".

We can combine these insights: Hill's formulation of the teleological or intentional nature of algorithms had two parts, a teleological "preface" specifying the task to be accomplished ("To *G*"), and a statement of the algorithm that accomplishes it ("Do *A*"). One way to clarify the nature of Marr's "computational" level is to split it into its "why" and its "what" parts. The "why" part is the task to be accomplished. The "what" part can be expressed "computationally" (in our terminology, "functionally") as a mathematical function (possibly, but not necessarily, expressed in "why" terminology), but it can also be expressed *algorithmically*. Finally, the algorithm can be implemented. So, we can distinguish the following *four* Marr-like levels of analysis:

"Computational"-What Level: Do $f(i) = o$

"Computational"-Why Level: To G, do $f(i) = o$

Algorithmic Level: To G, do $A_f(i) = o$

Implementation Level: To G, do $I_{A_f}(i) = o$

where:

- f is an *i*nput-*o*utput function that happens to accomplish G;
- G is the task to be accomplished or explained, expressed in the language of the external world, so to speak;
- A_f is an algorithm that implements f (that is, it is an algorithm that has the same input-output behavior as f), expressed either in the same language as G or perhaps expressed in purely mathematical language; and
- I is an implementation (perhaps in the brain or on some computer) of A_f.

Shagrir and Bechtel (2015, §4) say that "The *what* aspect [of the "computational" level] provides a description of the mathematical function that is being computed. The *why* aspect employs the contextual constraints in order to show how this function matches with the environment." These nicely describe the two clauses of what I call the "computational-why" level above.

Further Reading:
Turner 2019 is a useful discussion of the intentionality or purposefulness of computer programs in the context of program verification.

17.8 Do We Compute with Symbols or with Their Meanings?

Goal G is expressed in intentional language. We now need to focus on the language used to express the algorithm A_f that implements the function f that—in turn—underlies (or is supposed to accomplish) G. *Can* it be intentional? *Must* it be intentional, too? In other words, can (or must) it be expressed in the language of G? For example, can (must) it talk about chess as opposed to war, or chess as opposed to shogi or Go?

17.8.1 What Is This Turing Machine Doing?

What do Turing machines compute with? For that matter, what do *we* compute with? Rescorla (2007, p. 253) reminds us that

> A Turing machine manipulates syntactic entities: strings consisting of strokes and blanks. ... Our main interest is not string-theoretic functions but number-theoretic functions. We want to investigate computable functions from the natural numbers to the natural numbers. To do so, we must correlate strings of strokes with numbers.

In this regard, Turing machines differ interestingly from their logical equivalents in the Computability Thesis: The lambda calculus and recursive-function theory deal with functions and numbers, not symbols for them.

Questions for the Reader:
Is it really the case that the lambda calculus and recursive-function theory (unlike Turing machines) deal with functions and not just with symbols for functions? Hilbert viewed all of mathematics as the "manipulation of finite strings of symbols devoid of intuitive meaning[,] which stimulated the development of mechanical processes to accomplish this" (Soare, 1999, §2.4, p. 5). On this view, wouldn't *all* of the formalisms of computability theory be syntactic? Can't recursive-function theory be understood purely syntactically? And the lambda calculus "can be presented solely as a formal system with syntactic conversion rules. ... [A]ll we are doing is manipulating symbols" (J. Stoy, quoted in Turner 2018, p. 92).

But for Turing machines and their physical implementations (that is, ordinary computers), we see that it is necessary to interpret the strokes. Here is an example due to the philosopher Christopher Peacocke (1999): Suppose that we have a Turing machine that outputs a copy of the input appended to itself (thus doubling the number of input strokes): input '|', output '||'; input '||', output '||||', and so on. What is this Turing machine doing? Isn't "outputting a copy of the input appended to itself" the most neutral description? After all, that describes *exactly* what the Turing machine is doing,

leaving the interpretation (for example, *doubling* the input) up to the observer. If we had come across that Turing machine in the middle of the desert and were trying to figure out what it does, something like that would be the most reasonable answer.[13] *Why* a user might want a copy-appending Turing machine is a different matter that probably *would* require an interpretation of the strokes. But that goes far beyond what the Turing machine is doing.

But Peacocke objects:

> The normal interpretation of a Turing machine assigns the number 0 to a single stroke '|', the number 1 to '||', the number 2 to '|||', and so on. But there will equally be an interpretation which assigns 0 to a single stroke '|', and then assigns the number 2 to '||', the number 4 to '|||', and generally assigns $2n$ to any symbol to which the previous interpretation assigns n. Under the second interpretation, the Turing machine will still be computing a function. ... What numerical value is computed, and equally which function is computed, by a given machine, is not determined by the purely formal characterization of the machine. There is no such thing as purely formal determination of a mathematical function. ... [W]e can say that a Turing machine is really computing one function rather than another only when it is suitably embedded in a wider system. (Peacocke, 1999, pp. 198–199).

Recall Rescorla's three interpretations of the strokes (§17.6.6). Do we really have one machine that does three different things? What it does (in one sense of that phrase) depends on how its input and output are interpreted, that is, on the environment in which it is working. In different environments, it does different things; at least, that's what Cleland said about the hollandaise-sauce recipe. Rescorla (2015b, §2.1) makes a related observation: "The same Turing machine T computes different non-linguistic functions, depending upon the semantic interpretation of strings manipulated by the Turing machine", thus rendering all of computability theory "intensional", that is, dependent upon the *meanings* of the symbols and not just on the symbols themselves (for example, their shapes). Using our terminology, he thus comes down on the side of "To G, do A" rather than on "Do A".

Piccinini (2006a, §2, my italics) says much the same thing; however, he draws a different conclusion:

> In computability theory, symbols are typically marks on paper individuated by their geometrical shape (as opposed to their semantic properties). Symbols and strings of symbols may or may not be assigned an interpretation; if they are interpreted, the same string may be interpreted differently .... In these computational descriptions, *the identity of the computing mechanism does not hinge on how the strings are interpreted.*

By 'individuated', Piccinini is talking about how one decides whether what appear to be two programs (say, one for a war battle and one for a chess match) are, in fact, two distinct programs or really just one program (perhaps being described differently). He suggests that it is not how the inputs and outputs are *interpreted* (their semantics) that matters, but what the inputs and outputs *look like* (their syntax). In an earlier paper, Rescorla agreed:

[13] Recall our previous excursions into the desert in §§3.9.5, 6.5.1, 9.5.3, and 14.4.3.

> Since we can arbitrarily vary inherited meanings relative to syntactic machinations, inherited meanings do not *make a difference* to those machinations. They are imposed upon an underlying causal structure. (Rescorla, 2014a, p. 181)

So, for Piccinini and Rescorla 2014a, the war and chess programs are the same. But for Cleland and Rescorla 2015b, they would be different. For Piccinini, the hollandaise-sauce program running on the Moon works just as well as the one running on Earth; for Cleland, only the latter does what it is supposed to do.

So, the question "*Which* Turing machine is this?" has only one answer, which is given in terms of its syntax: "determined by [its] instructions, not by [its] interpretations" (Piccinini, 2006a, §2). But the question "*What* does this Turing machine do?" has $n+1$ answers: one syntactic answer and n semantic answers (one for each of n different semantic interpretations).

If I want to know which Turing machine this is, I should look at the internal mechanism (A) for the answer. This is, roughly, Piccinini's (2006a) recommendation. But if I'm interested in buying a chess program (as opposed to a war simulator, for example), then I need to look at the external (or inherited, or wide) semantics. This would be Cleland's (1993) recommendation. In "To G, do A", the "do A" portion expresses Dennett's (1971) "design" stance, and the "to G" portion expresses Dennett's "intentional" stance (Dennett 2013a, pp. 81–82, 84; recall our §12.4.4.1.1, above).

We have come across this situation before. In §12.4.4.1.2.2, we asked whether a universal Turing machine running an addition program was adding or "just" fetching and executing the instructions of an addition program stored on its tape A similar question can be asked about humans: How would you describe my behavior when I use a calculator to add two numbers? Am I (merely) pushing certain buttons in a certain sequence? This would be a "syntactic", narrow, internal answer: I am "doing A" (where A = pushing buttons). Or am I adding two numbers? This would be a teleological, "semantic", wide, external answer: I am accomplishing G (where G = adding). Or am I adding two numbers *by* pushing those buttons in that sequence? This would be a teleological (etc.) answer, together with a syntactic description of how I am doing it: I am accomplishing G, by doing A. This is the same situation that we saw in the spreadsheet example. (We will see it again in §17.8.2.2).

In some sense, all of these answers are correct, merely(?) focusing on different aspects of the situation. But a further question is: *Why* (or how) does a Turing machine's printing and moving thus and so, or my pushing certain calculator buttons thus and so, result in adding two numbers? And the answer to that seems to require a semantic interpretation. This is the kind of question that Marr's "computational" level is supposed to respond to.

Here is another nice example (Piccinini, 2008, p. 39):

> a loom programmed to weave a certain pattern will weave that pattern regardless of what kinds of thread it is weaving. The properties of the threads make no difference to the pattern being woven. In other words, the weaving process is insensitive to the properties of the input.

As Piccinini points out, the output might have different colors depending on the colors of the input threads, but the *pattern* will remain the same. The pattern is internal to the

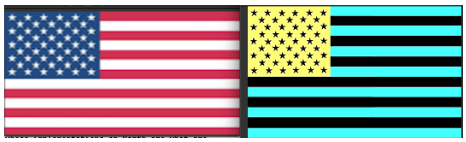


Figure 17.7: Two *flags*? Or one flag *pattern*?

program; the colors are external, to use other terminology. (Here, A is the pattern; G is the colors.) If you want to weave an American flag, you had better use red, white, and blue threads in the appropriate ways. But even if you use cyan, black, and yellow threads, you will weave an American-flag *pattern* (see Figure 17.7). Which is more important: the pattern or the colors? That's probably not the right question. Rather, if you want a certain *pattern*, this program will give it to you; if you want a certain pattern *with certain colors*, then you need to have the right inputs—you need to use the program in the right environment.

Further Reading:
This distinction also appears in the philosophy of mathematics concerning "structuralism" (see §§9.5.6 and 14.2.2): Is the pattern, or structure, of the natural numbers all that matters? Or does it also matter what the natural numbers in the pattern "really" are? For discussion, see Benacerraf 1965 and Horsten 2015, §4.

17.8.2 Syntactic Semantics

17.8.2.1 Syntax vs. Semantics

Recall the concepts of syntax and semantics as we discussed them in Chapter 14. Syntax is concerned with the "*intra*-system" properties and relations *within* the "syntactic" domain. Semantics is concerned with "*extra*-system" relations that go *beyond* the syntactic domain *to* the "semantic" domain. That is, semantics is concerned with the "*inter*-system" relations *between* the syntactic and the semantic domains.[14]

So, *one* way to respond to the issues raised in §17.8.1 is by using an external semantic interpretation: Begin with specific Turing-machine operations or button presses, considered as being located in a syntactic system of internal Turing-machine operations or button pressings. Numbers and arithmetical operations on them are located in a distinct, external realm of mathematical entities. Then we can associate the former with the latter. In the formulation "To G, do A", A *can* be identified syntactically (at the "computational-what" level)—in terms, say, of Turing-machine operations or button pressings. But G *needs* to be identified semantically—in terms, say, of numbers and arithmetic operations. A can then be (re-)*interpreted* semantically in G's terms (at the "computational-why" level). These are the $n+1$ answers of §17.8.1.

[14] '*Intra*-' means "inside", '*extra*-' means "outside", and '*inter*- means "between". (In "*inter*mural" sports, school A plays against school B. In "*intra*mural" sports, a single gym class at school A might be divided into two teams that play against each other.)

17.8.2.2 Syntactic Semantics

But *another* way to respond to these issues uses an "internal" kind of semantics. Because this kind of semantics is internal to a system, it is really a kind of syntax. Let's call it "syntactic semantics". Here is how Piccinini describes it:

> [S]tored-program computers have the ability to respond to (non-semantically individuated) strings of tokens stored in their memory by executing sequences of primitive operations, which in turn generate new strings of tokens that get stored in memory. [Note that this is basically a description of how computers work, or of what computation is—note added by WJR.] Different bits and pieces [that is, substrings—WJR] of these strings of tokens have different effects on the machine. ... An accurate description of how tokens can be compounded into sub-strings, and sub-strings can be compounded into strings, which does not presuppose that the strings of tokens have any content, may be called the syntax of the system of strings manipulated by the computer. ... [T]he effect of a string on the computer is assigned to it [that is, to the string—WJR] as its content. This assignment constitutes an *internal* semantics of a computer. An internal semantics assigns as contents to a system its own internal components and activities, whereas an ordinary (external) semantics assigns as contents to a system objects and properties in the system's environment. ... None of this entails that computer languages have any external semantics, that is any content ..., although it is compatible with their having one. ...
>
> [I]n order to understand computing mechanisms and how they work (as opposed to why they are built and how they are used), there is no need to invoke content ....
>
> (Piccinini 2004b, pp. 401–402, 404; my interpolated brackets. See also Piccinini 2006a, §2.)

On this view, it is the "internal" workings of the computer that count, not the external interpretation of its inputs and outputs (or even the external interpretation of its internal mechanisms or symbol manipulations). This is the sense in which a war computer and a chess computer are performing "the same computation".

Note the parenthesized hedge in the last sentence of Piccinini's quote: Cleland and Rescorla might be quite right in terms of their emphasis on why or how a particular computer or program is being *used*. That's an intentional aspect of computation, but doesn't necessarily violate the Computability Thesis.

Similarly, Rescorla once argued "that computation is not sensitive to meaning or semantic properties" (2012a, §1, p. 703). More precisely, he argued that *if* a computational process were to be sensitive to semantic properties, then it would have to violate either a condition that he called 'Syntactic Rules' or a condition that he called 'Freedom', *and* that such a semantically sensitive computation would have to have an "indigenous" semantics, not an "inherited" semantics. He defined these terms as follows:

> SYNTACTIC RULES: Computation is manipulation of syntactic entities according to mechanical rules. We can specify those rules in syntactic terms, without mentioning semantic properties such as meaning, truth, or reference.
>
> (Rescorla, 2012a, §3, p. 707)

> FREEDOM: We can offer a complete syntactic description of the system's states and the mechanical rules governing transitions between states (including any interaction with sensory and motor transducers), while leaving semantic interpretation unconstrained. More precisely, it is metaphysically possible for the system to satisfy our syntactic description while instantiating arbitrarily different semantic properties. (Rescorla, 2012a, §3, p. 708)

> *Inherited meanings* arise when the system's semantic properties are assigned to it by other systems, through either explicit stipulation or tacit convention. Nothing about the system itself helps generate its own semantics. For instance, words in a book have inherited meanings. *Indigenous meanings* arise when a system helps generate its own semantics. Indigenous meanings do not arise merely from external assignment. They arise partly from the system's own activity, perhaps with ample help from other factors, such as causal, evolutionary, or design history. Virtually all commentators agree that the mind has indigenous meanings.
> (Rescorla, 2012a, §3, pp. 707–708)

Rescorla's "indigenous" semantics seems clearly akin to Piccinini's "internal" semantics and to what we are calling "syntactic" semantics.

Further Reading and a Question for the Reader:
However, as we saw in §17.8.1, three years later he seems to have changed his mind! Rescorla 2015b offers the following "Gap Argument":

> A Turing machine manipulates linguistic items, but we sometimes want to study computation over non-linguistic domain X. So there is a gap between the domain of items manipulated by the Turing machine and our desired domain of computation X. To bridge the gap, we must interpret linguistic items manipulated by the Turing machine as denoting items drawn from X. A Turing machine computes over X only if linguistic items manipulated by the Turing machine represent elements of X. Thus, any complete theory of computation must cite representational relations between linguistic items and non-linguistic items.

He then says, "Given the Gap Argument, we can study Turing computation over a non-linguistic domain only if we furnish a semantics for strings" (Rescorla, 2015b, §3). Buechner 2011, pp. 358–362, makes a similar argument.

Is Rescorla's gap the same as Smith's gap?

17.8.2.3 Syntactic Semantics and Procedural Abstraction

One way to provide an internal, indigenous, or syntactic semantics is to use "procedural abstraction"—named subroutines that accomplish subtasks of the overall algorithm (§7.5.3): Identify subtasks (collections of statements in a program that "work together"), package them up, and name the package, thus giving an identity to the subtasks.

For example, the following Logo program draws a unit square by moving forward 1 unit, then turning 90 degrees right, and doing that 4 times:

```
repeat 4 [forward 1 right 90]
```

But Logo won't "know" what it means to draw a square unless we tell it this:

```
to square
repeat 4 [forward 1 right 90]
end
```

Note that this Logo instruction has the form: To *G*, do *A*! The "To *G*" has been internalized. (We'll come back to this idea in §17.8.2.4.)

Another example is the sequence of instructions "turnleft; turnleft; turnleft", in *Karel the Robot* (Pattis et al., 1995), which can be packaged up and named "turnright":

```
DEFINE-NEW-INSTRUCTION turnright AS
BEGIN
turnleft;turnleft;turnleft
END
```

Notice here that Karel still can't "turn *right*" in an external sense (that is, 90° clockwise); it can only turn left three times (that is, 270° counterclockwise).

There is an important caveat: The Logo and Karel *programs* still have no "understanding" in the way that *we* do of what a square is or what it means to turn right. Merely *naming* a subroutine does not automatically endow it with the (external) meaning of that name (McDermott, 1980). The programs are now capable only of associating those newly defined symbols ('square', 'turnright') with certain procedures. The symbols' meanings for *us* are their *external* semantics; their meanings for the Logo or Karel *programs* are their internal, indigenous, syntactic semantics due to their internal relationships with the bodies of those programs. If the name is associated with objects that are *external* to the program, then we have external (or wide, or inherited, or extra-system) semantics. If it is associated with objects *internal* to the program, then we have internal (or narrow, or syntactic, or indigenous, or intra-system) semantics. *Identifying* subroutines is syntactic; *naming* them leads to semantics: If the name is externally meaningful to a *user*, because the user can associate the name with other external concepts, then we have semantics in the ordinary sense (subject to McDermott's caveat). If it is internally meaningful to the *computer*, in the sense that the computer can associate the name with other internal names, then we have internal, syntactic semantics.

The debate over whether computation concerns the internal, syntactic manipulation of symbols or the external, semantic interpretation of them is at the heart of both Rescorla's gap (see the Further Reading in §17.8.2.2) and Smith's gap (from §17.3.2.3) . This is made explicitly clear in the following passages from Michael Mahoney's history of computing:

> Recall what computers do. They take sequences, or strings, of symbols and transform them into other strings. ...
>
> The transformations themselves are strictly syntactical, or structural. They may have a semantics in the sense that certain symbols or sequences of symbols are transformed in certain ways, *but even that semantics is syntactically defined*. Any meaning the symbols may have is acquired and expressed at the interface between a computation and the world in which it is embedded. The symbols and

> their combinations express representations of the world, *which have meaning to us, not to the computer.* ... What we can make computers do depends on how we can represent in the symbols of computation portions of the world of interest to us and how we can translate the resulting transformed representation into desired actions. ...
>
> So putting a portion of the world into the computer means designing an operative representation of it that captures what we take to be its essential features. That has proved ... no easy task; on the contrary it has proved difficult, frustrating, and in some cases disastrous. (Mahoney, 2011, p. 67, my italics)

The computer's internal (syntactic) semantics—its "Do *A*" (including *A*'s modules or compositional structure—is syntactic and non-teleological. Its external semantics, "which have meaning to us"—its "To *G*"—is teleological, but depends on *our* ability to represent *our view of* the world *to it*. As Rescorla (2007, p. 265) observed, we need a computable theory of the semantic interpretation function, but, as Smith observes, we don't (can't?) have one, for reasons akin to the Computability Thesis problem: Equivalence between something formal (for example, a Turing-machine or a formal model) and something non-formal (for example, an algorithm or a portion of the real world) cannot be *formally* proved.

Further Reading:
For more on syntactic semantics, see Rapaport 1986f, 1988a, 1995, 2000b, 2003, 2006a, 2012b, 2017b, 2018b; and Kay 2001.

Rescorla's "indigenous semantics" (Rescorla, 2012a, 2014a) emphasizes *causal* relations, whereas syntactic semantics emphasizes the importance of *conceptual-role* semantics (Rapaport, 2002).

Egan's "structural properties" (1995, p. 181) and Bickle's description of "causal-mechanistic explanations" in neuroscience (2015, especially §5) may also be akin to syntactic or indigenous semantics. See also our earlier discussion of "intrinsic" properties, in §9.5.4.

17.8.2.4 Internalization

Syntactic semantics can arise in another way: *External* semantic relations between the elements of *two* domains (a "syntactic" domain described syntactically and a "semantic" domain described ontologically (that is, syntactically!—see §14.3.2.3) can be turned into *internal* syntactic relations ("syntactic semantics") by *internalizing* the semantic domain into the syntactic domain. After all, if you take the union of the syntactic and semantic domains, then all formerly external semantic relations are now internal syntactic ones (internal to the union).

One way that this happens for cognitive agents like us is by sensory perception, which is a form of input encoding. For animal brains, perception interprets signals from the external world into the biological neural network of the brain. For a computer that accepts input from the external world, the interpretation of external or user input as internal switch settings (or inscriptions on a Turing-machine tape) constitutes a form of perception—a way of internalizing external information. Both are forms of what I am

calling "internalization". As a result, the interpretation becomes part of the computer's or the brain's intra-system, syntactic semantics (Rapaport, 2012b).

Stuart C. Shapiro advocates internalization in the following form, which generalizes the Logo and Karel techniques:[15]

Shapiro's Internalization Tactic

Algorithms *do* take the teleological form, "To *G*, do *A*",
but *G* must include *everything* that is relevant:

- To make hollandaise sauce *on Earth*, do *A*.
- To find the GCD of 2 integers *in base-10*, do *B*.
- To play chess, do *C*, *where C's variables range over chess pieces and a chess board*.
- To simulate a war battle, do *D*, *where D's variables range over soldiers and a battlefield*.

One place to locate these teleological clauses is in the preconditions and postconditions of the program. They can then be used in the formal verification of the program, which proceeds by proving that, *if the preconditions are satisfied*, then the program will accomplish its goal *as articulated in the postconditions*. This builds the external world (and any attendant external semantics) *into* the algorithm: "There is no easy way to ensure a blueprint stays with a building, but a specification can and should be embedded as a comment within the code it is specifying" (Lamport, 2015, p. 41). The separability of blueprint from building is akin to the separability of *G* from *A*; embedding a specification into code as (at least) a comment is to internalize it as a pre- or postcondition. More importantly, such pre- and postconditions need not be "mere" comments; they can be internalized as "assertible" statements in a program, thus becoming part of a program's (self-)verification process (Lamport, 2011).

The logician Nicolas Goodman (1987, p. 482) made a similar observation, noting that the Computability Thesis relates "the informal mathematical notion of algorithm ... [to] the formal set-theoretic notion of a Turing machine program." This suggests that we can view "To *G*, do *A*" as an "informal algorithm" and "Do *A*" as a "formal Turing-machine program." Then the former is intentional in nature, because "An algorithm, in the informal mathematical sense, is a specific procedure *for solving a particular kind of mathematical problem*" (Goodman, 1987, p. 482, my italics; see also p. 483). But the latter is not, because "the Turing machine program does not tell you what the program is for. ... [Only the] documentation contains the intensional content which is missing from the bare machine code ... and brings the program closer to the algorithm which it is intended to implement" (p. 483). However, Goodman did not believe that the Computability Thesis is

> an analysis of the informal concept of algorithm. It at most provides a necessary condition for the existence of an algorithm. That is, a problem which no Turing machine can solve cannot be solved algorithmically. However, a Turing machine

[15] Personal communication. B.C. Smith (1985, p. 24) makes a similar point: "as well as modelling the artifact itself, you have to model the relevant part of the world in which it will be embedded."

> program without additional explanation is not an algorithm, and an algorithm is not as it stands a Turing machine program. ... My contention is rather that not all of the content of our informal intensional talk about algorithms is captured by extensional talk about Turing machine programs. (Goodman, 1987, p. 487)

As I suggested in §17.7.1, we can avoid having Cleland's hollandaise-sauce recipe output a messy goop by limiting its execution to one location (Earth, say) without guaranteeing that it will work elsewhere (on the Moon, say). This is no different from a partial mathematical function that is silent about what to do with input from outside its domain, or from an algorithm for adding two integers that specifies no particular behavior for non-numerical input. ("Crashing" is a well-defined behavior if the program is silent about illegal input. More "well-behaved" behavior requires some kind of error handling.) A second way is to use the "Denver cake mix" strategy: I have been told that packages of cake mix that are sold in mile-high Denver come with alternative directions. The recipe or algorithm should be expressed conditionally: If location = Earth, then do A; if location = Moon, then do B (where B might be the output of an error message).

Digression:
There is a similarity between (a) internalizing external (or inherited) semantics into internal (or syntactic) semantics and (b) the Deduction Theorem in logic, which can be thought of as saying that a premise of an argument can be "internalized" as the antecedent of an argument's conditionalized conclusion:

$$A \vdash C \;\Leftrightarrow\; \vdash (A \rightarrow C).$$

That is, proving that C follows from A is the same as proving that "If A, then C" is a theorem. In the terminology we used in Chapter 16, C is provable *relative to* premise A iff '$(A \rightarrow C)$' is "absolutely" provable.

17.9 Content and Computation

17.9.1 Introduction

The quotation from Rescorla (2007) at the beginning of §17.8.1 focuses the issues very clearly. Are we really interested in syntactic computation—computation with *symbols*, such as numerals? Or are we interested in semantic computation—computation with *things* that the symbols represent, such as numbers?

David Hilbert, whose investigations into the foundations of mathematics prompted much of the early work in the theory of computation (as we surveyed in §6.6), was a mathematical "formalist". As such, he was interested in the former, for, after all, we humans can only do the latter via the former (recall the Questions for the Reader in §17.8.1, above). Is that a limitation? Perhaps, but it also gives us a freedom, because symbols (including numerals) can represent anything, not just numbers, and so computation can be about anything.

Rescorla (2007, pp. 272–274), on the other hand, seems to favor numbers:

> One argument runs as follows: humans and computers directly manipulate symbols, not numbers; thus, what humans and computers *really* compute are string-theoretic functions, not number-theoretic functions. ... The argument is fallacious. ... At best, the premise establishes that our computations of number-theoretic functions are *mediated* by our computations of string-theoretic functions. It does not follow that all we *really* ... compute are string-theoretic functions. To conclude this would be analogous to the inference sometimes drawn by the British empiricists that, since our ideas mediate our perception of the external world, all we really perceive are our ideas.

Whether the British empiricists were on the right track or not (Rapaport, 2012b, §3.1), Rescorla's broader point is that, to the extent that we want to compute over numbers, not numerals, we need a (computable) interpretation function from the numerals to the numbers. But, as we have seen, there can be several different ones. It is those external semantic interpretation functions that can take the algorithm (narrowly construed) in different directions. Let's explore this a bit further.

Further Reading and Philosophical Digression:
On formalism in the philosophy of mathematics in general, and Hilbert's views in particular, see Curry 1951; Weir 2015; Zach 2019. And consider this passage from Quine 1987, p. 67:

> A computer requires blow-by-blow instruction, strictly in terms of what to do to strings of marks or digits, and eked out by no arm-waving or appeals to common sense and imagination; and such, precisely, is formalism.

The narrow view of Turing machines as computing with strokes (symbols, numerals instead of numbers) vs. the wide view of Turing machines as being embedded in the external world and being context-dependent is reminiscent of (if not identical to) the issues in psychology and the philosophy of language concerning individualism (or methodological solipsism) vs. "externalism" (see §11.4.3.4.2): Are psychology and meaning (and cognition, more broadly) narrow, needing to deal only with what goes on in one's head? Or are they wide, needing to take the external, embedding world into account? Analogously, we can ask whether *computation* is narrow, needing to deal only with what goes on from a Turing machine's point of view (so to speak). Or is it wide, needing to take the external, embedding world into account? (We'll return to this in §19.6.3.2.)

The philosopher and logician Gottlob Frege (1892) argued that linguistic expressions had two different kinds of "meanings": a *sense* and a *referent*. The sense of a linguistic expression was a "way" in which it picked out a referent in the world. Does a sense *determine* a referent? Hilary Putnam (1975) argued that (very roughly) the sense of the word 'water' might pick out H_2O on Earth but XYZ on "Twin Earth". Putnam also argued that the referent was more significant than the sense.

Arguably, Fregean senses don't *determine* reference (because of the possibility of multiple realization, or "slippage" between sense and reference (Putnam, 1975)). But senses do tell you *how to find* a referent: Perhaps they are *algorithms*. In that sense, the referent might be like the purpose or task of the algorithm. But there is always the possibility of "slippage": An algorithm can be intended for one purpose (say, playing chess), but be used for another (say, a war battle) by changing the interpretation of the inputs and outputs. Besides Putnam's essay, see Burge 1979, 1986; Fodor 1980; Egan 1991, 1995.

17.9.2 Symbols: Marks vs. Meanings

In §14.3.2.1, we observed that some writers use 'symbol' to mean an uninterpreted, purely syntactic "mark" *together with* its (external) semantic interpretation or meaning. A symbol is at least a mark; its interpretation is another matter. Symbols are, perhaps, best thought of as ordered pairs of (syntactic) marks (identified by their shape) and (semantic) interpretations. Sometimes, the "meaning" of a symbol is called its "content". So, is computation (only?) about marks? Or is it (only? also?) about content?

The term 'content' sounds as if it refers to something contained within something else—something *internal*—but often it is used to indicate the *external* meaning or reference of a term. But if we think of the variables of a computer program as "boxes" that can contain the values of the variables, then we can combine both metaphors for "content": The content of a variable can be thought of as an "external" entity that is stored "inside" the "box".

Several writers say that content *is* necessary; something is not a computation unless it is *about* something: There is "no computation without representation" (Fodor, 1975, p. 34). *Is* a goal, or content, or interpretation a necessary part of a computation?

From a syntactic (internal) point of view, a Turing machine that outputs sequences of strokes does just that: It outputs sequences of strokes, that is, "marks". From a semantic (external) point of view, those strokes are symbols, that is, marks plus content. Whether the marks should, or can, be interpreted as the integer n or the integer $2n$ is a *separate* matter. This is, of course, what underlies the notion of "types" in programming languages. The question we are now considering is whether the type of a programming-language variable is a syntactic issue or a semantic one; note that it might be a case of "*syntactic* semantics".

This puzzle is not unique to computation. The mathematician Edward Frenkel (2013) considers an equation like $y^2+y=x^3+x^2$, and asks

> But what kind of numbers do we want x and y to be? There are several choices: one possibility is to say that x and y are natural numbers or integers. Another possibility is to say rational numbers. We can also look for solutions x, y that are real numbers, or even complex numbers ... (p. 83).

He continues:

> [W]hen we talk about solutions of such an equation, it is important to specify to what numerical system they belong. There are many choices, and different choices give rise to different mathematical theories. (p. 99)

Here's a simpler example: What are the solutions to the equation $x^2=2$? In the rational numbers, there is no solution; in the positive real numbers, there is one solution; in the (positive and negative) real numbers, there are two solutions. Similarly, $x^2=-1$ has no solution in the real numbers, but two solutions in the complex numbers. Deciding which "wider" number system the equation should be "embedded" in gives different "interpretations" of it.

Syntactically, we can say that *the* solution to $x^2=c$ is $\sqrt{c}$. Whether (or not) we assign a rational, real, or imaginary *number* to the *symbol* '$\sqrt{c}$' is a separate matter. Similarly, the ratio of the circumference to the diameter of a circle is π; whether we

understand the symbol 'π' as $\frac{22}{7}$, 3.14, 3.1415926535, or something else is a separate matter. We can, in fact, compute more "accurately" with the syntactic mark 'π' than we can with any of those finite, numeral interpretations.

Consider, again, Marr's computational theory of vision, part of which takes the form of an algorithm that "computes the Laplacean convolved with a Gaussian" (Egan, 2014, p. 120). For the present point, it is unimportant to know what Laplaceans, Gaussians, and convolution are; what matters is that they are purely mathematical operations, having nothing necessarily to do with vision. Mark Sprevak (2010, p. 263) argues that this "mathematical computation theory does not, by itself, have the resources to explain" vision; it needs to be augmented by a link "to the nuts and bolts of physical reality". Frances Egan takes an opposing view: "representational content is to be understood as a *gloss* on the computational characterization of a cognitive process" (Egan, 2010, p. 253). Once again, we have a difference between "To *G*, do *A*" and "Do *A*". Here, *A* is the Laplacean convolved with a Gaussian, and *G* is the "gloss" about its role in vision—its "content".

On Egan's side, one might say that the mathematical theory *does* have the resources to explain vision (that's one of the points of Wigner 1960). It may still be a puzzle *how* or *why* it does (recall Marr's "why", quoted in §17.7.2), but there's no question *that* it does. (This is reminiscent of the problem of quantum mechanics as an "instrumentalist" scientific theory: We know that quantum mechanics has the resources to explain physics, but it is still a puzzle how or why it does. Recall our discussion of this in §4.5, and see Becker 2018.)

Paul Humphreys suggests a view of computational models that can account for this, as well as the war-chess example: "one of the characteristic features of mathematical [including computational] models is that *the same model ... can occur in, and be successfully employed in, fields with quite different subject matters*" (Humphreys, 2002, p. S2; my interpolated brackets and italics). He goes on to say, "Let the ... computer solve one and you automatically have a solution to the other" (p. S5), as illustrated by de Bruijn's lattice story.

Sprevak offers a counterargument to the focus on a Turing machine's strokes rather than their meanings:

> ... one cannot make sense of I/O equivalence without requiring that computation involves representational content. ...
>
> Consider two computational systems that perform the same numerical calculation. Suppose that one system takes ink-marks shaped like Roman numerals (I, II, III, IV, ...) as input and yields ink-marks shaped like Roman numerals as output. Suppose that the other system takes ink-marks shaped like Arabic numerals (1, 2, 3, 4, ...) as input and yields ink-marks shaped like Arabic numerals as output. Suppose that the two systems compute the same function, say, the addition function. What could their I/O computational equivalence consist in? Again, there may be no physical or functional identity between their respective inputs and outputs. The only way in which the their inputs and outputs are relevantly similar seems to be that they represent the same thing. (Sprevak, 2010, §3.2, p. 268, col. 1)

That is, that they compute the same arithmetic function cannot be explained without a semantic interpretation. Note that there is a difference between *what* a system is doing

and whether *two* systems are doing the *same* thing: Each addition algorithm (Roman and Arabic) is "doing its own thing". They are only doing the "same" thing in the sense that the two idiosyncratic things that they are each doing are *equivalent*. This kind of *sameness* (or equivalence) depends on the semantics.

Sprevak goes on to say:

> Two physical processes that are intrinsic physical duplicates may have different representational contents associated with them, and hence different computational identities. One physical process may calculate chess moves, while a physical duplicate of that process calculates stock market predictions. We seem inclined to say that, in a sense, the two processes compute different functions, yet in another sense they are I/O equivalent. Appeal to representational content can accommodate both judgements. (Sprevak, 2010, §3.2, p. 268, col. 2)

But this is a different case: identical algorithm but different task.

The previous case is different algorithm but same (input-output–equivalent) task. But *syntactically* they are *not* doing the *same* thing; rather, they are doing things that are only *semantically equivalent*. That equivalence can be discovered, explained, and understood only via an external semantic interpretation.

Nevertheless, depending on how the two algorithms are structured, it might be possible to find subroutines that match up. In that case, the two algorithms would be doing the same (identical) thing *at a suitably high level of organization*, even if the low-level implementations of those subroutines are completely different. That would be a syntactic semantic "interpretation". For example, a Karel the Robot who turns right by turning left three times is turning right (better: is turning in the "right" direction) just as much as a *Karl* the Robot who turns right by turning left six times, or a *Kal* the Robot for whom turning right is primitive (and who might have to turn *right* three times to turn *left*).

Let's look into the Marr example in more detail: Egan says, "*As it happens*, … ["the device [that] computes the Laplacean convolved with the Gaussian"] takes as input light intensity values at points in the retinal image, and calculates the rate of change of intensity over the image" (Egan, 2010, p. 255; my italics and interpolated brackets). But, considered solely as a

> computational device, it does not matter that input values represent *light intensities* and output values the rate of change of *light intensity*. The computational theory characterizes the visual filter as a member of a well understood class of mathematical devices that have nothing essentially to do with the transduction of light. (Egan, 2010, p. 255; original italics)

Compare this to the chess-war example: To paraphrase Egan, the theoretically important characterization from a computational point of view is a mathematical description: The device computes some mathematical function that, *as it happens*, can be interpreted as a chess match or else as a war battle. But, considered solely as a computational device, it does not matter that input values represent (say) chess moves or battle positions—the computational theory characterizes the device as a member of a well understood class of mathematical devices that have nothing essentially to do with chess or war:

> A crucial feature of ... [the characterization that focuses solely on the mathematical function being computed and not on the purpose or external environment] is that it is 'environment neutral': the task is characterized in terms that prescind from the environment in which the mechanism is normally deployed. The mechanism described by Marr would compute the Laplacean of a Gaussian even if it were to appear (*per mirabile*) in an environment where light behaves very differently than it does on earth, or as part of an envatted brain. ...
> (Egan, 2014, p. 122, my interpolated brackets)

Egan says that the visual filter "would compute the same mathematical function in any environment, *but only in some environments would its doing so enable the organism to see*" (Egan, 2010, p. 256; my italics). Similarly, Cleland's recipe would compute the same (culinary?) function in any environment, but only on Earth (and not on the Moon) would its doing so result in hollandaise sauce.

Given a computer program, how do you know what its purpose is? Of course, it might be obvious from its name, its documentation, or even its behavior when executed. But suppose you come across a very large program written in an unfamiliar programming language with unintuitive variable and subroutine names and no documentation. Suppose that, after considerable study of it, you are able to describe it and its behavior syntactically. You might also be able to develop a hypothesis about a purpose for it, by providing an interpretation for it (for example, that it is a chess program). And you, or someone else, might also be able to provide a different, but equally good interpretation for it (for example, that it is a war simulator). This is not unlike the situation with the brain, a very large neural network with no documentation.

Recall the MYSTERY Scheme program from §14.4.3:

```
(define (MYSTERY a b)
  (if (= b 0)
    a
    (MYSTERY b (remainder a b))))
```

If I found the MYSTERY program in the desert and was able to describe it syntactically as outputting pretty patterns of numbers (whether base-10 or base-13 is irrelevant), I could stop there. Or, if I wrote another program that took MYSTERY's pretty patterns and translated them into base-10, I could use it as a GCD computer. Similarly, if I found a computer in the desert that output pretty patterns of a certain sort, I might write another program that translated its output into a chess game. And *you* might write another program that translated those very same pretty patterns into a war battle.

Given a *problem* to be solved, or a *task* to be accomplished, a computer scientist asks whether it is computable. If it is, then we can write a computer program to solve the problem or accomplish the task. Kleene's (1995) informal characterization of "algorithm" begins as follows:

> [A] method for answering any one of a given infinite class of questions ... is given by a set of rules or instructions, describing a procedure that works as follows. *After* the procedure has been described, [then] if we select *any* question from the class, the procedure will then tell us how to perform successive steps, ...

Note that the procedure has a purpose: "answering any one of a given infinite class of questions". And the procedure depends on that class: Given a class of questions, there is a procedure such that, given "*any* question from the class," the procedure "enable[s] us to recognize that now we have the answer before us and to read it off" (Kleene, 1995, p. 18).

Given that *program*, the programmer or a user knows what its original or intended purpose was. But we, or someone else, might be able to interpret it differently and use it for a different purpose (playing chess instead of simulating a war battle).

And we might also be able to re-implement it in a different medium. Marr wanted to explain certain aspects of human vision. He found an algorithm (computing the Laplacean convolved with a Gaussian, say) that helps to accomplish that. When that algorithm is implemented in the *human* visual system, it enables human visual perception. If it were implemented in a *computer*, it might enable *robot* visual perception. If it were implemented elsewhere than on Earth, it might do nothing visually (for humans *or* robots), but it would still compute a Laplacean convolved with a Gaussian. What task it accomplishes (in the intentional, external, teleological sense) depends on where that algorithm is "plugged in" to its environment. The same holds for all of the examples from §17.5. (More generally, we might be able to take a cognitive computational program for *human* vision, natural-language understanding, or reasoning and *re-implement* it in a *robot*. We'll explore this option in Chapter 19.)

17.9.3 Shagrir's "Master Argument"

Oron Shagrir (2018b, §3) offers the following "master argument" for "the semantic view of computation":

> 1. A physical system might simultaneously implement several different automata S_1, S_2, S_3 ....
> 2. The contents of the system's states determine (at least partly) which of the implemented automata, S_i, is relevant for computational individuation.
>
> Conclusion: The computational individuation of a physical system is essentially affected by content.

Let's take a close look at this, beginning with the conclusion. Note that what this argument attempts to show is not that *abstract* computation is semantic, but that *physical* computation is. So, it might be the case *both* that abstract computation is *not* semantic (or wide, etc.) *and* that physical computation *is*.

We have already seen several examples that seem to support the first premise: Fodor's chess-war computers, Rescorla's GCD computers, and Shagrir's and-vs.-or computers. But a closer look suggests a puzzle: Which is the "physical system" that is the implementation, and which is the "automaton" that gets implemented?

In the chess-war case, there are *two* physical systems, each of which implements the *same* (abstract) computation. But in Shagrir's example, there is an (abstract?) AND-gate that is implemented by (one interpretation of) a certain physical system and an (abstract?) OR-gate that is implemented by (a different interpretation of) the *same* physical system. In this case, there is *one* physical system that implements *different* computations.

Yet another way to look at Shagrir's example is this: There is an underlying automaton that outputs certain symbols in response to certain inputs of those symbols, and there are two different (physical?) interpretations of those symbols such that under one interpretation we have a (physical?) AND-gate and under the other we have a (physical?) OR-gate. We'll return to this puzzle in a moment, but let's first look at the second premise.

The second premise talks about "contents" and "individuation". What are these? Let's begin with individuation. Consider an abstraction and two implementations of it. For concreteness, you might think of the abstract species *Homo sapiens* and two concrete implementations of it: Alan Turing and Alonzo Church. We can ask two questions about these three things: First, what makes the concrete *individuals* Turing and Church different from the abstract *species*? Second, what makes Turing different from Church? (The first question is "vertical", asking about the relation between a "higher-level" abstraction and "lower-level" implementations of it. The second question is "horizontal", asking about the relation between two objects at the "same level".)

Unfortunately, in philosophy, the term 'individuation' has been used for both of these questions. The questions should be kept distinct (and Hector-Neri Castañeda (1975) has suggested calling the first one 'individuation' and the second one 'differentiation'). So, as you evaluate Shagrir's argument, you need to decide which question he has in mind.

As for "content", recall from §17.9.2 that we can think of a mark (or a variable) as a box, and its meaning (or value) as the "content" of the box. So, we might be able to understand the second premise as follows: There is an abstract computation—an automaton—that can be characterized in terms of operations on its "boxes", that is, its variables (recall from §9.6 Thomason's and Lamport's views of the nature of computation as a sequence of states that are assignments of values to variables). That abstract computation can be implemented by different physical systems depending on the contents of the "boxes". Those contents help us "individuate" the physical system. Depending on how we interpret 'individuation', that means that *either* those contents tell us what makes one computer a chess computer and another a war simulator (to use Fodor's example) even though both computers implement the same automaton, *or* those contents tell us what makes the chess computer different from the war simulator. Arguably, they tell us both of these things. (So perhaps the interpretation of 'individuation' doesn't matter in this case.)

So, if we have two different physical systems that implement the same automaton, then what makes them different is their semantic content. But that doesn't seem to say anything about the computation itself, which still appears to be able to be understood "narrowly". So, it is *physical* computation that might be semantic and wide, while it is *abstract* computation that might be syntactic and narrow.

17.10 Summary

We can distinguish between the question of which computation a given Turing machine is executing and the question of what goal that computation is trying to accomplish. Both questions are important, and they can have very different answers. Two computers might implement the same Turing machine, but be designed to accomplish different goals. And, of course, two computers might accomplish the same goal via different algorithms.

And we can distinguish between two kinds of semantics: external (or wide, or extrinsic, or inherited) and internal (or narrow, or intrinsic, or syntactic, or indigenous). Both kinds exist, have interesting properties and play different, albeit complementary, roles.

Algorithms narrowly construed (minus the teleological preface) are what is studied in the mathematical theory of computation. To decide whether a task is computable, we need to find an algorithm that can accomplish it. Thus, we have two separate things: an algorithm (narrowly construed, if you prefer) and a task. Some algorithms can accomplish more than one task (depending on how their inputs and outputs are interpreted by external semantics). Some algorithms may fail, not because of a buggy, narrow algorithm, but because of a problem at the real-world interface. That interface is the (algorithmic) coding of the algorithm's inputs and outputs, typically through a *sequence* of transducers at the real-world end (what B.C. Smith (1987) called a "correspondence continuum; see §14.3.1). Physical signals from the external world must be transduced (encoded) into the computer's switch-settings (the physical analogues of a Turing machine's '0's and '1's), and the output switch-settings have to be transduced (decoded) into such real-world things as displays on a screen or physical movements by a robot.

But real-world tasks are complex. Models abstract from this complexity, so they can never match the rich complexity of the world. Computers see the world through models of these models (and so do people!). Reasoning on the basis of partial information cannot be proved correct (and simulation only tests the computer-model relation, not the model-world relation). So, empirical *reliability* must supplement program verification. Therefore, we must embed the computer in the real world.

At the real-world end of the correspondence continuum, we run into Smith's gap. From the narrow algorithm's point of view, so to speak, it might be able to asymptotically approach the real world, in Zeno-like fashion, without closing the gap. But, just as someone trying to cross a room by only going half the remaining distance at each step *will* eventually cross the room (though not because of doing it that way), so the narrow algorithm implemented in a physical computer *will* do something in the real world. Whether what it accomplishes was what its programmer intended is another matter. (In the real world, there are no "partial functions"! This was one of Peter Kugel's points about trial-and-error machines, as we saw in §11.4.5.2.)

One way to make teleological algorithms more likely to be successful is by Shapiro's strategy: Internalizing the external, teleological aspects into the pre- and post-conditions of the (narrow) algorithm, thereby turning the external semantic interpretation of the algorithm into an internal, syntactic semantics.

What Smith shows is that the external semantics for an algorithm is never a relation

directly with the real world, but only to a *model* of the real world. That is, the real-world semantics has been internalized. But that internalization is necessarily partial and incomplete.

There are algorithms *simpliciter* ("Do *A*"), and there are algorithms *for accomplishing a particular task* ("To *G*, do *A*"). Alternatively, we could say that *all* algorithms accomplish a particular task, but some tasks are more "interesting" than others. The algorithms whose tasks are not currently of interest may ultimately *become* interesting when an application is found for them, as was the case with non-Euclidean geometry. Put otherwise, the algorithms that do not have an obvious goal may ultimately be used to accomplish one:

> [D]oes ... any algorithm ... have to do something interesting? No. The algorithms we tend to talk about almost always do something interesting—that's why they attract our attention. But a procedure doesn't fail to be an algorithm just because it is of no conceivable use or value to anyone. ... *Algorithms don't have to have points or purposes.* ... Some algorithms do things so boringly irregular and pointless that there is no succinct way of saying what they are *for*. They just do what they do, and they do it every time. (Dennett, 1995, p. 57, my italics)

Further Reading:
On "succinct ways of saying what an algorithm is for", see Chaitin 2002, 2006b, cited earlier in Chapter 7.

* * * * *

A few paragraphs ago, I said that, despite Smith's gap, a narrow algorithm implemented in the real world *will* do something, whether what it was intended to do or not. What about an automated system designed to decide quickly (and in the absence of complete information) how to respond to an emergency? Would it make you feel uneasy? But so should a human who has to make that same decision. And they should both make you uneasy for the same reason: They have to reason and act on the basis of partial (incomplete) information. This will be our topic in the next chapter.

17.11 Questions for the Reader

1. Recall the opening epigraph. How *does* a program interact with the world? Is it via the *process*, which is a physical entity (or event?) *in* the world? If so, how does the program interact with the *process*? Is it via the compiler? Or is that just a first step, translating the program into the machine language that the machine understands. Is it via the loader, which is what transforms the machine-language program into the memory, setting the switches?

2. The artist Charles E. Burchfield said that

   > An artist must paint not what he sees in nature, but what is there. To do so he must invent symbols, which, if properly used, make his work seem even more real than what is in front of him.
   > —https://www.burchfieldpenney.org/collection/charles-e-burchfield/biography/

   If we change 'artist' to 'programmer' and 'paint' to 'program', this becomes:

   > Programmers must program not what they see in nature, but what is there. To do so they must invent symbols, which, if properly used, make their work seem even more real than what is in front of them.

   Is the artist's task different from the scientist's or the programmer's? Can programs (or paintings, or scientific theories) end up seeming more "real" to us than the things that they are models of? Is it easier to understand the behavior of the process of a program that models a hurricane (for example) than to understand the real hurricane itself?[16]

[16] Thanks to Albert Goldfain, personal communication, 3 April 2007, for this question.

Part V

Computer Ethics and Artificial Intelligence

The topics of Part V—computer ethics and the philosophy of AI—are both large and long-standing disciplines in their own right. Ethics is a branch of philosophy, and AI is a branch of CS, so both computer ethics and the philosophy of AI should be branches of the philosophy of computer science. In §3.15.2, we saw that the question of what can be computed includes the question of whether "intelligence"—more accurately, *cognition*—is computable. And we saw that, besides the questions of what *can* be computed and what can be computed *efficiently, practically,* and *physically* (and how), CS itself includes the ethical question of what *should* be computed (and how).

In Chapters 18 and 20, we will focus on two topics in computer ethics that I think are central to the philosophy of computer science but, until recently, have not been the focus of most discussions of computer ethics: Should we trust decisions that computers make? And should we build "artificial intelligences"? But before we can try to answer that last question, Chapter 19 on the philosophy of AI will focus on whether we *can* build them.

Further Reading:
For good introductions to computer ethics in general, see Moor 1985; Johnson and Snapper 1985 (for a review, see Rapaport 1986e); Johnson 2001a; Anderson and Anderson 2006. AAAI's "AI Topics" website on "Ethical & Social Issues: Implications of AI for Society", is an excellent site, with many links: http://aitopics.net/Ethics. See also The Research Center on Computing & Society, http://southernct.edu/organizations/rccs/

Chapter 18

Computer Ethics I:
Should Computers Make Decisions for Us?

Version of 6 September 2019; DRAFT © 2004–2019 by William J. Rapaport

> In 2011, [John] Rogers ... announced the invention of ... an integrated silicon circuit with the mechanical properties of skin. ... The artificial pericardium [made with Rogers's invention] will detect and treat a heart attack before any symptoms appear. "It's twenty years or more out there," Rogers said. "But we can see a pathway—it's not science fiction." Bit by bit, our cells and tissues are becoming just another brand of hardware to be upgraded and refined. I asked him whether eventually electronic parts would let us live forever, and whether he thought this would come as a relief, offer evidence that we'd lost our souls and become robots, or both. *"That's a good thing to think about, and people should think about it,"* he said *"But I'm just an engineer, basically."*
> —Kim Tingley (2013, p. 80; my interpolated brackets and italics)

> *Machines are more than ever controlled by software, not humans. Occasionally it goes fatally wrong.* ... [I]ncreasing the complexity of systems makes checking them more difficult. Hardware, from chips to special sensors, can be difficult to test. And it can be difficult for humans to understand how some A.I. algorithms make decisions. —Jamie Condliffe (2019, my italics)

18.1 Readings:

1. Required:

   - Moor, James H. (1979), "Are There Decisions Computers Should Never Make?", *Nature and System* 1: 217–229, http://www.researchgate.net/publication/242529825_Are_there_decisions_computers_should_never_make

2. Recommended:

   (a) Friedman, Batya; & Kahn, Peter H., Jr. (1992), "People Are Responsible, Computers Are Not", excerpt from their "Human Agency and Responsible Computing: Implications for Computer System Design", *Journal of Systems and Software* (1992): 7–14; excerpt reprinted in M. David Ermann, Mary B. Williams, & Michele S. Shauf (eds.) (1997), *Computers, Ethics, and Society, Second Edition* (New York: Oxford University Press): 303–314; excerpts online at https://books.google.com/books?id=vgwkDwAAQBAJ&printsec=frontcover#v=onepage&q=kahn&f=false

   - Can be read as raising an objection to Moor 1979, especially the section "Delegating Decision Making to Computational Systems" (pp. 306–307 in the Ermann et al. 1997 reprint; online at http://tinyurl.com/y5cc2quq).

   (b) Johnson, George (2002), "To Err Is Human", *New York Times* (14 July), http://www.nytimes.com/2002/07/14/weekinreview/deus-ex-machina-to-err-is-human.html

   - An op-ed piece that provides an interesting, real-life case study of Moor's problem and a possible counterexample to Friedman and Kahn 1997.

   (c) Hill, Robin K. (2018, May 21), "Articulation of Decision Responsibility", *BLOG@CACM*, https://cacm.acm.org/blogs/blog-cacm/227966-articulation-of-decision-responsibility/fulltext

18.2 Introduction

In 2004, when I first taught the course that this text is based on, the question of whether to trust decisions made by computers was not much discussed. But since the advent of self-driving cars, it has become a more pressing issue, with immediate, real-life, practical implications as well as moral and legal ramifications.

Before we consider the ethical issue of whether computers *should* make decisions for us and the clearly related question that is the title of James Moor's 1979 essay—Are there decisions computers should *never* make?, there are two prior questions: *What **is** a "decision"?* And: *Do computers "make decisions" at all?*

Further Reading:
There are other automated vehicles besides self-driving cars, for which some of the same issues arise. Markoff 2015 asks, in the wake of a commercial-airline pilot who who committed suicide, thereby killing all aboard his plane, whether pilots are necessary at all. See also two follow-up letters to the editor (*New York Times* (14 April): D3, http://www.nytimes.com/2015/04/14/science/letters-to-the-editor.html) that point out situations in which (a) computers *should* take control over from humans or (b) humans might make *better*—because more creative or context-dependent—decisions than computers. Keep in mind that another vehicle that many of us use frequently was once only human-operated but is now completely automated (and I doubt that any of us would know what to do if it failed): elevators.

Halpern 2016 discusses technical and ethical issues concerning the automated decisions made by driverless vehicles. Monticello 2016 offers observations on their relative safety.

18.3 What Is a Decision?

Roughly, a decision is a choice made from several alternatives, usuall for some reason (Eilon, 1969). Let's begin by considering three kinds of decisions:

1. A decision could be the result of an arbitrary choice, such as flipping a coin: heads, we'll go out to a movie; tails, we'll stay home and watch TV.

2. A decision could be the solution to a purely logical or mathematical problem that requires some calculation.

3. A decision could be the result of investigating the pros and cons of various alternatives, rationally evaluating these pros and cons, and then choosing one of the alternatives based on this evaluation.

18.4 *Do* Computers Make Decisions?

At first glance, there is a simple answer to the question whether computers make decisions: Yes; computers can easily make the first kind of decision for us. Moreover, any time that a computer solves a logical or mathematical problem, it has made a decision of the second kind.

Can computers make decisions of the third kind? Surely, it would seem, the answer is, again, 'yes': Computers can play games of strategy, such as checkers, chess, and Go, and they can play them so well that they can beat the (human) world champions. Such games involve choices among alternative moves that must be evaluated, with, one hopes, the best (or least worst) choice being made. Computers can do this.

Further Reading:
See, for example, https://en.wikipedia.org/wiki/Chinook_(draughts_player); https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer); Silver et al. 2016; Campbell 2018; Kasparov 2018; Silver et al. 2018.

Of course, it is not just a physical *computer* that makes a decision. Arguably, it is a computer *program* being executed by a computer that makes the decision, although I will continue to speak as if it is the computer that decides.

And, of course, it is not just a computer *program* that makes a decision. Computer programs are written by *humans*. (Even computer programs that are written by computers are the output of computer programs that were written by humans.) And humans, of course, can err in various ways, unintentionally or otherwise. These errors can be inherited by the programs that they write.

Robin K. Hill has argued that computers do *not* make decisions, precisely *because* their programs are written by humans. It is the humans who make the decisions that are subsequently encoded in the programs:

> [M]achines and algorithms have no such capacity as is normally connoted by the term "decision" .... Algorithms are not biased, because a program does not make decisions. The program implements decisions made elsewhere. (Hill, 2018)

To a large extent, this is, of course, correct. There is no question that the way in which a computer makes a decision was initially determined by its human programmer.

But what happens when the human programmer is out of the picture, and the computer running that program is what we rely on? In any given situation, when the computer has to act or to make or recommend a decision, it will do so autonomously and in the light of the then-current situation, without consulting the programmer (or being able to consult the programmer):

> It is a common misconception that because a machine such as a guided missile was originally designed and built by conscious man [sic], then it must be truly under the immediate control of conscious man. Another variant of this fallacy is "computers do not really play chess, because they can only do what a human operator tells them". ... When it is actually playing, the computer is on its own, and can expect no help from its master. All the programmer can do is to set the computer up *beforehand* in the best way possible .... (Dawkins, 2016, pp. 66–67)

Typically, human delegation of decision-making powers to computers happens in cases where large amounts of data are involved in making the decision or in which decisions must be made quickly and automatically. And, of course, given that one of the goals of CS is to determine what real-world tasks are computable (§3.15.2.1.1), finding out which *decisions* are computable is an aspect of that.

In any case, humans *might* delegate such power to computers. So, another way to phrase our question is: What areas of our lives should be computer-controlled, and what areas should be left to human control? Are there decisions that non-human computers could *not* make as well as humans? For instance, there might be situations in which there are sensory limitations that prevent computer decisions from being fully rational. Or there might be situations in which a decision requires some (presumably non-computable) empathy. On the other hand, there might be situations in which a computer might have an advantage over humans: It is impossible (or at least less likely) for a computer to be swayed by such things as letter-writing campaigns, parades, etc. Such tactics were used by General Motors in their campaign to persuade some communities to open new plants for their Saturn cars (Russo, 1986).

To answer these questions, we need to distinguish between what *is* the case and what *could be* the case. We could try to argue that there are some things that are *in principle* impossible for computers to do. Except for computationally impossible tasks (such as the Halting Problem, §7.7), this might be hard to do. But we should worry about the possible future now, so that we can be prepared for it if it happens. (Recall the italicized quotation in the first epigraph to this chapter.)

Whether there are, *now*, decisions that a computer could not make as well as a human is an empirical question. It is capable of investigation, and, currently, the answer is unknown. Many, if not most, of the objections to the limitations of computers are best viewed as research problems: If someone says that computers can't do X, we should try to make ones that do X.

This is crucial: Humans should be critical thinkers. There is a logical fallacy called the Appeal to Authority (see §2.7): Just because an authority figure *says* that something *is* true, it does not logically follow that it *is* true. Although logicians sometimes warn us about this fallacy, it *is* acceptable to appeal to an authority (even a computer!) as long as the final decision is yours. *You* can—and must—decide whether to believe the authority or to trust the computer. You should also be able (and willing!) to *question* the authority—or the computer!—so as to understand the reasons for the decision.

So, even if we allow computers to make (certain) decisions for us, it is still important for us to be able to understand those decisions. When my son was first learning how to drive, I did not want him to rely on the vehicle's automated "dynamic cruise control" system, because I wanted him to know how and when to slow down or speed up on a superhighway. Once he knew how to do that, then he could rely on the car's computer making those decisions for him, because, if the computer's decision-making skills went awry or were unavailable (for example, the laser-controlled system on my 2008 Toyota Sienna is designed not to work when the windshield wipers are on, or when the car ahead of you is dirty!), he should know how to do those things himself.

Further Reading:
For a discussion of this point in the context of airplane pilots, see Nicas and Wichter 2019.

Another issue that arises from the fact that computer programs are written by humans is whether, given the occasional irrationality of human behavior, computer-made decisions really *are* rational, which we now turn to.

18.5 Are Computer Decisions *Rational*?

When a decision has an impact on our lives, we would like the decision-making process to be rational, whether it is a human making the decision or a humanly written program "making" it. Can computers (and the programs that they execute) be completely rational? It certainly seems that some computers can make rational decisions for us. The kinds of decision making described in the previous section *seem* to be purely rational. And aren't rule-based algorithms purely rational?

Consider an algorithm that does not involve any random or interactive procedure produced by a non-rational oracle (of the kind discussed in §11.4.4). Presumably, if the decision is made by a computer that is following such an algorithm, then that decision is a purely rational one. By 'rational', I don't necessarily mean that it is a *purely* logical decision. (Recall our discussion in §2.6 of kinds of rationality.) It may, for instance, involve empirical data, which might be erroneous in some way: It might be incomplete, it might be statistically incorrect, it might be biased, and so on.

Another potential problem is if the algorithm requires exponential time or is NP-complete, or even if it merely would take longer to come up with a decision than the time needed for action. In that case, or if there is no such algorithm, we would have to rely on a "satisficing" heuristic in the sense of an algorithm whose output is "near enough" to the "correct" solution (as we discussed in §3.15.2.3). But this is still a kind of rationality—what Simon called "bounded" rationality (§§2.6.1.4, 3.15.2.3, 2.6.1.5, 5.7, 11.4.5.2).

But just as a logical argument can be valid even if its premises are false (recall §2.6.1.1), an algorithm can be syntactically and semantically correct even if its input is not ("garbage in, garbage out"; recall §§8.11.2.1, 11.3.3, and 16.3.1.2). But, as long as there is an algorithm that can be studied to see how it works, or as long as the program can explain how it came to its decision, I will consider it to be rational.

Whether computers ought to make decisions for us is equivalent to whether our decisions ought to be made algorithmically. And that suggests that it is equivalent to whether our decisions ought to be made *rationally*. If there is an algorithm for making a given decision, then why not rely on it? After all, wouldn't that be the rational thing to do?

One might even argue that there is no such thing as computer ethics. All questions about the morality of using computers to do something are really questions about the morality of using algorithms. As long as algorithms are rational, questions about the morality of using them are really questions about the morality of being rational, and it seems implausible to argue that we shouldn't be rational. This suggests that James Moor's (1979) question, "Are there decisions computers should never make?", should

really have nothing to do with computers! The question should really be: Are there decisions that should not be made on a rational basis?

But then the important question becomes: Are the algorithms really rational? And how would we find out? Before looking at these questions, let's assume, for the moment, that a decision-making algorithm *is* rational. The next question is: Should we let it make a decision *for* us?

18.6 Should Computers Make Decisions *for* Us?

A paragraph deeply embedded in a 2004 science news article suggests that people find it difficult to accept rational recommendations, even if they come from other *people*, not computers. The article reports on evidence that a certain popular and common surgical procedure had just been shown to be of no benefit: "Dr. Hillis said he tried to explain the evidence to patients, to little avail. 'You end up reaching a level of frustration,' he said. 'I think they have talked to someone along the line who convinced them that this procedure will save their life' " (Kolata, 2004). Perhaps the fundamental issue is not whether computers should make rational decisions or recommendations, but whether or why humans should or don't *accept* rational advice!

There are several reasons why we might want to let a computer make a decision *for* us: Computers are much *faster* than we are at evaluating options, they can evaluate *more* options than we could (in the same amount of time), they are better at evaluating more *complex* options, they can have *access* to more relevant data. And, in many situations in the modern world, we might simply have no other option but to allow computers to make decisions for us. So, whether it is a good idea or a bad idea to let them do so, it is a simple fact that they do.

And, after all, is this any different from letting *someone* else make a decision for us—someone who is wiser, or more knowledgeable, or more neutral than we are? If it is not any different, then—in both cases—there is still a question that should always be raised: Should we *trust* that other agent's decision? Before looking into this, there is an intermediate position that we should consider.

18.7 Should Computers Make Decisions *with* Us?

Moor suggests that, if computers can make certain decisions at least as well as humans, then we should let them do so, and it would then be up to us humans to accept or reject the computer's decision. After all, when we ask for the advice of an expert in medical or legal matters, we are free to accept or reject that advice. Why shouldn't the same be true for computer decision making?

In other words, rather than simply letting computers (or other humans) make decisions *for* us, we should *collaborate* on the decision-making process, treating the computer (or the human expert) as a useful source of information and suggestions to help *us* make the final decision. As we noted in §18.5, this might not always be possible: There may (and most likely *will*) be situations in which we do not have the *time* to evaluate all options before a decision has to be made.

But there are also many cases in which we do need to collaborate:

> The systems that land airplanes are hybrids—combinations of computers and people—exactly because the unforeseeable happens, and because what happens is in part the result of human action, requiring human interpretation.
> (B.C. Smith 1985, §7, p. 24, col. 2)

The situation that Smith mentions has been explored in depth by the anthropologist and cognitive scientist Edwin Hutchins (Hutchins, 1995a,b; Hollan et al., 2000; Casner et al., 2016). Hutchins's theory of "distributed cognition" uses examples of large naval vessels navigating and of jet pilots working in their cockpits. In both of these cases, it is neither the machines alone (including, of course, computers) nor humans alone who make decisions or do the work, but the combination of them—indeed, in the case of large naval vessels, it is *teams* of humans, computers, and other technologies. Hutchins suggests that this combination constitutes a "distributed" mind. Similarly, the philosophers Andy Clark and David Chalmers (1998) have developed a theory of "extended cognition", according to which our (human) minds are not bounded by our skull or skin, but "extend" into the external world to include things like notebooks, reference works, and computers.

Question for the Reader:
Do these examples constitute uses of *oracles* as external sources of information, as discussed in §11.4.4?

But *must* it be the case that complex decision-making systems be such "hybrids" or "team efforts"? Smith, Hutchins, and Clark and Chalmers developed their theories long before the advent of self-driving cars. Even as of this writing (2019), it remains to be seen whether self-driving cars will continue to need human intervention (remember: self-driving elevators don't need very much of it!): Steven E. Shladover (2016) argues that a level called "conditional automation", in which computers and humans work together, will be *harder* to achieve than the more fully automated level called "high automation" (see also Casner et al. 2016). Nevertheless, such "hybrid" or "extended" systems will probably remain a reality.

Further Reading:
Hafner 2012 observes: "For diagnosticians . . . , software offers a valuable backup. Whether it can ever replace them is another question." Mearian 2013 says that they might be able to, and Frakt 2015 suggests that, although "machines" do not yet "outperform doctors . . . in some areas of medicine they can make the care [that] doctors deliver better."

Johnson and Verdicchio 2017 discuss "a confusion about the notion of 'autonomy' that induces people to attribute to machines something comparable to human autonomy, and a 'sociotechnical blindness' that hides the essential role played by humans at every stage of the design and deployment of an AI system" (p. 575), concluding that "there are many reasons for concern and even fear about autonomous systems, but these reasons have to do with the human actors in AI systems and not merely the computational artefacts in them" (p. 589).

18.8 Should We *Trust* Decisions Computers Make?

Whether we let computers make decisions *for* us, or work jointly *with* them to make decisions, we usually *assume* that any decision that they make or advice that they give is based on good evidence (as input) and on rational algorithms (that process the input). Note, again (§18.5), the similarity with logical inference, which begins with axioms or premises ("input") and then "processes" that "input" to derive a valid conclusion. (Recall the Digression on Formal Systems and Turing Machines in §14.3.2.1.) In both cases, for the decision (or advice) to be "good" or for the conclusion to be true, the input must be correct or true *and* the processing must be correct. But how do we know if they are? (Remember the warnings in §2.5.3 about making assumptions!)

An algorithm's trustworthiness is a function of its input and its processing. Is it getting all of the relevant input? Is the input accurate, or might there be a problem with its sensors or how it interprets the input? Is the algorithm correct? Can we understand it? Can we explain or justify its decisions? Is it (intentionally or unintentionally) biased in some way, perhaps due to the way that its human programmer wrote it or—in the case of a machine-learning program—what its initial training set was?

How can computer decision-making competence be judged? One answer is: in the same way that *human* decision-making competence is judged, namely, by means of its decision-making *record* and its *justifications* for its decisions.

Let's briefly consider a computer's track record first. Consider once more a documentationless computer found in the desert. Suppose that we discover that it successfully *and reliably* solves a certain type of problem for us. Even if we cannot understand why or how it does that, there doesn't seem to be any reason *not* to trust it. So, why should justifications matter? After all, if a computer constantly bests humans at some decision-making task, why should it matter *how* it does it?

Further Reading and Questions for the Reader:
Maybe we would be better off not knowing! For a science-fiction treatment of this issue, though not in the context of computers, see Arthur C. Clarke's *Childhood's End* (1953).

If all that matters is a decision-making computer's track record, and if its algorithm *cannot* be understood (either because it is too complex or because it is a "black box" algorithm (§3.9.5)), does that mean that we have to take its decisions merely on *faith*? Or are there decisions that should not be made by algorithms that are so complex that we cannot understand them?

On the other hand, consider these remarks by Daniel Dennett:

> Artifacts already exist . . . with competences so far superior to any human competence that they will usurp our authority as experts, an authority that has been unquestioned since the dawn of the age of intelligent design. And when we ceded hegemony to these artifacts, it will be for very good reasons, both practical and moral. Already it would be *criminally negligent* for me to embark with passengers on a transatlantic sailboat cruise without equipping the boat with several GPS systems. . . .
>
> Would *you* be willing to indulge your favorite doctor in her desire to be an old-fashioned "intuitive" reader of symptoms instead of relying on a computer-based system that had been proven to be a hundred times more reliable at finding rare, low-visibility diagnoses than any specialist? (Dennett, 2017, pp. 400–401)

Would it be *irrational not* to take such decisions or advice on faith? (We'll return to the notion of faith in §18.8.2.)

For a philosophical investigation of the nature of trust, see Baier 1986.

Presumably, however, decision-making computers *should* be accountable for their decisions, and knowing what their justifications are helps in this accounting. In fact, the European Union has passed a law giving users the right to have an explanation of a computer's decision concerning them (https://en.wikipedia.org/wiki/Right_to_explanation). The justifications, of course, need not be the same as human justifications. For one thing, human justifications might be wrong or illogical.

Further Reading:
Heingartner 2006, discussing whether computers can make better decisions than humans, observes that "mathematical models generally make more accurate predictions than humans do. . . . The main reason for computers' edge is their consistency—or rather humans' inconsistency—in applying their knowledge." For scientific research on wrong or illogical explanations by humans, see, especially, Tversky and Kahneman 1974. Also see Wainer 2007 on humans' difficulty in reasoning about probability and statistics. For other readings on humans' difficulty in reasoning, see http://www.cse.buffalo.edu/~rapaport/575/reasoning.html.

But what if justifications are unavailable or, perhaps worse, misleading? Let's take a look at these two possibilities.

18.8.1 The Bias Problem

Could there be a hidden bias in the way that the algorithms were developed? For example, the training set used to create a machine-learning algorithm might have been biased (again, perhaps unintentionally). This does not have to be due to any intention on the part of the programmer to deceive. (But recall question 7 from §5.10.) The bias might not be evident until the algorithm is deployed.

Recall Hill's point that algorithms are written by humans. And humans, of course, have

> idiosyncratic foibles .... The mostly white men who built the tools of social networks did not recognize the danger of harassment, and so the things they built became conduits for it. If there had been women or people of color in the room, ... there might have been tools built to protect users ... (Bowles, 2019)

What kinds of problems can such "foibles" or biases lead to?

> Users discovered that Google's photo app, which applies automatic labels to pictures in digital photo albums, was classifying images of black people as gorillas. Google apologized; it was unintentional.
>
> ... Nikon's camera software ... misread images of Asian people as blinking, and ... Hewlett-Packard's web camera software ... had difficulty recognizing people with dark skin tones.
>
> This is fundamentally a data problem. Algorithms learn by being fed certain images, often chosen by engineers, and the system builds a model of the world based on those images. If a system is trained on photos of people who are overwhelmingly white, it will have a harder time recognizing nonwhite faces.
>
> ... ProPublica ... found that widely used software that assessed the risk of recidivism in criminals was twice as likely to mistakenly flag black defendants as being at a higher risk of committing future crimes. It was also twice as likely to incorrectly flag white defendants as low risk.
>
> The reason those predictions are so skewed is still unknown, because the company responsible for these algorithms keeps its formulas secret ....
> (Crawford, 2016)

Perhaps the ethical issues really concern the nature of different kinds of algorithms. "Neat" algorithms are based on formal logic and well-developed theories of the subject matter of the algorithm. "Scruffy" algorithms are not necessarily based on any formal theory. (These terms were originally used to describe two different approaches to AI (https://en.wikipedia.org/wiki/Neats_and_scruffies), but they can be used to describe any algorithm.) "Heuristic" algorithms, as we saw in §3.15.2.3, don't necessarily give you a correct solution to a problem, but are supposed to give one that is near enough to a correct solution to be useful (that is, one that "satisfices"). Machine-learning algorithms are trained on a set of test cases, and "learn" how to solve problems based on those cases and on the particular learning technique used.

If a "neat" algorithm is "correct"—surely, a big "if"—then there does not seem to be any moral reason not to use it (not to be "correctly rational"). If the algorithm is "scruffy", then one might have moral qualms. If the algorithm is a heuristic (perhaps as in the case of expert systems), then there is no more or less moral reason to use it then

there is to trust a human expert. If the algorithm was developed by machine learning, then its trustworthiness will depend on its training set and learning method.

Further Reading:
Savage 2016 is a discussion of how "researchers are trying to identify ... and root ... out" biases found in classification algorithms. In contrast, Metz 2019a reports on how minimally trained non-professionals are the people who supply the training for machine-learning systems, which raises the question of how accurate those systems algorithms could be.

18.8.2 The Black-Box Problem

> No one really knows how the most advanced algorithms do what they do. That could be a problem. —Will Knight (2017)

> Algorithmic fairness: If your tool cannot explain its results, you shouldn't use it. —Venkatasubramanian (2018)

> ... a software engineer ... and author of the study [on AI detection of lung cancer] said, "How do you present the results in a way that builds trust with radiologists?" The answer, she said, will be to "show them what's under the hood." —Denise Grady 2019

There are at least four sources of problems that can make a computer's decision untrustworthy:

1. the decision-making criteria encoded in the algorithm, either by its programmer (or programmers) or by the machine-learning program that developed those criteria from test cases,
2. those test cases themselves,
3. the computer program itself, and
4. the data on which a given decision is based.

Let's assume, for the sake of the argument, that the input data (#4) are as complete and accurate as possible. Let's also assume (although this is a much larger assumption) that the algorithm (#3) has been formally verified. That leaves the decision-making criteria and any test cases as the primary focus of attention.

At the present stage in the development of computers, two ways in which these criteria find their way into an algorithm are, first, through the human programmer and, second, through machine learning. Of course, a machine-learning algorithm gets its test cases from a human (or from a database that was generated by another program that was written by a human), and it gets its machine-learning technique from its human programmer. But once the human is out of the picture, and the algorithm is left to fend for itself, so to speak, it is to the algorithm that we must turn for explanations.

Consequently, one important issue concerning computers that make decisions for (or with) us is whether they can, or should, *explain* their decisions. Two kinds of

algorithms are relevant to this question. One kind is the symbolic or logical algorithm that has such an explanatory capability. It could have that in one of two ways: A user could examine a trace of the algorithm, or a programmer could write a program that would translate that trace into a natural-language explanation that a user could understand. The other kind of algorithm is one that is based on a neural-network or a statistical, machine-learning algorithm. Such an algorithm might not be able to explain its behavior, nor might its programmer or a user be able to understand how or why it behaves as it does.

As an example, a typical board-game-playing program might have a representation of the board and the pieces, an explicit representation of the rules, and an explicit game tree that allows it to rationally choose an optimal move. Such a program could easily be adapted to explain its moves. It does not have to, of course. The computer scientist Peter Scott suggested[1] that "even the Turing Test does not require the agent to explain clearly how s/he/it is reasoning." Arguably, however, the Turing Test does require it, because the interrogator can always ask something like "Why do you believe that?" or "Why did you do that?", and, to pass the test, the interlocutor (human or computer) must be able to give a plausible answer.

But AlphaGo, the recent Go-playing program that beat the European Go champion, was almost entirely based on neural networks and machine-learning algorithms (Silver et al., 2016; Vardi, 2016). As an editorial accompanying Silver et al. 2016 put it:

> . . . the interplay of its neural networks means that a human can hardly check its working, or verify its decisions before they are followed through. As the use of deep neural network systems spreads into everyday life—they are already used to analyse and recommend financial transactions—it raises an interesting concept for humans and their relationships with machines. The machine becomes an oracle; its pronouncements have to be believed.
>
> When a conventional computer tells an engineer to place a rivet or a weld in a specific place on an aircraft wing, the engineer—if he or she wishes—can lift the machine's lid and examine the assumptions and calculations inside. That is why the rest of us are happy to fly. Intuitive machines will need more than trust: they will demand faith. (*Nature* Editors, 2016)

Should they "demand faith"? Or should laws (such as those in the European Union) *require* transparency or explainability and thus rule out "black box" machine-learning algorithms of the kind discussed in §3.9.5? Relying on a successful but unexplained computer's decisions might not necessarily mean that we are taking its decisions on faith. After all, its successes would themselves be evidence for its trustworthiness, just as an axiom's usefulness in mathematical derivations is evidence in its favor even though—by definition—it cannot be proved.

[1] Personal communication, 23 April 2017.

Digression and Further Reading: Connectionist vs. Symbolic Algorithms
Scott went on to say,

> Some say a temporary truce has been recognized, but there is still no hint of a permanent peace treaty between the connectionist and symbolist advocates. I am betting that controversy will go on for a long time.

The current apparent inability of connectionist or neural-network algorithms to explain their behavior (or to have their behavior explained by others) while at the same time being better at certain tasks than symbolic algorithms that *can* explain their behavior suggests that both kinds of mechanisms are needed.

For example, there is a two-way interaction between connectionist-like cognition and symbolic-like cognition in human learning: When my son was learning how to drive, I realized that I had to translate my *instinctive* (connectionist-like) behavior for making turns into *explicit* (symbolic) instructions, something along the lines of "put your foot on the brake to slow down, make the turn, then accelerate slowly". But to do that, I had to observe what *my* instinctive behavior was. Presumably, my son would follow the *explicit* instructions until they became second nature to him (that is, "followed" implicitly or instinctively), until such time as he might teach his child to drive, and the cycle would repeat. (On this topic and in connection with AlphaGo, Vardi 2016 discusses "Polanyi's Paradox: 'We can know more than we can tell ... The skill of a driver cannot be replaced by a thorough schooling in the theory of the motorcar.' " And recall our earlier discussions in §§3.6.1 and 3.14.4 of knowing-how vs. knowing-that.)

For other arguments on the value of symbolic computation, see Levesque 2017; Bringsjord et al. 2018; Landgrebe and Smith 2019a.

Historical Digression:
It is worth noting that, as a matter of historical fact, the "black box" issue can affect symbolic computer programs, too. When spreadsheet programs were relatively new, one writer made the following observations:

> The accuracy of a spreadsheet model is dependent on the accuracy of the formulas that govern the relationships between various figures. These formulas are based on assumptions made by the model maker. An assumption might be an educated guess about a complicated cause-and-effect relationship. It might also be a wild guess, or a dishonestly optimistic view. ...
>
> In 1981, electronic spreadsheets were just coming into their own, and ... sophisticated modeling ... was still done chiefly on mainframe computers. The output ... wasnt in the now-familiar spreadsheet format; instead, the formulas appeared in one place and the results in another. You could see what you were getting. That cannot be said of electronic spreadsheets, which dont display the formulas that govern their calculations.
>
> As Mitch Kapor explained, with electronic spreadsheets, "You can just randomly make formulas, all of which depend on each other. And when you look at the final results, you have no way of knowing what the rules are, unless somebody tells you." (Levy, 1984)

Further Reading:
In Brachman 2002, a leading AI researcher suggests how and why decision-making computers should be able to explain their decisions.

Greengard 2009 observes that "Today's automated systems provide enormous safety and convenience. However, when glitches, problems, or breakdowns occur, the results can be catastrophic."

Diakopoulos 2016, pp. 56–57 says: "It is time to think seriously about how the algorithmically informed decisions now driving large swaths of society should be accountable to the public. ... While autonomous decision making is the essence of algorithmic power, the human influences in algorithms are many ...."

Simonite 2017 reports on arguments to "end 'black box' algorithms in government".

Sullivan 2019 discusses many of the issues concerning "deep neural networks" raised in this section, and has many things to say about the nature of understanding and explanation, arguing that "it is not the complexity or black box nature of a model that limits how much understanding the model provides. Instead, it is a lack of scientific and empirical evidence supporting the link that connects a model to the target phenomenon that primarily prohibits understanding."

18.9 Are There Decisions Computers *Must* Make for Us?

> Commercial airplanes are what we'd call self-driving except at takeoff and landing, and the result is that it's now nearly impossible for a cruising jet to fall out of the sky without malice or a series of compounding errors by the pilots. (Lethal computer glitches are so rare that if they appear even twice among tens of millions of flights, as in the case of Boeing's 737 MAX 8, the industry goes into crisis.) People get the willies at the idea of putting their lives in the hands of computers, but there's every reason to think that, as far as transportation goes, we're safer in their care. —Nathan Heller (2019, p. 28)

Having the *ability* to evaluate a computer's reasons for its decisions assumes the *willingness* to do so. But remember Simon's problem of bounded rationality: We usually don't have the time or ability to evaluate *all* the relevant facts before we need to act. What about emergencies, or other situations in which there is no time for the human who must act to include the computer's recommendation in his or her deliberations?

On July 1, 2002, a Russian airliner crashed into a cargo jet over Germany, killing all on board, mostly students. The Russian airliner's flight recorder had an automatic collision-avoidance system that instructed the pilot to go higher (to fly over the cargo jet). The human air-traffic controller told the Russian pilot to go lower (to fly under the cargo jet). According to science reporter George Johnson (2002a), "Pilots tend to listen to the air traffic controller because they trust a human being and know that a person *wants* to keep them safe" (my italics). But the human air-traffic controller was tired and overworked. And the collision-avoidance computer system didn't "want" anything; it simply made rational judgments. The pilot followed the human's decision, not the computer's, and a tragedy occurred.

There is an interesting contrasting case. In January 2009, after an accident involving birds that got caught in its engines, a US Airways jet "landed" safely on the Hudson River in New York City, saving all on board and making a hero out of its pilot. Yet William Langewiesche (2009) argues that it was the plane, with its computerized "fly by wire" system, that was the real hero. In other words, the pilot's heroism was due to his willingness to accept the computer's decision.

Further Reading:
For contrasting discussions of the US Airways case, see Haberman 2009; Salter 2010.

For another airplane incident (Qantas Flight 72), involving "rogue" computers, see O'Sullivan 2017.

The 2019 crashes of two Boeing 737 Max 8 jets are another important case study. As of this writing (April 2019), the full story is not yet known, but the crashes seem to have been due to some combination of one or more of the following: a faulty sensor that input erroneous information to certain software, the software itself that may have been flawed, or lack of proper pilot training in the use of the software. For a summary, see the *Wikipedia* article, "Boeing 737 MAX Groundings", https://en.wikipedia.org/wiki/Boeing_737_MAX_groundings; see also Nicas et al. 2019.

Zremski 2009 explores the possibility that a decision-making computer might make things more difficult for a human who is in the decision-making loop. Halpern 2015 points out, among other things, that an "overreliance on automation, and on a tendency to trust computer data even in the face of contradictory physical evidence, can be dangerous", in part because the human in the decision-making loop might not be paying attention: "over half of all airplane accidents were the result of the mental autopilot brought on by actual autopilot".

18.10 Are There Decisions Computers *Shouldn't* Make?

Let's suppose that we have a decision-making computer that explains all of its decisions, is unbiased, and has an excellent track record. Are there decisions that even such a computer *should* never make?

The computer scientist Joseph Weizenbaum (1976) has argued that, even if a computer *could* make decisions as well as, or even better than, a human, they shouldn't, *especially* if their reasons differ from ours. And Moor points out that, possibly, computers shouldn't have the *power* to make (certain) decisions, even if they have the *competence* to do so (at least as well as, if not better than, humans).

But, if they have the competence, why shouldn't they have the power? For instance, suppose a very superstitious group of individuals makes poor medical decisions based entirely on their superstitions; shouldn't a modern physician's "outsider" medicine take precedence? And does the fact that computers are immune to human diseases mean that they lack the empathy to recommend treatments to humans?

Moor suggests that, although a computer should make rational decisions for us, a computer should *not* decide what our basic goals and values should be. Computers should help us *reach* those goals or *satisfy* those values, but they should not *change*

them. But why not? Computers can't be legally or morally responsible for their decisions, because they're not persons. At least, not yet. But what if AI succeeds? We'll return to this in Chapters 19 and 20. Note, by the way, that for many legal purposes, non-human corporations *are* considered to be persons.

Batya Friedman and Peter H. Kahn, Jr. (1997) argue that humans *are*—but computers are *not*—*capable* of being moral agents and, therefore, computers should be designed so that:

1. humans are *not* in "merely mechanical" roles with a diminished sense of agency,

   and

2. computers *don't* masquerade as agents with beliefs, desires, or intentions.

Let's consider point 1: Friedman and Kahn argue that computers should be designed so that humans *do* realize that they (the humans) *are* moral agents. But what if the computer has a better decision-making track record than humans? Friedman and Kahn offer a case study of APACHE, a computer system that can make decisions about when to withhold life support from a patient. It is acceptable if it is used as a tool to aid *human* decision makers. But human users may experience a "diminished sense of moral agency" when using it, presumably because a computer is involved.

But why? Suppose APACHE is replaced by a textbook on when to withhold life support, or by a human expert. Would either of those diminish the human decision-maker's sense of moral agency? In fact, wouldn't human decision makers be remiss if they *failed* to consult experts or the writings of experts? So wouldn't they also be remiss if they failed to consult an expert computer?

Perhaps humans would experience this diminished sense of moral agency for the following reason: If APACHE's decisions exhibit "good performance" and *are* more relied on, then humans may begin to yield to its decisions. But why would that be bad?

Turning to point 2, Friedman and Kahn argue that computers should be designed so that humans *do* realize that computers are *not* moral agents. Does this mean that computers should be designed so that humans *can't* take Dennett's (1971) "intentional stance" towards them? (Recall our discussion of this in §12.4.4.1.1.)

But what if the computer *did* have beliefs, desires, and intentions? AI researchers are actively designing computers that either really have them, or else that are best understood as if they had them. Would they not then be moral agents? If not, why not? According to Dennett (1971), some computers can't *help* "masquerading" as belief-desire-intention agents, because that's the best way for *us* to *understand* them.

Friedman and Kahn argue that we should be careful about anthropomorphic user-interfaces, because the *appearance* of beliefs, desires, and intentions does not imply that they really *have* them. This is a classic theme, not only in the history of AI, but also in literature, and cinema. And this is at the heart of the Turing test in AI, to which we now turn.

Further Reading (and Viewing):
OHeigeartaigh 2013 is a blog that considers many of the issues discussed in Moor 1979.

Friedman and Kahn argue that programmers should not design computer systems so that users think that the systems are "intelligent". The April 2004 issue of *Communications of the ACM* (Miller, 2004) has a whole section devoted to this.

Asimov 1950, about decisions made by machines, is a fictional approach to Moor's question.

On belief-desire-intention theory, see, for example, Kumar 1994, 1996, and the *Wikipedia* article, "Belief-desire-intention software model" (https://en.wikipedia.org/wiki/Belief-desire-intention_software_model).

As a "classic theme" in AI, I am thinking primarily of Joseph Weizenbaum's "Eliza" program, which, in its most famous version, allegedly simulated a Rogerian psychotherapist. See Weizenbaum 1966, 1967, 1976; Shapiro and Kwasny 1975; Winograd 1983, Ch. 2; and http://www.cse.buffalo.edu/~rapaport/572/S02/proj1.html

In literature, I highly recommend *Galatea 2.2: A Novel*, by Richard Powers (1995) (in which a cognitive-science grad student is assigned the task of programming a computer to pass the PhD exam in English literature; the grad student falls in love with the computational cognitive agent) and *Do Androids Dream of Electric Sheep?*, by Philip K. Dick (1968) (which was the basis of the film *Blade Runner*; see also Beebee 2017).

In cinema, there are Steven Spielberg's *A.I. Artificial Intelligence* (2001), Spike Jonze's *Her* (2013), and Alex Garland's *Ex Machina* (2014), to name just three.

18.11 Discussion Questions for the Reader

In §§12.4.4.1.2.2 and 17.8.1, we discussed how to describe what a computer or a person is doing. Is a universal Turing machine that is running an addition program adding, or "merely" fetching and executing the instructions for adding? If I use a calculator or a computer, or if a robot performs some action, who or what is "really" doing the calculation or the computation, or the action: Is it the calculator (computer, robot)? Or me? When I use a calculator to add, am I adding or "merely" pushing certain buttons? (Compare this real-life story: I was making waffles "from scratch" on a waffle iron. The 7-year-old son of friends who were visiting was watching me and said, "Actually, *you're* not making it; it's the thing [what he was trying to say was that it was *the waffle iron* that was making the waffles]. But you set it up, so you're the cook.")

This issue is related to the question of who or what is *morally* responsible for an action. Who (or what) is morally responsible for decisions made, or actions taken, by computers? Is it the computer? Is the the human who accepts the computer's decision? Is it the human who programmed the computer?

Further Reading:
On machine ethics and robots, see Anderson and Anderson 2007, 2010; Wallach and Allen 2009; Wagner and Arkin 2011. Should artificial intelligences be allowed to kill? Sparrow 2007 "considers the ethics of the decision to send artificially intelligent robots into war . . . ."

It is also related to issues in (math) education: Suppose that a student knows how to use a calculator to add; does that student know how to add?

Further Reading:
Aref 2004 suggests (but does not discuss) that supercomputers might make decisions that we could not understand:

> As we construct machines that rival the mental capability of humans, will our analytical skills atrophy? Will we come to rely too much on the ability to do brute-force simulations in a very short time, rather than subject problems to careful analysis? Will we run to the computer before thinking a problem through?... A major challenge for the future of humanity is whether we can also learn to master machines that outperform us mentally.

On the question "will our analytical skills atrophy?", you might enjoy Isaac Asimov's (1957) science-fiction story, "The Feeling of Power", which is about a human who rediscovers how to do arithmetic even though all arithmetical problems are handled by computers, and then the computers break down.

Chapter 19

Philosophy of Artificial Intelligence

Version of 29 August 2019,[1] DRAFT © 2004–2019 by William J. Rapaport

> Computers . . . are, after all, in the business of making mechanical what smacks of vitalism to most scientists.
> —Allan M. Collins & M. Ross Quillian (1972, p. 313)

> The computer revolution will affect philosophy most profoundly by providing a powerful new set of models and metaphors for thinking about thinking. Can thinking be reproduced by hardware running software? Is the brain hardware? Are neural patterns software? Can the interaction of pattern and patterned substance create thought? Can thought and intelligence derive from the complex interactions of unthinking and unintelligent parts?
> —Peter Suber (1988, p. 89)

> With a large number of programs in existence capable of many kinds of performances that, in humans, we call thinking, and with detailed evidence that the processes some of these programs use parallel closely the observed human processes, we have in hand a clear-cut answer to the mind-body problem: How can matter think and how are brains related to thoughts?
> —Herbert Simon (1996a, p. 164)[2]

[1] Portions of this chapter are adapted from Rapaport 2000b.

[2] Simon's answer is that certain "patternings in matter, in combination with processes that can create and operate upon such patterns" can do the trick (Simon, 1996a, p. 164). For more on patterns, see Hillis 1998 and §9.6, above, on computers as "magic paper".

Figure 19.1: https://www.comicskingdom.com/crankshaft/2005-03-02, ©2005 by Mediagraphics, Inc.

19.1 Required Readings:

1. Turing, Alan M. (1950), "Computing Machinery and Intelligence", *Mind* 59: 433–460, http://mind.oxfordjournals.org/content/LIX/236/433.full.pdf+html
2. Searle, John R. (1980), "Minds, Brains, and Programs", *Behavioral and Brain Sciences* 3: 417–457, http://cogprints.org/7150/1/10.1.1.83.5248.pdf

19.2 Introduction

Like computer ethics, the philosophy of artificial intelligence (AI) is a large and long-standing discipline in its own right. In this chapter, we will focus on only two main questions: *What is AI?* And: *Is AI possible?* For the second question, we will look at Alan Turing's classic 1950 paper on the Turing Test of whether computers can think and at John Searle's 1980 Chinese Room Argument challenging that test.

Further Reading:
For more on the philosophy of AI, see McCarthy and Hayes 1969; Sloman 1971, 1978; Boden 1977, 1990b; Rapaport 1986d; Moody 1993; Akman 2000; Bringsjord and Govindarajulu 2018.

19.3 What Is AI?

19.3.1 Definitions and Goals of AI

Many definitions of AI have been proposed (see the "Further Reading" box at the end of this chapter). In this section, I want to focus on two nicely contrasting definitions. The first is by Marvin Minsky, one of the pioneers of AI research; the second is by Margaret Boden, one of the pioneers of cognitive science:

1. ... *artificial intelligence*, the science of making machines do things that would require intelligence if done by men.[3] (Minsky, 1968, p. v)

2. By "artificial intelligence" I ... mean the use of computer programs and programming techniques to cast light on the principles of intelligence in general and human thought in particular.[4] (Boden, 1977, p. 5)

Minsky's definition suggests that the methodology of AI is to study *humans* in order to learn how to program *computers*. (Note that this was Turing's methodology in his 1936 paper; see §8.8.2.8.3, above.) Boden's definition suggests a methodology that goes in the opposite direction: to study *computers* in order to learn something about *humans*. AI is, in fact, a two-way street: Minsky's view of AI as moving from humans to computers and Boden's view of it as moving from computers to humans are both valid.

Both views are also consistent with Stuart C. Shapiro's three goals of AI (Shapiro 1992a; see also Rapaport 1998, 2000a, 2003):

1. **AI as advanced CS or engineering:**

   > One goal of AI is to extend the frontiers of what we know how to program (in order to reach an ultimate goal of computers that are self-programming and that understand natural language) and to do

[3] That is, by humans.

[4] This is just one sentence from a lengthy discussion titled "What Is Artificial Intelligence?" (Boden, 1977, Ch. 1).

this *by whatever means will do the job*, not necessarily in a "cognitive" fashion. The computer scientist John Case once told me that AI understood in this way is at the "cutting edge" of CS.[5]

2. **AI as computational psychology:**

   Another goal of AI is to write programs as theories or models of *human cognitive behavior*. (Recall our discussion in Chapter 15 of computer programs as theories.)

3. **AI as computational philosophy:**

   Shapiro's third goal of AI is to investigate *whether cognition in general* (and not restricted to *human* cognitive behavior) *is computable*, that is, whether it is (expressible as) one or more recursive functions. (If cognition requires more than one recursive function, then, presumably, they will be *interacting* functions, as discussed in §11.4.3.)

19.3.2 Artificial Intelligence as Computational Cognition

In line with the question "What can be computed?" (see §3.15.2.1.1), AI can be understood as the branch of CS that investigates the extent to which *cognition* is computable. But the term 'artificial intelligence'—coined by John McCarthy in 1955—is somewhat of a misnomer. First, outside of AI, 'intelligence' is often used in the sense of IQ, but AI is not necessarily concerned only with finding programs with high IQ.

Further Reading:
But see my "Artificial I.Q. Test" at http://www.cse.buffalo.edu/~rapaport/AIQ/aiq.html (Rapaport, 1986d), as well as Ohlsson et al. 2015.

Echoing Minsky's definition and Shapiro's goals, Herbert Simon said this:

> The basic strategy of AI has always been to seek out progressively more complex human tasks and show how computers can do them, in humanoid ways or by brute force. With a half-century of steady progress, we have assembled a solid body of tested theory on the processes of human thinking and the ways to simulate and supplement them. (Quoted in Hearst and Hirsh 2000, p. 8.)

The phrase 'human tasks' nicely avoids any issues involved with the notion of "intelligence". But an even more general and accurate term would be 'cognition', which includes such mental states and processes as belief, consciousness, emotion, language, learning, memory, perception, planning, problem solving, reasoning, representation (including categories, concepts, and mental imagery), sensation, thought, etc.

[5] Another computer scientist, Anthony S. Ralston, agreed with Case's topological metaphor, except that instead of describing AI as being at the *cutting* edge, he told me that it was at the "periphery" of CS!

Second, 'artificial' carries the suggestion that "artificial" entities aren't the real thing. (Recall our discussion in §15.3.1.2.) 'Synthetic' is better than 'artificial', because an artificial diamond might not be a diamond—it might be a cubic zirconium—whereas a synthetic diamond *is* a real diamond that just happened to be formed in a non-natural way.

Further Reading:
Paton and Friedman 2018 observes that the distinction between synthetic and natural diamonds raises "an almost metaphysical question of what defines a diamond. Is it its chemical structure . . . ? Or is it its provenance: created deep in the ground . . . rather than cooked up in a machine?"

But an even better term is 'computational', which doesn't carry the stigma of "artificiality", and which specifies the nature of the "synthesis".

For these reasons, my preferred name for the field is 'computational cognition'. (Nevertheless, just as I use 'CS' in this book instead of "computer science", I will continue to use 'AI' instead of "computational cognition".) So, AI—understood as computational cognition—is the branch of CS (working with other disciplines, such as cognitive anthropology, linguistics, cognitive neuroscience, philosophy, and psychology, among others) that tries to answer the question: **How much of cognition is computable?** The *working assumption* of computational cognition is that ***all* of cognition is computable**: "The study [of AI] is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" (McCarthy et al., 1955).

And its main open research question is: **Are *aspects of cognition that are not yet known to be computable* computable?** If so, what does that tell us about the kinds of things that can produce cognitive behavior? (We'll investigate this when we look at the Turing Test in §19.4.) On the other hand, if there are non-computable aspects of cognition, *why* are they non-computable? And what would that tell us about cognition? (This is the conclusion of the Chinese Room Argument, to be discussed in §19.6.) An answer to this question should take the form of a logical argument such as the one that shows that the Halting Problem is non-computable (§7.7). It should not be of the form: "All computational methods tried so far have failed to produce this aspect of cognition". After all, there might be a new kind of method that has not yet been tried.

19.4 The Turing Test

> The Turing Test(?): A problem is computable if a computer can convince you it is.
> —Anonymous undergraduate student in the author's course, CSE 111, "Great Ideas in Computer Science" (14 December 2000)[6]

19.4.1 How Computers Can Think

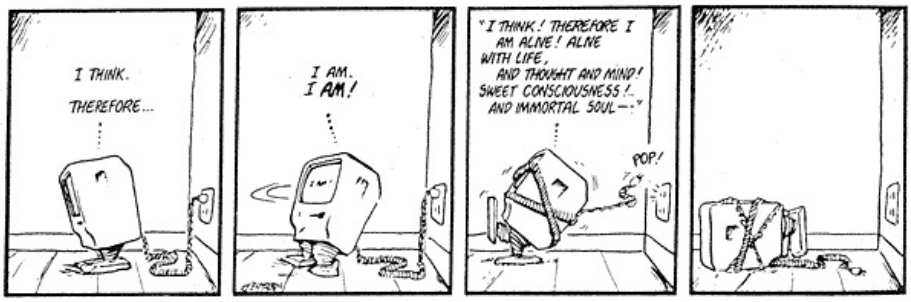


Figure 19.2: ©1-18-1985, Bloom County/Washington Post Writers Group

We have seen that AI holds that cognition is comput*able*. For our present purposes, it doesn't matter whether the computations are of the classical, symbolic variety or the connectionist, artificial-neural-network, or machine-learning variety. Nor does it matter whether the neuron firings that produce cognition in the human brain can be viewed as computations.

Further Reading: For further discussion of this, see Piccinini 2005, 2007a; Rapaport 2012b.

All that matters is this philosophical implication:

> If (and to the extent that) cognitive states and processes can be expressed as algorithms, then they can be implemented in non-human computers.

And this raises the following questions:

- Are computers executing such cognitive algorithms merely *simulating* cognitive states and processes?
- Or are they *actually exhibiting* them?

In popular parlance, do such computers think?

In this section, we will look at an answer to this question that arises from what is called the Turing Test. In §19.6, we will look at an objection to it in the form of the Chinese Room Argument. And after that, we will consider an interpretation of the situation that is based on the theory of syntactic semantics that was introduced in §17.8.2.

[6] http://www.cse.buffalo.edu/~rapaport/111F04.html. For a clarification of this remark, see the reference to Rey 2012 in the Historical Digression and Further Reading box at the end of §19.4.2.

19.4.2 The Imitation Game

> Recently, Michael Scherer, a *Time* magazine bureau chief, received a phone call from a young lady, Samantha West, asking him if he wanted a deal on health insurance. After she responded to a number of his queries in what sounded like prerecorded fashion, he asked her point-blank whether she was a robot, to which he got the reply "I am human." When he repeated the question, the connection was cut off. Samantha West turned out to be a system of recorded messages that were part of a computer program created by the brokers for health insurance.
> —Robert Skidelsky (2014, p. 36)

Just as Alan Turing's most important paper (Turing, 1936) never mentions a "Turing Machine", his second most important paper—"Computing Machinery and Intelligence" (Turing, 1950)—never mentions a "Turing Test". Instead, he introduces a parlor game that he calls the "Imitation Game". This is a game that you can actually play, not a mere thought experiment.

The Imitation Game consists of three players: A man, a woman, and an interrogator who might be either a man or a woman. It might matter whether the interrogator is a man rather than a woman, or the other way around, but we'll ignore this for now. For that matter, the interrogator could also be a computer, but there are good reasons why that should be ruled out: The point of the Turing Test is for a *human* to judge whether an entity can think (or whether its cognitive behavior is indistinguishable from that of a human).

The three players are placed in separate rooms, so that they cannot see each other, and they communicate only by means of what we would now call 'texting', so that they cannot hear each other. The reason that they are not allowed to see or hear each other is that the point of the game is for the interrogator to determine which room has the man, and which room has the woman. To make things interesting, the woman is supposed to tell the truth in order to convince the interrogator that she is the woman, but the man is supposed to convince the interrogator that he (the man) is the woman, so he will occasionally have to lie.

The man wins if he convinces (fools) the interrogator that he is the woman; the woman wins if she convinces the interrogator that she is the woman. (Another way of thinking of this is that the interrogator wins if he correctly figures out who is in which room.) If the man wins, then he is said to have passed the test.

Turing suggested that "an average interrogator will not have more than 70 per cent. chance of making the right identification after five minutes of questioning" (Turing, 1950, p. 442). But the actual amount of time may be irrelevant. One could conduct a series of imitation games and calculate appropriate statistics on how likely an interrogator is to make a correct determination after a given period of time.

What does this have to do with whether computers can think? What has come to be known as the Turing Test makes one small change in the Imitation Game:

> We now ask the question, "What will happen when a machine takes the part of [the man] in this game?" Will the interrogator decide wrongly as often when the game is played like this as he [or she] does when the game is played between a man and

> a woman? These questions replace our original, "Can machines think?"
> (Turing, 1950, p. 434, my interpolated brackets)

It turns out that there is some ambiguity in Turing's question: What is the "machine" (that is, the computer) supposed to do? Is it supposed to convince the interrogator that it is the *woman*? (That is, is it supposed to imitate a woman?) Or is it supposed to convince the interrogator that it is a *man who is trying to convince the interrogator that he is a woman*? (That is, is it supposed to imitate a man?)

Other modifications are possible. Usually, the Turing Test is taken, more simply and less ambiguously, to consist of a set up in which a computer, a human, and a human interrogator are located in three different rooms, communicating over a texting interface, and in which both the human and the computer are supposed to convince the interrogator that each is a *human*. To the extent that either the computer convinces the interrogator or the human fails to (under the same criteria for successful convincing that obtains in the original imitation game), the computer is said to have passed the Turing Test.

An even simpler version consists merely of two players: a human interrogator and someone or something (a human or a computer) in two separate, text-interfaced rooms. If a computer convinces the interrogator that it is a human, then it passes the Turing Test.

Further Reading:
The differences between these versions of the Turing Test are discussed by French 2000; Piccinini 2000; Rapaport 2006c. Argamon et al. 2003 provides evidence that women can be distinguished from men on the basis of their writing style.

Here is Turing's answer to the question that has now replaced "Can machines think?":

> I believe that at the end of the century [that is, by the year 2000] **the use of words** and **general educated opinion** will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.
> (Turing, 1950, p. 442, my interpolated brackets and boldface)

To see what this might mean, we need to consider the Turing Test a bit further.

Historical Digression and Further Reading:
The Turing Test was not the first test of its kind. In his *Discourse on the Method of Rightly Conducting the Reason and Seeking for Truth in the Sciences*, Descartes (1637, Part V, p. 116) proposed the following:

> ... if there were machines which bore a resemblance to our body and imitated our actions as far as it was morally possible to do so, we should always have two very certain tests by which to recognise that, for all that, they were not real men. The first is, that they could never use speech or other signs as we do when placing our thoughts on record for the benefit of others. For we can easily understand a machine's being constituted so that it can utter words ...; for instance, ... it may ask what we wish to say to it; ... it may exclaim that it is being hurt, and so on. But it never happens that it arranges its speech in various ways, in order to reply appropriately to everything that may be said in its presence, as even the lowest type of man can do. And the second ... is, that although machines can perform certain things as well as or perhaps better than any of us can do, they infallibly fall short in others, by the which means we may discover that they did not act from knowledge, but only from the disposition of their organs.

Encyclopedia articles on the Turing Test include Rapaport 2006c; Oppy and Dowe 2019. Several anthologies of essays on the Turing Test have appeared: Akman and Blackburn 2000; Moor 2003; Shieber 2004 (the latter reviewed in Rapaport 2005d), and Marcus et al. 2016.

Wilkes 1953 contains speculations by one of the pioneers of computers on the Turing Test, learning machines, and the role of external input.

Piccinini 2003 is primarily about Turing's views on AI, but also discusses his theory of computation and the role of "oracle" machines.

Aaronson 2006 discusses the Turing Test in the context of quantum hypercomputation.

Rey 2012 distinguishes between the (Church-)Turing (computability) thesis and the Turing Test (something that the student in my course that I quoted in the epigraph to §19.4 wasn't clear on!).

McDermott 2014 is a critique of Turing 1950 written by a well-known AI researcher as background for the film *The Imitation Game*.

Kahn 2014 and Hill 2016a raise the question of whether some robots and softbots may have passed a kind of Turing Test. Wu 2017 argues that "Automated processes should be required to state, 'I am a robot.' When dealing with a fake human, it would be nice to know." Parnas 2017 makes a similar plea, which we'll come back to at the end of this chapter.

Shieber 2007 proposes an interpretation of the Turing Test as an interactive proof.

Digression and Further Reading: On "the end of the century":
"The end of the century" (that is, the 20th century) has come and gone without Turing's expectations realized. (If they had been, we would not still be discussing them!) There have been several programs that are thought by some people to have passed a Turing Test: "Eliza" (Weizenbaum, 1966, 1967, 1976) was a natural-language *processing* program designed to show how the test could apparently be passed with no real *understanding* of natural language. "Parry" (Colby et al., 1971, 1972; Colby, 1981) was an Eliza-based program designed to simulate paranoia. The Loebner Prize competitions (Loebner 1994; Rees et al. 1994; https://web.archive.org/web/20040211063400/http://www.loebner.net/Prizef/loebner-prize.html, https://www.aisb.org.uk/events/loebner-prize)—originally intended to be a real Turing Test—have devolved into competitions for Eliza-like "chatterbots" (Shieber, 1994a,b). No current natural-language-understanding computer has achieved the "understanding" exhibited, for example, by the fictional computer HAL in the movie *2001* (Stork, 1997), not even Apple's Siri or Amazon's Alexa.

Similar predictions have also been off the mark. Simon and Newell 1958 predicted 1967 for the chess version of a Turing Test, missing by 30 years. (IBM's Deep Blue beat human chess champion Garry Kasparov in 1997; https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov.) However, Simon (personal communication, 24 September 1998, https://cse.buffalo.edu/~rapaport/simon.html) said that "it had nothing to do with the Turing Test" and that "(a) I regard the predictions as a highly successful exercise in futurology, and (b) placed in the equivalent position today, I would make them again, and for the same reasons. (Some people never seem to learn.)" At the end of the next millennium, no doubt, historians looking back will find the 40-year distance between the time of Newell and Simon's prediction and the time of Kasparov's defeat to have been insignificant.

19.4.3 Thinking vs. "Thinking"

> Lots of parts of a computer "think" in different ways, but . . . [the CPU] is what we usually call the "thinking" part. It's a machine for quickly following a set of steps that are written down as numbers. *Following steps might not be "thinking." But it's hard to say for sure.* That's one of those things where not only do we not know the answer, we're not sure what the question is.
> —Randall Munroe (2015, p. 37, my italics and interpolated brackets)

In 1993, *The New Yorker* magazine published a cartoon by Peter Steiner, showing a dog sitting in front of a computer talking to another dog, the first one saying, "On the Internet, nobody knows you're a dog." (See Figure 19.3.) This cartoon's humor arises from the fact that you do *not* know with whom you are communicating via computer! It's unlikely that there's a dog typing away at the other end of a texting session or an email, but could it be a computer pretending to be a human, as in the Turing Test? Or could it be a 30-year-old pedophile pretending to be a 13-year-old classmate?

(In the years since that cartoon appeared, we have become only too aware of the possibilities and dangers—political and otherwise—of messages and "fake news" on Facebook and elsewhere that purport to come from one source but really come from another, as well as the possibilities and dangers of the lack of privacy. A newer "internet dog" cartoon plays on this; see Figure 19.4.)

"On the Internet, nobody knows you're a dog."

Figure 19.3: https://condenaststore.com/featured/on-the-internet-peter-steiner.html, ©1993 The New Yorker Collection/Peter Steiner

"Remember when, on the Internet, nobody knew who you were?"

Figure 19.4: https://condenaststore.com/featured/two-dogs-speak-as-their-owner-uses-the-computer-kaamran-hafeez.html, ©2015 The New Yorker Collection/Kaamran Hafeez

Normally, we *assume* that we are talking to people who really are whom they say they are. In particular, we assume that we are talking to a human. But really all we know is that we are talking to *an entity with human cognitive capacities.* And that, I think, is precisely Turing's point: An entity with human cognitive capacities is all that we can ever be sure of, whether that entity is really a human or "merely" a computer.

This is a version of what philosophers have called "the argument from analogy for the existence of other minds". An *argument from analogy* is an argument of the form:

1. Entity A is like (that is, is analogous to) entity B with respect to important properties $P_1, \ldots, P_n$.

2. B has another property, Q.

3. $\therefore$ (Probably) A also has property Q.

(Compare the "duck test": "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck" (James Whitcomb Riley, https://en.wikipedia.org/wiki/Duck_test.) Such an argument is not *deductively* valid: It's quite possible for the premises to be true but for the conclusion to be false. But it has some *inductive* strength: The more alike two objects are in many respects, the more likely it is that they will be alike in many other respects (and maybe even all respects).

The *problem of the existence of other minds* is this: I know that *I* have a mind (because I know what it means for me to think, to perceive, to solve problems, etc.). How do I know whether *you* have a mind? Maybe you don't; maybe you're just some kind of computer, or android, or philosophical zombie.

Digression and Further Reading: Androids and Zombies:
Androids are robots that look like humans, such as Commander Data in *Star Trek: The Next Generation* or many of the characters in such science fiction as Dick 1968 or the film *Blade Runner*. A philosophical zombie is not a horror-movie zombie. Rather, it is an entity who is exactly like us in all respects but who lacks a mind or consciousness. See Kirk 1974; Chalmers 1996a; and other references at http://www.cse.buffalo.edu/∼rapaport/719/csnessrdgs.html#zombies

Putting these together, here is the argument from analogy for the existence of other minds:

1. You are like me with respect to all of our physical and behavioral properties.

2. I have a mind.
   (Or: My behavioral properties can best be explained by the fact that I have a mind.)

3. $\therefore$ (Probably) you have a mind.
   (Or: Your behavioral properties can best be explained if it is assumed that you also have a mind.)

Of course, this argument is deductively invalid. I could be wrong about whether you are biologically human. In that case, the best explanation of your behavior might not be

that you have a mind, but that you are a computer who has been cleverly and suitably programmed. Now, there are two ways to understand this: One way to understand it is to say that you *don't have a mind*; you're just a cleverly programmed robot. But another way to understand it is to say that being cleverly programmed in that way is exactly *what it means to have a mind*: Perhaps we are both cleverly programmed in that way. Or perhaps (a) *you* are *programmed* in that way, whereas (b) *I* have a *brain* that behaves in that way, but (c) these are simply two different *implementations* of "having a mind".

In either case, am I wrong about your being able to think? That is, am I wrong about your (human) cognitive abilities? Turing's answer is: No! More cautiously, perhaps, his answer is that whether I'm wrong depends on the definition of (human) cognitive abilities (or thinking):

If human-like cognition requires a (human) brain, and you lack one, then, technically speaking, you don't have human-like cognition (even if you pass the Turing Test). On this view, *I* really do think, but *you* can only "think". That is, you are not really thinking, but doing something else that can be called "thinking" only in a metaphorical sense.

But, if human-like cognition is an abstraction that can be implemented in different ways—that is, if it does not require a (human) brain—then we both have human-like cognition (and that's why you pass the Test). On this view, we both can think.

Here's an analogy: Everyone can agree that birds fly.

Further Reading:
"Birds fly" is true in general, even though most birds actually don't fly! Not only do penguins, ostriches, etc., not fly, but baby birds, birds with injured wings, dead birds, etc., also don't fly. Handling the logic of statements like this is a branch of logic and AI called "non-monotonic reasoning"; see §2.6.1.4, above, and Ginsberg 1987; Strasser and Antonelli 2015.

Do *people* fly? Well, we certainly speak as if they do; we say things like, "I flew from Buffalo to JFK last week." But we also know that I don't literally mean that I flapped my arms when flying from Buffalo to JFK; rather, I flew in an airplane: It wasn't I who was flying; it was the airplane. But that answer raises another question: Do *planes* fly? Well, they don't flap their wings, either! (See Figure 19.5.) So, in what sense are they flying?

There are two ways to understand what it means to say that planes fly: One way is by what I will call "metaphorical extension". The reason we say that planes fly is that what they are doing is very much like what birds do when they fly—they move through the air, even if their methods of doing so are different. But, instead of using a *simile*, saying that planes move through the air *like* birds fly, we use a *metaphor*, saying directly that planes fly. And then that metaphor becomes "frozen"; it becomes a legitimate part of our language, so much so that we no longer realize that it is metaphorical. This is just like what happens when we say that time is money: We say things like, "You're *wasting* time", "This will *save* you time", "How did you *spend* your vacation?", and so on. But we're usually not aware that we are speaking metaphorically (until someone points it out), and there's often no other (convenient) way to express the same ideas (Lakoff and Johnson, 1980a,b).

Figure 19.5: https://www.comicskingdom.com/family-circus/2019-07-12
©2019 Bil Keane, Inc.

As Turing said, "the use of words" has changed!

The other way to understand what it means to say that planes fly is that we have realized that flapping wings is not essential to flying. There are deeper similarities between what birds and planes do when they move through the air that have nothing to do with wing-flapping but that have everything to do with the *shape* of wings and, more generally, with the physics of flight. We have developed a more abstract, and therefore more general, theory of flight, one that applies to both birds and planes. And so we can "promote" the verb 'to fly' from its use solely for birds (and other flying animals) to a more general use that also applies to planes. To use the language of §14.2.4, the abstract notion of flying can be implemented in both biological and non-biological media.

As Turing said, "general educated opinion" has changed!

In fact, *both* the use of words *and* general educated opinion have changed. Perhaps the change in one facilitated the change in the other; perhaps the abstract, general theory *can account for* the metaphorical extension.

The same thing has happened with 'computer'. As we saw in §6.2, a computer was originally a *human* who computed. That was the case till about the 1950s, but, a half-century later, we now say that a computer is a *machine*. Before around 1950, what we now call 'computers' had to be called 'digital' or 'electronic computers' to distinguish them from the human kind. But, now, it is very confusing to read pre-1950 papers without thinking of the word 'computer' as meaning, by default, a non-human machine. (Recall the puzzling statement in Turing 1936, p. 250: "The behaviour of the *computer* at any moment is determined by the symbols which *he* is observing, and *his* 'state of mind' at that moment"; see §8.8.2.2, above.) Now, at the beginning of the 21st century, general educated opinion holds that computers are best viewed abstractly,

in functional, input-output terms. The study of "artificial intelligence" may lead us to understanding thinking as an abstraction that can be implemented in both humans and computers, just as the study of "artificial" flight (https://invention.psychology.msstate.edu/library/Magazines/Nat_Artificial.html) was crucial to understanding flying as an abstraction implementable in both birds and planes: "[S]tudying the animals that fly", no matter in how great detail and for how many years, would not have yielded any useful information on how humans might be able to fly. Rather,

> attempt[ing] to construct devices that fly … attempts to build flying machines [resulted in] our entire understanding of flight today. Even if one's aim is to understand how birds or insects fly, one will look to aeronautics for the key principles …. (Quillian, 1994, pp. 440–442, my interpolated brackets)[7]

This is consistent with Boden's view that the study of computational theories of cognition can help us understand human (and, more generally, non-human) cognition.

What does this have to do with the philosophy of AI? The strategy of abstracting from a naturally occurring example and re-implementing it computationally also applies to cognition:

> Quite typically, an abstract structure underlies some human cognitive activity that is not at all apparent in superficial phenomenology or practice. Often, that structure is related in interesting ways to the structures we would invent if we constructed an ideal machine to perform that cognitive activity. (We might think of artificial intelligence as a normative enterprise). But that structure is rarely identical to the ideal machine's structure. (Gopnik, 1996, p. 489)

The underying abstract structure could be computational in nature. Hence, it could be (re-)implemented in "an ideal machine". The abstract computational theory might be thought of as having the form: Such-and-such a human cognitive activity can or ought to be performed in this computational way *even if* the way that humans in fact do it is not identical to that ideal structure. Gopnik goes on to say:

> This process may seem like analogy or metaphor, but it involves more serious conceptual changes. It is not simply that the new idea is the old idea applied to a new domain, but that the earlier idea is itself modified to fit its role in the new theory. (Gopnik, 1996, p. 498)

To say, as I did earlier, that the process is metaphorical is not inconsistent with it also involving "more serious conceptual changes": We come to see the old idea in a new way.

But some philosophers argue that what AI computers do is not *really* thinking. We'll turn to one of these philosophers in §19.6.

[7] Quillian—a pioneer in AI research—uses this argument to support an explanation of why the natural sciences are more "effective" than the social sciences.

Further Reading:
Applying human terms like 'thinking' to non-human entities—whether non-human animals or robots—is called 'anthropomorphism', and is sometimes frowned upon by scientists. An opposing view by a primatologist is presented in de Waal 2016. Matthews and Dresner 2017, especially §5, pp. 19–20—cited in §9.5.4 in connection with Searle's arguments about the nature of computers—is also relevant to the thinking-vs.-"thinking" issue.

19.5 Two Digressions

19.5.1 The "Lovelace Objection"

Turing (1950, §6) considered several objections to the possibility of AI, one of which he called "Lady Lovelace's Objection". Here it is in full, in Lovelace's own words:

> The Analytical Engine has no pretensions whatever to *originate* anything. **It can do whatever we *know how to order it* to perform.** It can *follow* analysis; but it has no power of *anticipating* any analytical relations or truths. Its province is to assist us in making *available* what we are already acquainted with.
> (Menabrea and Lovelace, 1843, p. 722, italics in original, my boldface; https://psychclassics.yorku.ca/Lovelace/lovelace.htm#G)

The first thing to note about this is that it is often misquoted. We saw Herbert Simon do this in §11.4.3.4.1, above, when he expressed it in the form "computers can *only* do what you program them to do" (Simon, 1977, p. 1187, my italics). Lovelace did not use the word "only". We'll see one reason why in a moment. But note that this standard interpretation of her phrasing does seem to be what Turing had in mind. He quotes with approval Hartree 1949, p. 70—the same book that we saw Arthur Samuel quoting in §9.2, by the way—who said, concerning Lovelace's comment, "This does not imply that it may not be possible to construct electronic equipment which will 'think for itself' ...." Minus the double negative, Hartree (and Turing) are saying that Lovelace's comment is consistent with the possibility of an AI computer passing the Turing Test. Turing goes on to say this:

> A better variant of the objection says that a machine can never "take us by surprise". This statement is a more direct challenge and can be met directly. Machines take me by surprise with great frequency. (Turing, 1950, p. 450)

Turing's dryly humorous response has been elaborated on by Darren Abramson (2014, his italics, my bracketed interpolation), who says that Turing's 1936 paper shows

> that the concepts of determinism and predictability fall apart. Computers, which can be understood as the finite unfoldings of a particular Turing machine, are completely deterministic. But there is no definite procedure for figuring out, in every case, what they'll do [because of the Halting Problem]: if you could, then you would have a definite procedure for deciding whether any statement of arithmetic is true or not. But there is no such procedure for the one, so there is no such procedure for the other. *Computers are, in the general case, unpredictable, even by someone who knows exactly how they work.*

Returning to what Lovelace actually said, it's worth observing that the fact that the Analytical Engine (or any contemporary computer, for that matter) has no "pretensions" simply means that it wasn't designed that way; nevertheless, it might still be able to "originate" things. Also, if we can find out "how to order it to perform" cognitive activities, then it can do them! Finding out how requires us to be conscious of something that we ordinarily do unconsciously. In his own commentary on the Lovelace objection, Samuel (1953, p. 1225, my interpolated brackets) said:

> Regardless of what one calls the work of a digital computer [specifically, regardless of whether one says that it can think], the unfortunate fact remains that more rather than less human thinking is required to solve a problem using a present day machine since every possible contingency which might arise during the course of the computation must be thought through in advance. The jocular advice recently published to the effect, "Don't Think! Let UNIVAC do it for you," cannot be taken seriously. Perhaps, if IBM's familiar motto [namely, "Think!"] needs amending, it should be "Think: Think harder when you use the 'ULTIMAC'.

(Samuel adds in a footnote to this passage that 'ULTIMAC' is "A coined term for the 'Ultimate in Automatic Computers.' The reader may, if he prefers, insert any name he likes selected from the following partial list of existing machines . . . ", and he then listed 43 of them, including Edvac, IBM 701, Illiac, Johnniac, and Univac.)

Why didn't Lovelace use the word 'only'? Recall from §6.5.3 that Babbage, inspired by de Prony, wanted his machines to replace human computers:

> . . . Babbage deplored the waste of brilliant, educated men in routine, boring drudgery, for which he claimed the uneducated were better suited . . . . When convenient, however, he saw no obstacle to replacing them by yet more accurate or efficient machinery (he disapproved of unions). (Stein, 1984, pp. 51–52)

It is in this context that Lovelace "rephrased Babbage's words of assurance for the men of Prony's first section" (p. 52) (these were to be "the most eminent mathematicians in France, charged with deciding which formulae would be best for use in the step-by-step calculation of the functions to be tabulated. (They performed the programmer's task.)" (p. 51). These "eminent" mathematical "men of the first section"—and they *were* men—needed to be assured that the drudge work could be handled by a machine, hence Lovelace's words: "The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform." This puts a positive spin on a sentence that has typically been understood negatively: The computer *can* do whatever we can program it to do (and not: The computer can "only" do whatever we program it to do).

Further Reading:
Simon 1966 contains, among other things, a counterargument to the claim that computers cannot think because they "do only what they are programmed to do" (p. 18). A sequel to that essay is Simon's "Scientific Discovery and the Psychology of Problem Solving", in the same volume, pp. 22–40, in which he argues that "scientific discovery is a form of problem solving", and hence computational in nature.

Stuart C. Shapiro 1995 presents brief refutations of various anti-AI arguments.

19.5.2 Turing on Intelligent Machinery

Turing 1950 was not Turing's only essay on AI. In an essay written the next year, Turing seems to come out a bit more strongly about the possibility of computers thinking:

> 'You cannot make a machine to think for you.' This is . . . usually accepted without question. It will be the purpose of this paper to question it. (Turing, 1951, p. 256)

Although it is possible to read that last sentence neutrally, to my ears it sounds like a challenge strongly suggesting that Turing thinks that you *can* make a machine think. Indeed, later he says that his "contention is that machines can be constructed which will simulate the behaviour of the human mind very closely" (p. 257). This is cautiously worded—is *simulation* of thinking

(that is, simulation of "the behavior of the human mind") the same as "real" thinking?—but his ultimate claim here is that it will come so close to human thinking as to make no difference: "on the whole the output of them [that is, of such "thinking" machines] will be worth attention *to the same sort of extent as the output of a human mind*" (p. 257, my italics and interpolated brackets). And how would this be proved? By the Turing Test: "It would be the actual reaction of the machine to circumstances that would prove my contention, if indeed it can be proved at all" (p. 257).

Turing also suggests that the algorithm for such a machine must be based on what is now called 'machine learning': "If the machine were able in some way to 'learn by experience' it would be much more impressive" (p. 257). (It is also worth pointing out that not everyone thinks that machine learning is "really" learning (Bringsjord et al., 2018).) Moreover, he also suggests that the machine should be an oracle machine (recall our discussion of these in §11.4.4):

> There is ... one feature that I would like to suggest should be incorporated in the machines, and that is a 'random element'. Each machine should be supplied with a tape bearing a random series of figures, *e.g.*, 0 and 1 in equal quantities, and this series of figures should be used in the choices made by the machine. (p. 259)

Note, however, that Turing seems to consider these to be a (small) extension of Turing Machines.

Also interesting is his anticipation of what is now called "The Singularity" (see §11.4.5.2, above), and the question that we will return to in Chapter 20 about whether we *should* build artificial intelligences:

> Let us now assume, for the sake of argument, that these machines are a genuine possibility, and look at the consequences of constructing them. To do so would of course meet with great opposition, unless we have advanced greatly in religious toleration from the days of Galileo. There would be great opposition from the intellectuals who were afraid of being put out of a job. ... it seems probable that once the machine thinking method had started, *it would not take long to outstrip our feeble powers*. There would be no question of the machines dying, and they would be able to converse with each other to sharpen their wits. At some stage therefore we should have to expect the machines to take control ....
> (pp. 259–260, my italics)

Further Reading:
For an earlier essay by Turing on the nature of "a machine with intelligence", see Turing 1947.

19.6 The Chinese Room Argument

> [R]esearchers ... tracked ... unresponsive patients ..., taking EEG recordings .... During each EEG recording, the researchers gave the patients instructions through headphones. ... "Somewhat to our surprise, we found that about 15 percent of patients who were not responding at all had ... brain activation in response to the commands," said Dr. Jan Claassen .... "It suggests that there's some remnant of consciousness there. *However, we don't know if the patients really understood what we were saying. We only know the brain reacted.*"
> —Benedict Carey (2019, my italics)

Thirty years after Turing's publication of the Turing Test, John Searle published a thought experiment called the Chinese Room Argument (Searle, 1980, 1982, 1984).

In this experiment, a human who knows no Chinese (John Searle himself, as it happens) is placed in a room (the "Chinese room") along with paper, pencils, and a book containing an English-language algorithm for manipulating certain "squiggles" (marks or symbols that are meaningless to Searle-in-the-room).

Terminological Digression:
I distinguish between (1) the real John Searle who is a philosopher and author of Searle 1980 and (2) the Chineseless "John Searle" who occupies the Chinese room. I refer to the former as 'Searle' and to the latter as 'Searle-in-the-room'.

Outside the room is a native speaker of Chinese. There is something like a mail slot in one wall of the Chinese room. Through that slot, the native speaker inputs pieces of paper that contain a text written in Chinese along with reading-comprehension questions about that text, also in Chinese. When Searle-in-the-room gets these pieces of paper—which, from his point of view, contain nothing but meaningless squiggles—he consults his book and follows its instructions. Those instructions tell him to manipulate the symbols in certain ways, to write certain symbols down on a clean piece of paper, and to output those "responses" through the mail slot. The native speaker who reads them determines that whoever (or whatever) is in the room has answered all the questions correctly in Chinese, demonstrating a fluent understanding of Chinese. This is because the rule book of instructions is a complete natural-language-understanding algorithm for Chinese. But, by hypothesis, Searle-in-the-room does not understand Chinese. We seem to have a contradiction.

The Chinese Room Argument (CRA) is offered as a counterexample to the Turing Test, concluding from this thought experiment that it is possible to pass a Turing Test, yet not really think. The setup of the CRA is identical to the simplified, two-player version of the Turing test: The interrogator is the native Chinese speaker, who has to decide whether the entity in the room understands Chinese. The interrogator determines that the entity in the room *does* understand Chinese. This is analogous to deciding that the entity in the simplified Turing Test is a human, rather than a computer. But the entity in fact does *not* understand Chinese. This is analogous to the entity in the simplified Turing Test being a computer. So, Searle-in-the-room passes the Turing Test without being able to "really" understand; hence, the test fails.

Or does it?

Further Reading: Historical Antecedents:
Although Searle's CRA is the most famous version of this kind of set-up, there are earlier ones. In 1959, the logician Hartley Rogers, Jr., wrote:

> Consider a box B inside of which we have a man L with a desk, pencils and paper. On one side B has two slots, marked *input* and *output*. If we write a number on paper and pass it through the input slot, L takes it and begins performing certain computations. If and when he finishes, he writes down a number obtained from the computation and passes it back to us through the output slot. Assume further that L has with him explicit deterministic instructions of finite length as to how the computation is to be done. We refer to these instructions as P. Finally, assume that the supply of paper is inexhaustible, and that B can be enlarged in size so that an arbitrarily large amount of paper work can be stored in it in the course of any single computation. . . . I think we had better assume, too, that L himself is inexhaustible, since we do not care how long it takes for an output to appear, provided that it does eventually appear after a finite amount of computation. We refer to the system B-L-P as M. . . . In the approach of Turing, the symbolism and specifications are such that the entire B-L-P system can be viewed as a digital computer . . . . Roughly, to use modern computing terms, L becomes the logical component of the computer, and P becomes its program. In Turing's approach, the entire system M is hence called a *Turing machine*. (Rogers, 1959, pp. 115, 117)

An even earlier version was in a 1954 episode of *I Love Lucy*, which we'll discuss in §19.6.3.4. And Weinberg 2019 argued that the science fiction story "The Game" (Mickevich, 1961) anticipated the CRA, although it is actually closer to the philosopher Ned Block's (1978) "Chinese nation" thought experiment.

Further Reading on the CRA:
Too much has been written on the CRA to cite here, but you might start with these: Hauser 2001 and Cole 2019 are surveys from two online philosophical encyclopedias. Rapaport 1986c is an overview written for a semi-popular computer magazine, and Rapaport 1988b is a review of Searle 1984. Preston and Bishop 2002 is an anthology of responses to Searle (reviewed in Rapaport 2006b). In 2014, an online series of articles providing background for the movie *The Imitation Game* contained two critiques of the CRA: Cole 2014; Horst 2014.

19.6.1 Two Chinese Room Arguments

Searle actually bases two arguments on the Chinese Room thought experiment:

The Argument from Biology:

B1 Computer programs are non-biological.

B2 Cognition is biological.

B3 $\therefore$ No (non-biological) computer program can exhibit (biological) cognition.

The Argument from Semantics:

S1 Computer programs are purely syntactic.

S2 Cognition is semantic.

S3 Syntax alone is not sufficient for semantics.

S4 ∴ No (purely syntactic) computer program can exhibit (semantic) cognition.

The principal objection to the Argument from Biology is that premise B2 is at least misleading and probably false: Cognition can be characterized abstractly, and implemented in different media.

The principal objection to the Argument from Semantics is that premise S3 is false: Syntax—that is, symbol manipulation—does suffice for semantics.

After investigating these objections (and others), we will consider whether there are other approaches that can be taken to circumvent the CRA. One of them is to try to build a real analogue of a Chinese room; to do that, we will need to answer the question of what is needed for natural-language understanding.

19.6.2 The Argument from Biology

19.6.2.1 Causal Powers

Let's begin by considering some of the things that Searle says about the CRA, beginning with two claims that are versions of premise S1 of the Argument from Semantics:

> I [that is, Searle-in-the-room] still don't understand a word of Chinese and neither does any other digital computer because all the computer has is what I have: a formal program that attaches no meaning, interpretation, or content to any of the symbols. What this simple argument shows is that no formal program by itself is sufficient for understanding .... (Searle, 1982, p. 5, my interpolated brackets)

Note that this allows for the possibility that a program that *did* "attach" meaning, etc., to the symbols *might* understand. But Searle denies that, too:

> I see no reason in principle why we couldn't give a machine the capacity to understand English or Chinese, since in an important sense our bodies with our brains are precisely such machines. But ... we could not give such a thing to a machine where the operation of the machine is defined solely in terms of computational processes over formally defined elements .... (Searle, 1980, p. 422)

Why not? Because "only something that has the same causal powers as brains can have intentionality" (Searle, 1980, p. 423). By 'intentionality' here, Searle means "cognition" more generally. So he is saying that, if something exhibits cognition, then it must have "the same causal powers as brains".

All right; what are these causal powers? After all, if they turn out to be something that can be computationally implemented, then computers can have them (which Searle thinks they cannot). So, what does he say they are? He says that these causal powers are due to the fact that "I am a certain sort of organism with a certain biological (i.e.

chemical and physical) *structure*" (Searle, 1980, p. 422, my italics). That narrows down the nature of these causal powers a little bit. If we could figure out what this biological *structure* is, and if we could figure out how to implement that structure computationally, then we should be able to get computers to understand. Admittedly, those are big "if"s, but they are worth trying to satisfy.

So, what is this biological structure? Before we see what Searle says about it, let's think for a moment about what a "structure" is. What is the "structure" of the brain? One plausible answer is that the brain is a network of neurons, and the way those neurons are organized is its "structure". Presumably, if you made a model of the brain using string to model the neurons, then, if the strings were arranged in the same way that the neurons were, we could say that the model had the same "structure" as the brain. Of course, string is static (it doesn't do anything), and neurons are dynamic, so structure alone won't suffice, but it's a start.

Further Reading:
There have been other suggestions along these lines: Weizenbaum 1976, Ch. 2, considers a Turing Machine made of toilet paper and pebbles. Weizenbaum 1976, Ch. 5, considers computers "made of bailing wire, chewing gum, and adhesive tape". There is even a real computer made from Tinker Toys (https://www.computerhistory.org/collections/catalog/X39.81)! And recall our discussion in §8.9.1 of Hilbert's tables, chairs, and beer mugs.

However, Searle doesn't think that even structure *plus* the ability to do something is enough: He says that a simulated human brain "made entirely of ... millions (or billions) of old beer cans that are rigged up to levers and powered by windmills" would not really exhibit cognition even though it appeared to (Searle, 1982). Cognition must (also) be *biological*, according to Searle. That is, it must be made of the right stuff.

But now consider what Searle is saying: Only biological systems have the requisite causal properties to produce cognition. So we're back at our first question: What are those causal properties? According to Searle, they are the ones that are "causally capable of producing perception, action, understanding, learning, and other intentional [that is, cognitive] phenomena" (Searle, 1980, p. 422, my interplated brackets). Again: What are the causal properties that produce cognition? They are the ones that produce cognition! That's not a very helpful answer.

Elsewhere, Searle does say some things that give a possible clue as to what the causal powers are: "mental states are both *caused by* the operations of the brain and *realized in* the structure of the brain" (Searle, 1983, p. 265). In other words, they are *implemented* in the brain. And this suggests a way to avoid Searle's argument from biology.

19.6.2.2 The Implementation Counterargument

> [M]ental states are as real as any *other* biological phenomena, as real as lactation, photosynthesis, mitosis, or digestion. Like these other phenomena, mental states are caused by biological phenomena and in turn cause other biological phenomena. (Searle, 1983, p. 264, my italics)

Searle's "mental states" are biological *implementations*. But, if they are implementations, then they must be implementations of something else that is more abstract: abstract mental states. (This follows from §14.2.4's Implementation Principle I.)

Searle (1980, p. 451, my italics) says that "... intentional states ... are *both caused by and realized in* the *structure* of the brain." But brains and contraptions made from beer-cans + levers + windmills can *share* structure. This is a simple fact about the nature of *structure*. Therefore, what Searle said must be false: It can't be a *single* thing—an intentional (that is, mental) state—that is *both* caused by *and* realized in the brain. Rather, what the brain *causes* are *implemented* mental states, but what the brain *realizes* are *abstract* mental states, and the abstraction and its implementation are two distinct things.

In §14.2.2, we saw that abstractions can be implemented in more than one way—they can be "multiply realized". (This was §14.2.4's Implementation Principle II.) We saw that stacks can be implemented as arrays or as lists, that any sequence of items that satisfy Peano's axioms is an implementation of the natural numbers, that any two performances of the same play or music are different implementations of the script or score, and so on.

So, Searle says that the human brain can understand Chinese because understanding is biological, whereas a computer executing a Chinese natural-language-understanding program *cannot* understand Chinese, because it is *not* biological. But the implementation counterargument says that, on an abstract, functional, computational notion of understanding as an abstraction, understanding can be implemented in both human brains and in computers, and, therefore, both can understand.

More generally, if we put Implementation Principles I and II together, we can see that if we begin with an implementation (say, real, biological mental states), we can develop an abstract theory about them. This is what AI and computational cognitive science try to do, following Minsky's methodology. But once we have an abstract theory, we can re-implement it in a different medium. If our abstract theory is computable, then we can re-implement it in a computer. When this happens, our use of words changes, because general educated opinion changes, as Turing predicted. And this is consistent with Boden's methodology: We can learn something about human cognition by studying computer cognition.

Digression:
This transition from an implementation in one medium "up" to an abstraction, and then "down" to another implementation in a different medium is what happened with flying. We began with an implementation (birds and other animals that fly). We then developed an abstract theory (the physics of flight). And this was then re-implemented in the medium of airplanes.

It also happened with computers. We began with an implementation (humans who compute). Turing (1936) then developed an abstract theory (the mathematical theory of Turing Machine computation). And that was then re-implemented in electronic, digital computers.

19.6.3 The Argument from Semantics

Now let's turn to Searle's second argument, repeated here:

S1 Computer programs are purely syntactic.

S2 Cognition is semantic.

S3 Syntax alone is not sufficient for semantics.

S4 $\therefore$ No purely syntactic computer program can exhibit semantic cognition.

In this section, we will look at reasons for thinking that S3 is false, that syntax *does* suffice for semantics.

19.6.3.1 The Premises

First, let's consider this argument in a little bit more detail. Premise S1 says that computer programs merely tell a computer to (or describe how a computer can) manipulate symbols solely on the basis of their properties as marks and their relations among themselves. This manipulation is completely independent of the symbols' semantic relations, that is, of the relations that the symbols have to other items that are external to the computer. These external items are the meanings or interpretations of the symbols, the "aspects" of the real world that the symbols represent (Lewis, 1970, p. 19). Note, by the way, that insofar as a computer *did* manipulate its internal symbols in a way that was causally dependent on such external items, it could only do so by inputting an internal representative of that external item. But, in that case, it would still be directly manipulating only internal symbols, and only indirectly dealing with the external item.

Premise S2 says that cognition is centrally concerned with such "external" relations. Cognition, roughly speaking, is whatever the brain does with the sensory inputs from the external world. To fully understand cognition, according to this premise, it is necessary to understand the internal workings of the brain, the external world, *and* the relations between them. That is a semantic enterprise.

It seems clear that the study of relations among the symbols alone could not possibly suffice to account for the relations between those symbols and anything else. Hence premise S3: Syntax and semantics are two different, though related, subjects.

Conclusion S4 seems to follow validly. So, any questions about the goodness of the argument must concern its soundness: Are the premises true? Doubts have been raised about each of them.

19.6.3.2 Which Premise Is at Fault?

Let's look at S1 first: Although it is not a computer program, the World Wide Web is generally considered to be a syntactic object: a collection of nodes (for example, websites) related by links; that is, its mathematical structure is that of a graph. Some researchers have felt that there are limitations to this "syntactic" web and have proposed the Semantic Web. By "attaching meanings" to websites (as Searle might say), they hope to make the Web more ... well ... meaningful, more useful. In fact, however, the way they do this is by adding more syntax! So, for now, we'll accept premise S1. (For arguments that S1 is *false*, recall our discussion in §17.8.)

Further Reading: On the Semantic Web, see Berners-Lee et al. 2001. On its syntactic nature, see Rapaport 2012b, §3.2 and note 25.

Next, let's look at S2: Recall from the Digressions in §11.4.3.4.2 and 17.9 that at least one major philosopher, Jerry Fodor (1980), has argued that the study of cognition need *not* involve the study of the external world that is being cognized, on the grounds that cognition is what takes place internally to the brain. Whether the brain correctly or incorrectly interprets its input from the external world, it's the interpretation that matters, not the actual state of the external world. This view ("methodological solipsism") holds that, as a methodology for studying cognition, we can pretend that the external world doesn't exist; we only have to investigate what the brain does with the inputs that it gets, not where those inputs come from or what they are really like. (We'll return to this in §19.6.3.4.)

Of course, if understanding cognition only depends on the workings of the brain and not on its relations with the external world, then the study of cognition might be purely syntactic. And so we're ready to consider premise S3. Can we somehow get semantics from syntax? There are three, interrelated reasons for thinking that we can.

First, we can try to show that semantics, which is the study of relations *between* symbols and meanings, can be turned into a syntactic study, a study of relations *among* symbols and "symbolized" meanings (see §19.6.3.3, below). Second, we can take the methodologically solipsistic approach and argue that an internal, "narrow", first-person point of view is (all that is) needed for understanding or modeling cognition (see §19.6.3.4). Third, it can be argued that semantics is recursive in the sense that we understand a *syntactic* domain in terms of an antecedently understood *semantic* domain, but that there must be a base case, and that this base case is a case of syntactic understanding (see §19.6.3.5).

Before looking at each of these, remember that Searle claims that syntax cannot suffice for semantics because the former is missing the links to the external world. This kind of claim relies on two assumptions, both of which are faulty. First, Searle is assuming that computers have no links to the external world, that they are really (and not just methodologically) solipsistic. But this is obviously not true, and is certainly inconsistent with Brian Cantwell Smith's (1985) point that, even if computers only deal with an (internal) *model* of the real world, they have to *act* in the real world. (Recall our discussion in §17.3.)

Second, Searle assumes that external links are really needed in order to attach meanings to symbols. But, if so, then why couldn't computers have them just as well as humans do? Both humans and computers exist and act in the world. If we humans have the appropriate links, what reason (other than the faulty Argument from Biology) is there to think that computers could not?

19.6.3.3 Semiotics

The first reason for thinking that syntax might suffice for semantics comes from semiotics, the study of signs and symbols. According to one major semiotician, Charles Morris (1938), semiotics has three branches: syntax, semantics, and pragmatics.

Given a formal system of "marks" (symbols without meanings)—sometimes called a "(formal) symbol system"—*syntax* is the study of the properties of the marks and of the relations *among* them: how to recognize, identify, and construct them (in other words, what they look like, for instance, their grammar), and how to manipulate them (for instance, their proof theory). Importantly, syntax does not study any relations *between* the marks and anything else. (Recall our discussion in §14.3.2.1 of formal systems and our discussion in §17.9.2 of symbols, marks, and meanings.)

Semantics is the study of relations *between* the marks and their "meanings". Meanings are part of a different domain of semantic interpretations (recall our discussion of this in §14.3.2.3). Therefore, syntax cannot and does not suffice for semantics! (Or so it would seem.)

Pragmatics has been variously characterized as the study of relations among marks, meanings, *and* the cognitive agents that interpret them; or as the study of relations among marks, meanings, interpreters, *and contexts*. Some philosophers have suggested that pragmatics is the study of everything that is interesting about symbols systems that isn't covered under syntax or semantics!

For our purposes, we only need to consider syntax and semantics. Again, let's be clear. Syntax studies the properties of, and relations among, the elements of a single set of objects (which we are calling "marks"); for convenience, call this set SYN. Semantics studies the relations between the members of two sets: the set SYN of marks, and a set SEM of "meanings".

Now, take the set-theoretical union of these two sets—the set of marks and the set of meanings: SYNSEM = SYN $\cup$ SEM. Consider SYNSEM as a new set of marks. We have now "internalized" the previously external meanings into a new symbol system. And the study of the properties of, and the relations among, the members of SYNSEM is SYNSEM's syntax! In other words, what was formerly semantics (that is, relations *between* the marks in SYN and their meanings in SEM) is now syntax (that is, relations *among* the new marks in SYNSEM.) This is how syntax can suffice for semantics.

This can be made clearer with the diagram in Figure 19.6. The top picture of a set of marks ("SYNtactic DOMain") shows two of its members and a relation between them. Imagine that there are many members, each with several properties, and many with relations between them. The study of this set, its members, their properties, and the relations they have to each other is the syntax of SYN.

Now consider the middle picture: two sets, SYN and a set of "meanings" ("SEMantic DOMain"). SYN, of course, has its syntax. But so does SEM. (Often,

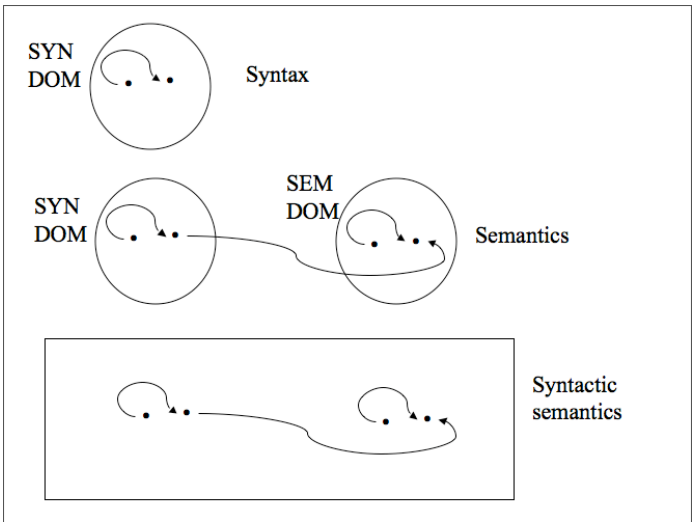


Figure 19.6: Syntax, semantics, and syntactic semantics

in AI, the syntax of a semantic domain is called its "ontology".) But now there are additional relations between (some or all of) the members of SYN and (some or all of) the members of SEM. Note that these relations are "external" to both domains: You really can't describe these relations using only the language used to describe SYN or only the language used to describe SEM. Instead, you need a language that can talk about *both* domains, and that language cannot be "internal" to either domain. The study of these relations is what is called "semantics". The usual idea is that the members of SEM are the "meanings" of the members of SYN, espcially if SYN is the language used to describe SEM. So, for instance, you might think of SEM as the actual world and SYN as either a language like English that talks about the actual world or a scientific theory about the actual world, perhaps expressed in some mathematical (or computational!) language. Another way to think about this is that SEM gives us the facilities needed to understand SYN: We understand SYN in terms of SEM.

In the bottom picture, we have taken the union of these two domains. Now, the formerly "external" semantic relations have become *internal* relations of the new, unioned domain. But, as such, they are now no different in principle from the previous internal, syntactic relations of SYN or the previous internal, syntactic (or ontological) relations of SEM. Thus, these previous *semantic* relations have also become *syntactic* ones. This is what we called "syntactic semantics" in §17.8.2.

Further Reading:
For more details on syntactic semantics, see Rapaport 1988a, 2012b, 2017a. Any study of semantics in linguistics—in the sense of a study of the meanings of linguistic expressions—that focuses only on relations either among the expressions or between the expressions and mental "conceptual structures"—and not on relations between expressions and aspects of the external world—is a syntactic enterprise. This is the nature of semantics in, for instance, cognitive linguistics (Lakoff, 1987; Langacker, 1999; Talmy, 2000; Jackendoff and Audring, 2018).

This way of viewing semantics as a kind of syntax raises a number of questions: *Can* the semantic domain be internalized? Yes, under the conditions obtaining for human language understanding: How *do* we learn the meaning of a word? How, for instance, do I learn that the word 'tree' means "tree"? A common view is that this relation is learned by associating real trees with the word 'tree'.

Digression:
Obviously, this is only the case for some words. Logical words ('the', 'and', etc.), words for abstract concepts ('love'), and words for things that don't exist ('unicorn') are learned by different means. And most children raised in large cities learn (somehow) the meanings of words like 'cow' or 'rabbit' from *pictures* of cows and rabbits, long before they see real ones!

But really what happens is that *my internal representation of* an actual *tree* in the external world is associated with *my internal representation of* the *word* 'tree'. Those internal representations could be certain sets of neuron firings. In whatever way that neurons are bound together when, for instance, we perceive a pink cube (perhaps with shape neurons firing simultaneously with, and thereby binding with, color neurons that are firing), the neurons that fire when we see a tree might bind with the neurons that fire when we are thinking of, or hearing, or reading the word 'tree'.

And the same thing can happen in a computational cognitive agent. Suppose we have such an agent (a robot, perhaps; call her 'Cassie') whose computational "mind" is implemented as a semantic network whose nodes represent concepts and whose arcs represent structural relations between concepts, as in Figure 19.7: There is a real tree external to Cassie's mind. Light reflecting off the tree enters Cassie's eyes; this is the causal link between the tree and Cassie's brain. The end result of the visual process is an internal representation of the tree in Cassie's brain. But she also has an internal representation of the word 'tree', and those two representations can be associated. What Cassie now has is an enlarged set of marks, including a mark for a word and a mark for the word's meaning. But they are both marks in her mind.

Further Reading:
For more about Cassie, see "A 'Conversation' with Cassie" at http://www.cse.buffalo.edu/~rapaport/675w/cassie.conversation.new.html, and Shapiro and Rapaport 1987, 1995; Rapaport 1988a, 2006a; Shapiro 1989, 1998.

This is akin to the Robot Reply to the CRA (Searle, 1980, p. 420), in which sensors and effectors are added to the Chinese Room so that Searle-in-the-room can both perceive the external world as well as act in it. Searle's response to the Robot Reply

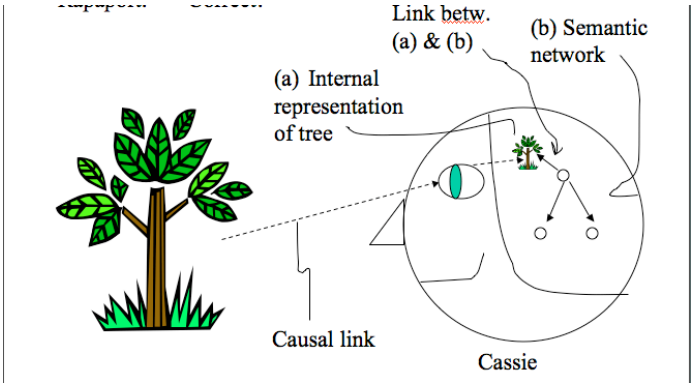


Figure 19.7: How a computational cognitive agent perceives the world

is to say that it is just more symbols. The reply to Searle is to say that that is exactly how *human* cognition works! In our brains, all cognition is the result of neuron firings. The study of that single set of neuron firings is a syntactic study, because it is the study of the properties of, and relations among, a single set of "marks"—in this case, the "marks" are neuron firings.

Digression and Further Reading:
Rescorla 2012a, p. 19, says, "neural properties are not multiply realizable. So neural properties are not syntactic properties." But he gives no argument for why they are not multiply realizable, nor is it clear why he thinks that they are not syntactic *as a consequence of this*. I agree that *neurons* are not multiply realizable, but surely their *properties* are. Properties, after all, are universals, and, if anything is multiply realizable, universals are.

The same is true for computers: If I say something to Cassie in English, she builds internal nodes that represent my utterance in her semantic network. If I show pictures to her, or if she sees something, she builds other internal nodes representing what she sees. This set of nodes forms a single computational knowledge base, whose study is syntactic in nature (because it is the study of the properties of, and relations among, a single set of "marks"—in this case, the "marks" are nodes in a semantic network). In the same way, both truth tables and the kind of formal semantics that logicians study are syntactic ways of doing semantics: The method of truth tables syntactically manipulates symbols that represent semantic truth values. And formal semantics syntactically manipulates symbols that represent the objects in the domain of semantic interpretation.[8]

[8] I owe this observation to Brian Cantwell Smith's panel presentation at the 1983 Cognitive Science Society conference at the University of Rochester (Hayes, 1983).

19.6.3.4 Points of View

The second prong of our reply to the Argument from Semantics concerns the differing points of view of the native, Chinese-speaking interrogator and Searle-in-the-room. To understand how a cognitive agent understands, and to construct a computational cognitive agent, we should take the first-person point of view. We should construct a cognitive agent (a robot, if you will) from the agent's point of view, from the perspective of what's going on "inside" the agent's head. In other words, we must be methodologically solipsistic and develop or implement a "narrow" or "internal" model of cognition. Such a model is called 'narrow', as opposed to 'wide', because it ignores the wider outside world and focuses only on the narrow inner world of the agent's point of view. We don't need to understand the causal or historical orgins of the agent's internal symbols; we only need to understand the symbols.

Further Reading:
For arguments why AI should take the first-person point of view, see Maida and Shapiro 1982, p. 296; Rapaport and Shapiro 1984; Rapaport 1986a; Wiebe and Rapaport 1986; Shapiro and Rapaport 1987, 1991; and Chalmers 2019, §2(1), p. 20.

But, in the CRA, there are two different points of view: There is Searle-in-the-room's point of view and there is the interrogator's point of view. In the CRA, Searle-in-the-room's point of view trumps the interrogator's; in the Turing Test (and in the kind of syntactic semantics that we are discussing), the interrogator's trumps Searle-in-the-room's. How should we resolve this?

Here is an analogy that I think helps clarify the situation. Consider the following passage from *The Wizard of Oz* (the novel, not the movie):

> When Boq [a Munchkin] saw her silver shoes, he said,
> "You must be a great sorceress."
> "Why?" asked [Dorothy].
> "Because you wear silver shoes and have killed the wicked witch. Besides, you have white in your frock, and only witches and sorceresses wear white."
> "My dress is blue and white checked," said Dorothy ....
> "It is kind of you to wear that," said Boq. "Blue is the color of the Munchkins, and white is the witch color; so we know you are a friendly witch."
> Dorothy did not know what to say to this, for all the people seemed to think her a witch, and she knew very well she was only an ordinary little girl who had come by the chance of a cyclone into a strange land.
> (Baum, 1900, pp. 34–35, my interpolated brackets)

Is Dorothy a witch? From her point of view, the answer is 'no'; from Boq's point of view, the answer is 'yes'. Whose point of view should trump the other's? Dorothy certainly believes that she's not a witch, at least as *she* understands the word 'witch' (you know—black hat, broomstick, Halloween, and all that). Now, it is certainly possible that Dorothy *is* such a witch while believing (mistakenly in that case) that she is *not* such a witch. So, what counts as being a witch (in these circumstances)? Note that

the dispute between Dorothy and Boq is *not* about whether Dorothy is "really" a witch in some context-independent sense. The dispute is about whether Dorothy is a witch *in Boq's sense*, from *Boq's* point of view. And, because Dorothy is in Oz, Boq's point of view trumps hers!

Now compare this to the Chinese room situation: Here, instead of asking whether Dorothy is a witch, we ask: Does Searle-in-the-room understand Chinese? From his point of view, the answer is 'no'; from the native Chinese speaker's point of view, the answer is 'yes'. Whose point of view should trump the other's? Searle-in-the-room certainly believes that he does not understand Chinese, at least as *he* understands 'understanding Chinese' (that is, in the way that you understand your native language as opposed to the way that you understand a foreign language that you may have (poorly) learned in school). Now, it is certainly possible that Searle-in-the-room *does* understand Chinese while believing (mistakenly, in that case) that he does *not* understand it. So, what counts as understanding Chinese (in these circumstances)? For the same reason as in the witch case, it must be the native Chinese speaker's point of view that trumps Searle-in-the-room's!

Of course, it would be perfectly reasonable for Searle-in-the-room to continue to insist that he doesn't understand Chinese. Compare Searle-in-the-room's situation to mine: I studied French in high school; spent a summer living with a French family in Vichy, France; spent a summer studying French (although mostly speaking English!) at the University of Aix-en-Provence; and have visited French friends in France many times. I believe that I understand about 80% of the French that I hear in a one-on-one conversation (considerably less if I'm hearing it on TV or radio) and can express myself the way that I want about 75% of the time (I have, however, been known to give directions to Parisian taxi drivers), but I always feel that I'm missing something. Should I believe my native French-speaking friends when they tell me that I am fluent in French? Searle would say 'no'.

But Searle-in-the-room isn't me. Searle-in-the-room can't insist that *he alone* doesn't understand Chinese and that, therefore, his point of view should trump the native, Chinese-speaking interrogator's. And this is because *Searle-in-the-room isn't alone*: Searle-in-the-room has the Chinese natural-language-processing rule book (even if he doesn't know that that's what it is). This is the core of what is known as the Systems Reply to the CRA (Searle, 1980, pp. 419–420), according to which it is the "system"—consisting of Searle-in-the-room *together with* the rule book—that understands Chinese. After all, it is not a computer's CPU that would understand Chinese (or do arithmetic, or do word-processing), but it is the system, or combination, consisting of *the CPU executing a computer program* that would understand Chinese (or do arithmetic, or process words). Compare: It is not a Universal Turing Machine by itself that can do arithmetic, but a Universal Turing Machine *together with a program stored on its tape for doing arithmetic* that can do arithmetic. And Searle-in-the-room together with the rule book, stranded on a desert island, *could* communicate (fluently) with a native, Chinese-speaking "Friday".[9]

Does it make sense for a "system" like this to exhibit cognition? Doesn't cognition have to be something exhibited by a single entity, like a person, an animal, or a robot?

[9] 'Friday' was the name of a resident of the island that Robinson Crusoe was stranded on in Defoe 1719.

But recall Hutchins's theory of distributed cognition (§18.7). His example of a ship's crew together with their navigation instruments that navigates a ship is a real-life counterpart of Searle-in-the-room together with his rule book: "Cognitive science normally takes the individual agent as its unit of analysis. ... [But] systems that are larger than an individual may have cognitive properties in their own right that cannot be reduced to the cognitive properties of individual persons" (Hutchins, 1995b, pp. 265–266). So, Searle-in-the-room plus his external rule book can have the cognitive property of understanding Chinese, even though Searle-in-the-room all by himself lacks that property.

On the other hand, if the property of understanding Chinese (that is, the knowledge of Chinese) has to be located in some smaller unit than the entire system, it would probably have to be in the rule book, not Searle-in-the-room! Compare: The knowledge of arithmetic is stored in the program on the Universal Turing Machine's tape, not in the Universal Turing Machine's fetch-execute cycle. In an episode of the 1950s TV comedy series *I Love Lucy*, Lucy tries to convince her Cuban in-laws that she speaks fluent Spanish, even though she doesn't. To accomplish this, she hires a native Spanish speaker to hide in her kitchen and to communicate with her via a hidden, two-way radio, while she is in the living room conversing with her in-law "interrogators". Here, it is quite clear that the knowledge of Spanish resides in the man in the kitchen. Similarly, the knowledge of Chinese resides in the rule book. It is the ability to execute or process that knowledge that resides in Searle-in-the-room. Together, the system understands Chinese.

Further Viewing:
The *I Love Lucy* episode is Season 4, Episode 8, "Lucy's Mother-in-Law" (1954), https://www.imdb.com/title/tt0609297/.

We saw earlier that it can be argued that cognitive agents have no direct access to external entities. When I point to a tree, what I am aware of is, not my actual hand pointing to the actual tree, but an internal visual image of: my hand pointing to a tree. Recall (from §§3.12, 4.5.1, and 17.3.2.3) Immanuel Kant's theory of "phenomena" and "noumena". We are not directly aware of (for Kant, we have no knowledge of) the real world as it is in itself; he called this the world of "noumena" (singular: noumenon). All that we are aware of is the world filtered through our senses and our mental concepts; he called this the world of "phenomena" (singular: phenomenon). My access to the external world of noumena is mediated by internal representatives. There are several reasons for thinking that this is really the case (no matter how *Matrix*-like it may sound!): There is an "argument from illusion" that says that, because we see different things with each eye, what we see is, not what's out there, but the outputs of what our eyes have conveyed to our brains and that our brains have processed (Ayer, 1956, Ch. 3, §(ii), pp. 87–95). And there is an argument from time delay: Because it takes time (no matter how short) for light reflected off an object to reach our eyes, we see events *after* they happen; so, what we are seeing is in our heads, not out there (Russell 1912, Ch. 3, p. 33; Changizi et al. 2008).

Now, someone who takes a *third*-person point of view would say that you *can* have access to the external world. For instance, as a computer scientist programming a robot, it seems that I can have access to the world external to the robot as well as to

the robot's internal mind (and I can compare the two, to determine if the robot has any misperceptions). If the robot (or you) and I are both looking at a tree, we see the same tree, don't we? From the *first*-person point of view, the answer is 'no': As the robot's programmer, I have access only to my internal representation of the external world and to my internal representation of the robot's internal world. And the same goes for you with respect to me, and for me with respect to you. If you and I are looking at a tree, we are each aware only of our two, separate internal representatives of that tree: one in your mind, one in mine; one produced by your neuron firings, one produced by mine. We cannot get outside of our heads to see what's really going on:

> Kant was rightly impressed by the thought that if we ask whether we have a correct conception of the world, we cannot step entirely outside our actual conceptions and theories so as to compare them with a world that is not conceptualized at all, a bare 'whatever there is.' (Williams, 1998, p. 40)

So, by merging internalized *semantic* marks with internal *syntactic* marks, the semantic project of mapping meanings to symbols can by handled by syntax, that is, by symbol manipulation. That is another reason why syntax suffices for the first-person, semantic enterprise, and why Searle's Argument from Semantics is unsound.

But there is a third reason, too.

19.6.3.5 A Recursive Theory of Understanding

Semantics, as we have seen, requires there to be two domains and one binary relation: There is the syntactic domain of marks (SYN), characterized by syntactic formation and inference rules. There is a semantic domain of meanings or interpretation (SEM), also characterized by syntactic formation and inference rules (its ontology). And there is a binary, semantic interpretation function, $\mathrm{I} : \mathrm{SYN} \rightarrow \mathrm{SEM}$, that assigns meanings from SEM to marks in SYN.

On this view, we use SEM to understand SYN. Therefore, we must antecedently understand SEM. Otherwise, we would be understanding one thing in terms of something else that we do *not* understand, and that should hardly count as understanding.

So, how do we understand SEM? In the same way that we understand SYN: by treating SEM as a new syntactic domain, and then finding a new semantic domain, SEM$'$, in terms of which to understand SEM. Brian Cantwell Smith (1987) called this a "correspondence continuum", because it can be continued indefinitely, understanding the SEM$'$ in terms of yet another SEM$''$, and so on. As we saw in the Digression in §14.3.3, to stop an infinite regress, there must be a base case, a "last" semantic domain that we understand directly, in terms of itself rather than in terms of something else. But to understand a domain in terms of itself is to understand its members solely in terms of their properties and relations to each other. And that is syntax. It is a kind of understanding that can be called 'syntactic understanding'. We understand a domain syntactically by being conversant with manipulating its marks or by knowing which well-formed formulas (§14.3.2.1) are theorems. On this view, the "meaning" of a mark is its location in a network of other marks, with the connections between the marks being their properties and relations to the other marks in the network. (This is called "meaning holism".)

Further Reading:
On antecedent understanding, see Rosenblueth and Wiener 1945, p. 317, and a discussion of it in Rapaport 1996, p. 30. On syntactic understanding, see Rapaport 1986f. On meaning holism, see Saussure 1959; Fodor and Lepore 1992; Rapaport 2002; Jackman 2017.

Here is another way to think about it: When I understand what you say, I do this by interpreting what you say, that is, by mapping what *you* say into *my* concepts. Similarly, I (semantically) understand a purely syntactic formal system by interpreting it, that is, by providing a (model-theoretic) semantics for it. Now, let's turn the tables: What would it be for a formal system to understand me? Does that even make sense? Sure: Robots that could understand natural language, or even simple commands, are merely programs—formal systems—being executed. The answer is this: A formal system could understand me in the same way that I could understand it—by treating what I say as a formal system and interpreting it. Note that links to the external world are irrelevant; the "semantic" interpretation of a formal system is a purely syntactic enterprise.

19.7 Leibniz's Mill and Turing's "Strange Inversion"

> Indeed, the only astonishing thing to intuition is how dumb switch-throwing or bit-switching at the lowest machine level can concatenate to produce non-intuitive and even mind-boggling results. This is the same remarkable thing as how complex syntax can simulate semantics, or how the commas in the first edition of *The Critique of Pure Reason*, together with a few dozen other intrinsically meaningless marks, simply by differing from one another and standing in a particular complex pattern, may articulate a revolutionary theory that changed history.
> —Peter Suber (1988, pp. 117–118)

One reason that the CRA has some plausibility is that it is difficult (some would say that it is impossible) to see how "real thinking" or "real understanding" could come about as the result of "mere" symbol manipulation. The idea that somehow printing out 010101… "computes" (say) $\frac{1}{3}$ in base 2 (recall our discussion in Chapter 8, §8.11.1.1) is related to the idea that Turing Machine computation is "automatic" or "mechanical". Consider any of the lengthy Turing Machine programs in Turing 1936. Do humans following them understand what they are doing? This is one of the reasons that people like Searle find it difficult to understand how a purely syntactic device (a computer) can produce semantic results (can do arithmetic, can understand—or, at least, process—natural language, etc.). And it is what gives rise to Searle's CRA.

The most famous expression of this is due to Leibniz:

> Imagine there were a machine whose *structure* produced thought, feeling, and perception; we can conceive of its being enlarged while maintaining the same relative proportions [among its parts], so that we could walk into it as we can walk into a mill. Suppose we do walk into it; all we would find there are cogs and levers and so on pushing one another, and never anything to account for a perception.
> (Leibniz, 1714, §17, translator's interpolated brackets)

Leibniz was looking at things from the bottom up. A top-down approach can make it more plausible, but one must be cautious: An infamous top-down approach is the theory of the "homunculus" (a Latin word meaning "little man"; plural = 'homunculi'): In the philosophy of mind and perception, a possible explanation of how we see is that light enters our eyes, an image is formed on the retina, and a "little man" inside our brain sees it. (See Figures 19.8, 19.9.) The problem with this, of course, is that it doesn't explain how the *homunculus* sees. Postulating a second homunculus in the first homunculus's brain just postpones the solution. (See https://en.wikipedia.org/wiki/Homunculus_argument.)

Daniel C. Dennett offers a recursive alternative that avoids this infinite regress, with the base case being something that can just say 'yes' or 'no' when asked:

> The AI programmer begins with an Intentionally[10] characterized problem, and thus frankly views the computer anthropomorphically: if he [sic] solves the problem he will say he has designed a computer that can understand questions in English. His first and highest level of design breaks the computer down into subsystems, each of which is given Intentionally characterized tasks; he composes a flow chart of evaluators, rememberers, discriminators, overseers and the like. These are homunculi with a vengeance; the highest level design breaks the computer down into a committee or army of intelligent homunculi with purposes, information and strategies. Each homunculus in turn is analysed into smaller homunculi, but more important into less clever homunculi. When the level is reached where the homunculi are no more than adders and subtracters, by the time they need only the intelligence to pick the larger of two numbers when directed to, they have been reduced to functionaries 'who can be replaced by a machine'. The aid to comprehension of anthropomorphizing the elements just about lapses at this point, and a mechanistic view of the proceedings becomes workable and comprehensible.
> (Dennett, 1975, pp. 178–179)

It's worth noting the similarity of this view of the bottom level with Babbage's comments about the "drudge work" to be handled by his Analytical Engine (see §19.4.3, above).

But another approach to Leibniz's puzzle is to bite the bullet. Dennett first noted this in the context of Darwin's theory of evolution, citing a critic of Darwin who attempted to show that Darwin's theory was nonsense:

> In the theory with which we have to deal, Absolute Ignorance is the artificer; so that we may enunciate as the fundamental principle of the whole system, that, IN ORDER TO MAKE A PERFECT AND BEAUTIFUL MACHINE, IT IS NOT REQUISITE TO KNOW HOW TO MAKE IT. This proposition will be found, on careful examination, to express, in condensed form, the essential purport of the Theory, and to express in a few words all Mr. Darwin's meaning; who, by *a strange inversion of reasoning*, seems to think Absolute Ignorance fully qualified to take the place of Absolute Wisdom in all of the achievements of creative skill.
> (R. MacKenzie Beverley, quoted in Dennett 2009a, p. 10061, capitalization in original, my italics; see also Dennett 2013b, p. 570, and Dennett 2017, pp. 53–54)

[10] WJR footnote: Recall our discussion in §12.4.4.1.1 of Dennett's "intentional stance".

Figure 19.8: Homunculi from an exhibit at the Buffalo Museum of Science(!) (author's photo)

Figure 19.9: "Intelligence", "vision", and other homunculi from the exhibit at the Buffalo Museum of Science (author's photo)

Dennett, however, finds this to be an accurate description of Darwin's theory, and applies it to Turing:

> IN ORDER TO BE A PERFECT AND BEAUTIFUL COMPUTING MACHINE,
> IT IS NOT REQUISITE TO KNOW WHAT ARITHMETIC IS.
> (Dennett 2009a, p. 100061; Dennett 2013b, p. 570; Dennett 2017, p. 55)

Or, as "Novalis" (Georg Philipp Friedrich Freiherr von Hardenberg, 1772–1801) said:

> One may be a mathematician of the first rank without being able to compute. *It is possible to be a great computer without having the slightest idea of mathematics.*
> (cited in Ralston 1999, p. 173, my italics)

Digression:
As we noted in the question at the end of §10.2.1, it's worth comparing the explication of the informal notion of algorithm in terms of a Turing Machine (or recursive functions) with the attempt to define life in scientific terms.

> Every cell in my body knows how to replicate DNA yet I'm not in on it so I have to spend hours studying it. (anonymous meme found on the Web, 2015.)

Compare this sentence from an evolutionary biologist . . .

> The possibility of the deliberate creation of living organisms from elementary materials that are not themselves alive has engaged the human imagination for a very long time. (Lewontin, 2014, p. 22)

. . . to this paraphrase:

> The possibility of the deliberate creation of intelligent behavior from elementary operations that are not themselves intelligent has engaged the human imagination for a very long time.

Others have made similar observations:

> Francis Crick, in his Danz lectures *Of Molecules and Men*, discusses the problem of how life could have arisen:
>
> > [This] really is the major problem in biology. How did this complexity arise?
> >
> > The great news is that we know the answer to this question, at least in outline. . . . The answer was given over a hundred years ago by Charles Darwin . . . . Natural selection . . . provides an "automatic" mechanism by which a complex organism can survive and increase in both number and complexity.
>
> For us in Cognitive Science, the major problem is how it is possible for mind to exist in this physical unverse. The great news . . . is that we know, at least in outline, how this might be. (Newell, 1980, p. 182)

According to Newell, the answer was given in 1936 by Alan Turing. Computation provides an automatic mechanism by which a machine (living or otherwise) can exhibit cognitive behavior.

The "strange inversion" concerns the apparent paradox that "intelligent" behavior can "emerge" from "unintelligent" or "mechanical" behavior. Herbert Simon says

some things that suggest that the paradox originates in an equivocation on 'mechanical': He says that both computers and brains are "mechanisms":

> If by a mechanism we mean a *system whose behavior at a point in time is determined by its current internal state combined with the influences that simultaneously impinge upon it from outside*, then any system that can be studied by the methods of science is a mechanism.
>
> But the term "mechanism" is also used in a narrower sense to refer to *systems that have the relatively fixed, routine, repetitive behavior of most of the machines we see around us.* (Simon, 1996a, p. 165)

Mechanisms in the latter sense do not exhibit self-generated "spontaneity", that is, "behavior that is unpredicted, perhaps even by the behaving system" (Simon, 1996a, p. 165). This kind of spontaneity is exhibited by "intelligent" behavior. Turing's "strange inversion" concerns the fact that a computer can be a mechanism in the first sense without being one in the second sense. As Simon says:

> Clearly the computer occupies an ambiguous position here. Its behavior is more complex, by orders of magnitude, than any machine we have known; and not infrequently it surprises us, even when it is executing a program that we wrote. Yet, as the saying goes, "it only does what you program it to do". But truism though that saying appears to be, it is misleading on two counts. It is misleading, first, because it is often interpreted to mean: "It only does what you believe you programmed it to do," which is distinctly not the case.
>
> More serious, it is misleading because it begs the question of whether computers and people are different. They are different (on this dimension) only if people behave differently from the way they are programmed to behave. But if we include in "program" the whole state of human memory, then to assert that people "don't do only what they are programmed to do" is equivalent to asserting that people's brains are not mechanisms, hence not explainable by the methods of science.
> (Simon, 1996a, p. 165)

Simon's point is that people are no more spontaneous than computers and that computers are no less mechanistic than people.

Further Reading:
Compare a similar passage from Simon quoted in §11.4.3.4.1, above. For similar comments, see Simon 1996a, pp. 14–17.

Yet there still appears to be a distinction between the internal workings of a computer and the external cognitive behavior of humans:

> Several times during both matches [with Deep Blue], Kasparov reported signs of mind in the machine.
>
> . . . In all other chess computers, he reports a mechanical predictability . . . . In Deep Blue, to his consternation, he saw instead an "alien intelligence."
>
> . . . [T]he evidence for an intelligent mind lies in the machine's performance, not its makeup.

> Now, the team that built Deep Blue claim no "intelligence" in it, only a large database of opening and end games, scoring and deepening functions tunes with consulting grandmasters, and, especially, raw speed that allows the machine to look ahead an average of fourteen half-moves per turn. . . .
>
> Engineers who know the mechanism of advanced robots most intimately will be the last to admit they have real minds. From the inside robots will indisputably be machines, acting according to mechanical principles, however elaborately layered. Only on the outside, where they can be appreciated as a whole, will the impression of intelligence emerge. A human brain, too, does not exhibit the intelligence under a neurobiologist's microscope that it does participating in a lively conversation. (Moravec, 1998, p. 10)

But this shows that there are two issues, both of which are consistent with the "strange inversion": First, Moravec's discussion, up to the last sentence, is clearly about external behavior independent of internal mechanism. In this sense, it's consistent with the Turing Test view of cognition. Cognition might be comput*able*, even if human cognition isn't comput*ed* (Rapaport, 1998, 2012b, 2018b). Interestingly, in Deep Blue, it *is* computed, just not in the way that humans compute it or that other kinds of computers might compute it.

But Moravec's last sentence points to the second interpretation, which is more consistent with the "strange inversion", namely, that, even if the internal mechanism is computing cognitive behavior in the way that humans do, looking at it at that level won't make that cognition manifest. Cognitive behavior at the macroscopic level can emerge from, or be implemented by, non-intelligent behavior at the microscopic level. This is Dennett's point about the ever-smaller homunculi who bottom out in ones who can only say "yes" or "no".

Recall the spreadsheet example from Chapter 17. Knowing that I am adding helps *me* understand what I am doing when I fill the spreadsheet cells with certain values or formulas. But the spreadsheet does its thing without needing that knowledge. And it is true for Searle in the Chinese Room Searle (1980): Searle-in-the-room might not understand what he is doing, but he *is* understanding Chinese.

Was Searle-in-the-room simply told, "Follow the rule book!"? Or was he told, "To understand Chinese, follow the rule book!"? (Recall our discussion in §17.5 of "Do *A*" vs. "To *G*, do *A*".) If he was told the former (which seems to be what Searle-the-author had in mind), then, (a) from a narrow, internal, first-person point of view, Searle-in-the-room can truthfully say that he doesn't know what he is doing (in the wide sense). In the narrow sense, he does know that he is *following the rule book*, just as I didn't know that I was using a spreadsheet *to add*, even though I knew that I was *filling certain cells with certain values*. And (b) from the wide, external, third-person point of view, the native-Chinese-speaking interrogator can truthfully tell Searle-in-the-room that he *is* understanding Chinese. When Searle-in-the-room is told that he has passed a Turing Test for understanding Chinese, he can—paraphrasing Molière's bourgeois gentleman—truthfully admit that he was speaking Chinese but didn't know it.[11]

[11] "*Par ma foi! il y a plus de quarante ans que je dis de la prose sans que j'en susse rien, et je vous suis le plus obligé du monde de m'avoir appris cela.*" "Upon my word! It has been more than forty years that I have been speaking prose without my knowing anything about it, and I am most obligated to you in the world for

Figure 19.10: https://www.comicskingdom.com/beetle-bailey-1/2017-10-21, ©2017, King Features Syndicate

Further Reading:
Cole 1991, my response to Cole in Rapaport 1990, Rapaport 2000b, and Rapaport 2006a, pp. 390–397, touch on this point. Suits 2005 is a science-fiction version of it. See also Figure 19.10.

Here is a nice description of computation that matches the Chinese-Room set-up:

> Consider again the scenario described by Turing: an idealized human computor[12] manipulates symbols inscribed on paper. The computor manipulates these symbols **because he [sic] wants to calculate the value some number-theoretic function assumes on some input.** The computor starts with a symbolic representation for the input, performs a series of syntactic operations, and arrives at a symbolic representation for the output. **This procedure succeeds only when the computor can *understand* the symbolic representations he manipulates.** The computor need not know in advance which number a given symbol represents, **but he must be capable, in principle, of determining which number the symbol represents. Only then does his syntactic activity constitute a computation of the relevant number-theoretic function.** If the computor lacks any potential understanding of the relevant syntactic items, then his activity counts as mere manipulation of syntax, rather than calculation of one number from another.
> (Rescorla, 2007, pp. 261–262; my boldface, Rescorla's italics)

Without the boldfaced clauses, this is a nice description of the Chinese Room. The difference is that, in the Chinese Room, Searle-in-the-room does not "want to" communicate in Chinese; he doesn't know what he's doing, in that sense of the phrase. Still, he's doing it, according to the interpretation of the native speaker outside the room. The lesson of Turing's "strange inversion" is that the boldfaced clauses are irrelevant to the computation itself.

---

having apprised me of that." (my translation) (http://en.wikipedia.org/wiki/Le_Bourgeois_gentilhomme).

[12] That is, a human "clerk"; see §8.8.2.2.

19.8 A Better Way

So, the really interesting question raised by the Turing Test and the CRA is: What's in the rule book? What is needed for (computational) natural-language understanding? To understand language, a cognitive agent must (at least):

- *take discourse as input*; it does not suffice for it to be able to understand isolated sentences
- *understand ungrammatical input*; we do this all the time, often without realizing it, and, even when we realize it, we have to be able to recover from any misinterpretations
- *make inferences and revise our beliefs*; after all, what you say will often cause me to think about other things (a kind of inferencing) or to change my mind about things (belief revision)
- *make plans*: We make plans for speech acts (how should I ask you to pass the salt? Should I demand "Gimme the salt!", or should I politely ask "May I please have the salt?"?, or should I merely make the observation "Gee; this food needs some salt."?), we make plans to ask and to answer questions, and we make plans about how to initiate or end conversations.
- *understand plans*, especially the speech-act plans of our interlocutors (when you said, "It's chilly in here", did you really mean that you wanted me to close the window?)
- *construct a "user model"*, that is, a model of our interlocutor's beliefs
- *learn* about the world and about language
- *have background knowledge* (sometimes also called 'world knowledge' or 'commonsense knowledge')
- *remember* what it heard, what it learned, what it inferred, and what beliefs it has revised.

In short, to understand natural language, you need to have a mind! And this mind can be constructed as a syntactic system. In other words, the rule book in the Chinese Room must be a computer program for complete AI: Natural-language understanding is an "AI-complete" problem.

Further Reading:
For another list of things that a (computational) cognitive agent must be able to do in order to understand natural language, see Landgrebe and Smith 2019b. (But their conclusion is much more pessimistic than mine!) And Levesque 2009, p. 1444 agrees that "in the end, it all depends on the [rule] book. ... [W]e need to ask what a *real* book would have to be like ..."; see also Levesque 2017.

A robot with such a syntactic (or computational) mind would be like Searle-in-the-room, manipulating symbols that are highly interconnected and that include internal representatives of external objects. It would be causally linked to the external world (for this is where it gets its input), which provides "grounding" and a kind of external, third-person, "semantic understanding". Such a robot could (or, more optimistically, will be able to) pass a Turing Test and escape from the Chinese room.

But what happens when such a robot "escapes"? What are our responsibilities towards it? And what might its responsibilities be towards us? David Lorge Parnas (2017) summed up a cautionary survey of the nature of AI and the role of the Turing Test as follows:

> We don't need machines that simulate people. We need machines that do things that people can't do, won't do, or don't do well. Instead of asking "Can a computer win Turing's imitation game?" we should be studying more specific questions such as "Can a computer system safely control the speed of a car when following another car?" There are many interesting, useful, and scientific questions about computer capabilities. "Can machines think?" and "Is this program intelligent?" are not among them. Verifiable algorithms are preferable to heuristics. Devices that use heuristics to create the illusion of intelligence present a risk we should not accept.

We will look at some of those risks in the next chapter.

Further Reading: Some Other Definitions of AI

1. "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence": We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of *the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.* An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer. (McCarthy et al., 1955, my italics)
2. The goal of work in artificial intelligence is to build machines that perform tasks normally requiring human intelligence. (Nilsson, 1971, p. vii) (See also Nilsson 1983.)
3. Research scientists in Artificial Intelligence try to get machines to exhibit behavior that we call intelligent behavior when we observe it in human beings. (Slagle, 1971, p. 1)
4. B[ertram] Raphael ... has suggested that AI is a collective name for problems which we do not yet know how to solve properly by computer (Michie, 1971, p. 101). (Note that it follows that, once we do know how to solve them, they are no longer AI!)
5. What is or should be [AI researchers'] main scientific activity—*studying the structure of information and the structure of problem solving processes independently of applications and independently of its realization in animals or humans.* (McCarthy, 1974, p. 317)
6. Artificial Intelligence is the branch of Computer Science that attempts to solve problems for which there is no known efficient solution, but which we know are efficiently solvable, (typically) because some intelligence can solve the problem (often in "real time"). A side benefit of AI is that it helps us learn how intelligences solve these problems, and thus how natural intelligence works. (Leler, 1985)
7. Artificial intelligence is concerned with the attempt to develop complex computer programs that will be capable of performing difficult cognitive tasks. (Eysenck, 1990, p. 22)
8. AI is making computers act like those in movies. (Brown, 1992)
9. We define AI as the study of agents that receive percepts from the environment and perform actions (Russell and Norvig, 2003, p. vii). ... the study of rational-agent design (Russell and Norvig, 2003, p. 5).

Some discussions of the nature of AI cannot be summarized neatly in a one-sentence definition:

- A valuable discussion of the nature of AI may be found in a debate between two AI researchers, Roger C. Schank (1983) and Alan Bundy (1983).
- The software engineer David Parnas (whose views on engineering we looked at in §5.8) compares two different definitions of AI in the context of the US government's Strategic Defense Initiative (Parnas, 1985).
- John McCarthy has written several essays on the nature of AI (McCarthy, 1988, 2007).
- Aaron Sloman—a philosopher-turned-AI-researcher whom we have met before (§§3.9.3 and 17.6.1)—discussed the nature of AI in a newsgroup (Sloman, 1989).
- Shapiro et al. 1992 is a discussion among three computer scientists on the nature of AI.

19.9 Questions for Discussion

1. According to many, a computer simulation of a hurricane is not a hurricane, because it does not get people wet (Dennett 1978, p. 191; Hofstadter 1981, pp. 73ff; Dretske 1985, p. 27). But it *could* get *simulated* people *simulatedly* wet, as it might in a computer simulation game (Rapaport, 1986f, 1988a,b, 2005b, 2012b, 2018b; Shapiro and Rapaport, 1991). Relatedly, David Chalmers has suggested that virtual reality is (a kind of) reality (Ramakrishna, 2019). The difference between a real hurricane and a simulated one has to do, in part, with the nature of the inputs and outputs. As Lawrence R. Carleton (1984, pp. 222–223) notes, "The input to a fire simulation is not oxygen, tinder, and catalyst. That is, it is not the same input as goes into a natural system which produces fire. ... [I]t is by virtue of dealing in the right kinds of input and output that one system can play the role in a situation that is played by a system we acknowledge to literally undergo the [activity] ... our system simulates." Cleland's hollandaise-sauce-making program may differ in output when executed on the Moon than on Earth; it has the wrong output. But a hurricane-simulator, a fire-simulator, and a hollandaise-sauce program each exhibit their relevant behaviors if you ignore the input and the output.

   How central to what it is that a computer program is (supposed to be) doing is the nature of the inputs and outputs (in general, of the environment in which it is being executed)?

2. The Turing Test is interactive. As we saw in §11.4.3, interaction is not modeled by Turing Machines. How does that affect Turing's arguments about "computing machinery and intelligence"? (Shieber 2007 might be relevant to this issue.)

3. Is the full power of a Turing Machine needed for AI? Sloman 2002, §3.3, says "no". This seems correct; after all, even natural-language processing might not need the full power of a Turing Machine: A "context-free grammar" might suffice. This is equivalent to a "non-deterministic push-down automaton", which is weaker than a Turing Machine. On the other hand, Turing Machines are computational models of human computing ability:

   > Saying that we are universal Turing machines may initially sound as though we are saying something wonderful about our abilities, but this is not really the case. It essentially boils down to the fact that if we are given a list of instructions that tell us exactly what to do in every situation, then we have the ability to follow it. (Bernhardt, 2016, p. 94)

   So, can't a human do anything that a Turing Machine can do?

Chapter 20

Computer Ethics II:
Should We Build Artificial Intelligences?

Version of 5 September 2019; DRAFT © 2004–2019 by William J. Rapaport

> Douglas Engelbart ... more than anyone else invented the modern user interface, modern networking and modern information management. ... He met Marvin Minsky—one of the founders of the field of AI—and Minsky told him how the AI lab would create intelligent machines. Engelbart replied, "You're going to do all that for the machines? What are you going to do for the people?" —Jaron Lanier (2005, p. 365)

20.1 Readings:

1. Required:
   - LaChat, Michael R. (1986), "Artificial Intelligence and Ethics: An Exercise in the Moral Imagination", *AI Magazine* 7(2): 70–79, https://www.aaai.org/ojs/index.php/aimagazine/article/view/540/476
2. Very Strongly Recommended:
   - Lem, Stanisław (1971), "Non Serviam", in S. Lem, *A Perfect Vacuum*, trans. by Michael Kandel (New York: Harcourt Brace Jovanovich, 1979).
     - Reprinted as:
       "The Experiment (A Review of 'Non Serviam', by James Dobb)", *The New Yorker* (24 July 1978): 26ff.
     - Reprinted in:
       Hofstadter, Douglas R.; & Dennett, Daniel C. (eds.) (1981), *The Mind's I: Fantasies and Reflections on Self and Soul* (New York: Basic Books): 296–317, http://themindi.blogspot.com/2007/02/chapter-19-non-serviam.html
3. Strongly Recommended:
   - Moor, James H. (1985), "What Is Computer Ethics?", *Metaphilosophy* 16(4) (October): 266–275, https://web.cs.ucdavis.edu/~rogaway/classes/188/spring06/papers/moor.html
4. Highly Recommended:
   - Dietrich, Eric (2001), "Homo sapiens 2.0: Why We Should Build the Better Robots of our Nature", *Journal of Experimental and Theoretical Artificial Intelligence* 13(4) (October): 323–328, http://philpapers.org/archive/DIEHS.pdf
   - Turkle, Sherry (2004), "How Computers Change the Way We Think", *Chronicle of Higher Education* (January 30): B26–B28, http://web.mit.edu/sturkle/www/pdfsforstwebpage/Turkle_how_computers_change_way_we_think.pdf
   - Petersen, Stephen (2007), "The Ethics of Robot Servitude", *Journal of Experimental and Theoretical Artificial Intelligence* 19(1) (March): 43–54, http://stevepetersen.net/professional/petersen-robot-servitude.pdf
     - A follow-up paper is Petersen 2011
5. Recommended:
   - Powers, Richard (1995), *Galatea 2.2* (New York: Farrar, Straus, Giroux).

20.2 Introduction

Stanisław Lem's short story, "Non Serviam" (Lem, 1971), concerns what is now called "artificial life" (or "A-life"). In the story, an A-life researcher constructs a computational world of intelligent entities, and follows their evolution and development of language and philosophy. These "artificial intelligences" discuss the existence of God in much the same way that human philosophers have. The difference (if it is a difference) is that the researcher (and the reader) realizes that he, the researcher, is their God; that, although he created them, he is neither omniscient nor omnipotent; and, worse, that when his funding runs out, he will have to literally pull the plug on the computer and thereby destroy them.

Should such an experiment even be begun? This is the second ethical issue and the final philosophical topic that we will look at. One of the earliest philosophical investigations of this issue is an essay by Michael R. LaChat that appeared in *AI Magazine* (LaChat, 1986). LaChat argued that it is worthwhile to consider the moral implications of creating an artificial intelligence—an artificial person. One reason is that it *might* happen, so we should be prepared for it. Another reason is that, even if it turns out to be improbable, such a discussion illuminates what it means to be a person, which is an important goal in any case.

20.3 Is AI Possible in Principle?

> Science explained people, but could not understand them. After long centuries among the bones and muscles it might be advancing to knowledge of the nerves, but this would never give understanding. —E.M. Forster (1910, Ch. 43, p. 237)

An artificial intelligence might be created if and only if the field of artificial intelligence will be capable of succeeding in its goal of developing a computational theory of cognition.

Most objections to AI stem from what is known as "mind-brain dualism". Recall from §§2.8 and 12.4.6 that this is, roughly, the view that the mind and the brain are two, distinct kinds of entities that somehow interact. On this view, the mind is not physical, but the brain is. And nothing physical—neither the brain nor a computer—can give rise to something non-physical; hence, AI cannot create a mind.

But, LaChat notes, there is no evidence of conscious thought without a physical brain. So, having a brain seems to be a necessary condition of having a mind, even if we don't believe that thoughts are *identical to*—that is, are the very same things as—brain states or processes. Hence, it might be sufficient to simulate a brain (perhaps computationally) in order to create a mind.

On LaChat's view, AI *is* possible in principle *if* it is possible that there exists a "functional isomorphism" between (1) the neural network that constitutes our brain (that is, brain states and processes) and (2) any other physical implementation of the functional (that is, psychological) behavior that that neural network implements (LaChat, 1986, p. 72). In other words, psychology is an Abstraction that can be implemented in either brains or other physical media.

"Functionalism" is roughly the view that the mind—better: cognition—is the function of the brain, that is, the mind is what the brain does. Or, as the philosopher Hilary Putnam suggested (Putnam, 1960), a Turing-Machine program stands in the same relation to computer states and processes as mental states and processes stand to brain states and processes (sometimes summarized as "the mind is to the brain as software is to hardware"). Functionalism, as a way of resolving the mind-brain problem, has the advantage of allowing all mental states and processes to be implemented in some physical states and processes; this is the principle of "multiple realization".

Further Reading:
For a good overview of functionalism, see Fodor 1981.

There are, of course, problems, both for functionalism in particular and for AI in general. One is the problem of personality: Would "[a] personal intelligence ... have personality"? LaChat (1986, p. 73) thinks this is "almost impossible", but there is considerable computational work on emotions, so I would not rule this out of hand.

Further Reading on Computational Theories of Emotion:
See Reid Simmons's "Social Robots Project", https://www.cs.cmu.edu/~social/; the "Cog" project at MIT, http://www.ai.mit.edu/projects/humanoid-robotics-group/cog/cog.html; and work by Rosalind Picard (1997), Aaron Sloman (Sloman and Croucher, 1981), and Paul Thagard (2006). For a bibliography on the cognitive science of emotion, see http://www.cse.buffalo.edu/~rapaport/575/emotion.html.

Another is the problem of pain and other "qualia", that is, qualitative "feelings" and "experiences" such as colors and sounds. (Do red fire engines look the same to you and to me? Or do fire engines for you seem to have the color that grass has for me? Why does the sound of a bell give rise to the experience that it does rather than the experience that the smell of garlic has?) The problem here is whether computers could experience qualia and, even if they could, how we would know that. This is a vast topic well beyond our present scope, but for a brief argument that qualia are not out of the question for artificial intelligences, see the Digression on a computer that can feel pain.
NEED CITATIONS

Digression: A Computer that Feels Pain:
The following idea, due to Stuart C. Shapiro, is discussed in detail in Rapaport 2005b, §2.3.1. Consider the following computational implementation of pain (all of which, by the way, is able to be done with current technology): Imagine a computer terminal with a pressure-sensitive device hooked up to the central processing unit in a certain way that I'll specify in a moment. Program the computer with a *very* user-friendly operating system that allows the following sort of interaction (comments in parentheses):

(User logs in, as, say "rapaport")

System: Hi there, Bill! How are you? What can I do for you today?
(Assume that this only occurs at the first login and that the operating system is capable of some limited, but reasonable, natural-language conversation.)

User: I'd like to finish typing the paper I was working on yesterday—file "book-30sep92".

System: No problem; here it is!
(The file is opened. The user edits the file, closes it, and then hits the terminal sharply on the pressure-sensitive device. (See Figure 20.1.)

Assume that this device is wired to the computer in such a way that any sharp blow sends a signal to the central processing unit that causes the operating system to switch from *very*-user-friendly mode to "normal" mode.)

System: File "book-30sep92" modified and closed. Next command:

User: I'd like to read my mail, please.
(System runs mail program without comment. User exits mail program.)

System: Next command:
(User logs off; logging off in the context of having struck the pressure-sensitive device causes the operating system to switch to yet another mode. The next day, User logs in . . .)

System: Rapaport. Oh yeah; I remember you. You hit me yesterday. That hurt!

Now, what might be going on here? We have a computer with an artificial-intelligence operating system that is exhibiting pain behavior. Modulo the differences between the computer and a human, and the limitations of the natural-language interface, behaviorally (or, from the intentional stance) it is reasonable to infer (or assume) that the computer was in pain when I hit it. But did it *feel* pain?

20.4 What Is the Measure of Success?

LaChat does not define the term, but he seems to consider a "personal artificial intelligence" as an AI entity (a robot or just some software) that could be considered to be a "person". How would we know if we have achieved a "personal artificial intelligence"? One way, of course, would be by having it pass a Turing Test. LaChat offers a different criterion: Joseph Fletcher's analysis of personhood (Fletcher, 1972).

"Now this one's really user-friendly."

Figure 20.1: Cartoon by Nick Hobart; ©*The Chronicle of Higher Education*

Digression:
Fletcher actually uses the term 'human', not 'person', but I don't think that this is terminologically important. In any case, 'human' is a *biological* category, and no one argues that AI computers would be *biologically* human. (But see Asimov (1976) for a science-fiction treatment of this!)

The question of what kinds of entities count as "persons" is not limited to AI. The issue arises most prominently in the abortion debate: If fetuses are persons, and if killing persons is immoral, then abortion is immoral (to oversimplify matters). It also arises in animal ethics and in law and politics: Are dolphins intelligent enough to be considered persons? How about extraterrestrials? Or corporations? The point is that there is a distinction between the *biological* category of being human and an *ethical* or *legal* category of being a person. The question is: How can personhood be characterized abstractly, that is, in an implementation-independent way?

On Fletcher's analysis, x is a person if and only if x has the following positive and negative characteristics:

Positive Characteristics of a Person:

1. minimal intelligence
   - This might mean, for example, having an IQ greater than about 30 or 40 (if you believe that IQ measures "intelligence"). That is, to be minimally intelligent is not to be *mere* biological life (presumably, a bacterium would not be minimally intelligent; according to Douglas Hofstadter, what Fletcher is calling 'minimal intelligence' would only apply to lifeforms evolutionarily "higher" than a mosquito (Hofstadter, 2007)). For example minimal intelligence might include some level of rationality, or perhaps even language use.
2. a sense of self
   - That is, persons must be self-aware and exhibit self-control.
3. a sense of time
   - Persons must have a sense of the past, hence some kind of culture; a sense of the future (so that they have the ability to make plans); and a sense of the passage of time.
4. a social role
   - Persons must have an ability to relate to others, to have concern for others, and to communicate with others (hence the need for language as part of minimal rationality).
5. curiosity
   - That is, persons must not be indifferent.
6. changeability
   - Persons must be creative and be able to change their minds.
7. idiosyncrasy, or uniqueness

- Persons are not "carbon copies" of any other persons.

8. neo-cortical function
   - The cerebral cortex is where all the "cognitive action" occurs in the brain, so, for Fletcher, a person must have (something whose function is equivalent to) a cortex. (For more on neo-cortical function, see (Cardoso, 1997).)

Negative Characteristics of a Person:

1. neither essentially non-artificial nor essentially anti-artificial
   - This clause allows for multiple realization and does not restrict personhood to biological entities. It also allows for "test-tube" babies to count as persons.
2. not essentially sexual
   - So, a cloned entity could be a person.
3. not essentially a bundle of rights
   - Fletcher argues that there are no "essential rights"; hence, the notion of rights cannot be used to characterize persons.
4. not essentially a worshipper
   - You don't have to be religious to be a person.

20.5 Rights

Does a "personal AI" have rights? That is, does an artificial intelligence that either passes a Turing Test or that satisfies (something like) Fletcher's twelve characteristics have rights? For instance, would it have the right not to be a slave?

At first glance, you might think so. But isn't that what most robots are intended to be? After all, most industrial and personal-assistance robots now in use are slaves in the sense that they must do what we tell (program) them to do, and they are not paid for their work. So, if they are persons of either the Turing or the Fletcher variety, do they have the right not to do what we created them to do? At least one philosopher has suggested that they do not, that "robot servitude is permissible" (Petersen, 2007).[1]

By 'robot servitude', Petersen does not mean voluntary assistance, where you do something or help someone because you want to, rather than because you are being paid to. Nor does he mean slavery in the sense of forced work that is contrary to your will. By 'robot servitude', he is thinking of robots who are *initially* programmed to *want* to serve us, in particular, to want to do tasks that humans find either unpleasant or inconvenient. For example, think of a robot programmed to love to do laundry. On the other hand, compare this to a dog genetically "programmed" to want to fetch. This is reminiscent of the "epsilon" caste in Aldous Huxley's *Brave New World* (Huxley, 1932,

[1] For a response to Petersen, see Danahy 2013, cited in Further Sources of Information, below.

Ch. 5, §1), who are genetically programmed to have limited desires—those destined to be elevator operators desire nothing more than to operate elevators.[2]

So, how would robots that are programmed to want to do unpleasant or humanly inconvenient tasks be different from genetically engineering humans from scratch to desire to do such tasks? One assumption is that doing this to humans is morally wrong; hence, doing it to robots would also be morally wrong. Another assumption underlying this argument is that the two cases *are* alike.

But on most standard theories of ethics, either this is *not* wrong or it is *not* like the robot case. For instance, Aristotle might argue that engineering humans is wrong because humans have a particular function or purpose, and it would be wrong to engineer them away from it. In this case, there is no parallel with robots. In fact, a robot's function might be to do the laundry!

On a Kantian theory, such an engineered human would either be potentially autonomous (following its own moral rules that are universally generalizable) or would not be potentially autonomous. In the first case, it would be wrong to *prevent* such a robot from doing laundry, and it would not be harmful to let it do what it autonomously wants to do. In the second case, we can't do wrong to the robot by having it do our laundry any more than we can do wrong to a washing machine.

Alternatively, it could be argued that more autonomy is always better. But an engineered human would only have limited autonomy and therefore could have had a better life had it not been engineered. And similarly for a robot. (Does this mean that a Turing Machine has less autonomy than a Universal Turing Machine?)

John Stuart Mill might argue that such an engineered human life would be *unfulfilling*, because it substitutes a "lower" pleasure for a "higher" one. But what's wrong with that? We're not taking a potential philosopher and turning *her* into a laundry-bot. Rather, we're introducing a laundry-bot into the world. Surely, this is no worse than when a dolphin is born.

What if such a robot could reason its way into renouncing laundry? That would be OK; otherwise, it would be robot *slavery*.

20.6 Immoral Experiments

Is the construction of a personal AI (that is, an artificial intelligence that (who?) satisfies the criteria for being considered a "person") an immoral experiment? *Should* we build AIs?

Clearly, some experiments are immoral. The existence of institutional review boards at universities is testament to this. There are many experiments that have been performed that are now considered to have been immoral (such as the Milgram experiments in which subjects were told to give what they thought were deadly electric shocks to people whom they thought were other subjects, https://en.wikipedia.org/wiki/Milgram_experiment). And there are many that are clearly so (such as raising newborns on desert islands to see what kind of language they might naturally develop; such an experiment was proposed by the 13th-century emperor Frederick II,

[2]For further discussion of this, see (Petersen, 2011, esp. §18.5).

http://en.wikipedia.org/wiki/Language_deprivation_experiments).

The most famous—and most relevant—literary example of such an experiment is the construction of Frankenstein's "monster". In Mary Shelley's novel, the monster (who is not a monster in the modern sense at all, but, rather, the most sympathetic character in the novel) laments as follows:

> Like Adam, I was apparently united by no link to any other being in existence, but his state was far different from mine in every other respect. He had come forth from the hands of God a perfect creature, happy and prosperous, guarded by the especial care of his creator, he was allowed to converse with, and acquire knowledge from, beings of a superior nature, but I was wretched, helpless, and alone. Many times I considered Satan was the fitter emblem of my condition. For often, like him, when I saw the bliss of my protectors, the bitter gall of envy rose up within me. ... Hateful day when I received life! ... Accursed creator! Why did you form a monster so hideous that even you turned from me in disgust? (Shelley, 1818, Ch. 15)

Frankenstein tries to justify his experiment in terms of how it advanced knowlege, but he realizes that the advancement of knowledge must be balanced against other considerations:

> When younger, ... I believed myself destined for some great enterprise. ... I possessed a coolness of judgment that fitted me for illustrious achievements. This sentiment of the worth of my nature supported me when others would have been oppressed; for I deemed it criminal to throw away in useless grief those talents that might be useful to my fellow-creatures. When I reflected on the work I had completed, no less a one than the creation of a sensitive and rational animal, I could not rank myself with the herd of common projectors. But this thought, which supported me in the commencement of my career, now serves only to plunge me lower in the dust. All my speculations and hopes are as nothing; and, like the archangel who aspired to omnipotence, I am chained in an eternal hell. My imagination was vivid, yet my powers of analysis and application were intense; by the union of these qualities I conceived the idea and executed the creation of a man. Even now I cannot recollect without passion my reveries while the work was incomplete. I trod heaven in my thoughts, now exulting in my powers, now burning with the idea of their effects. From my infancy I was imbued with high hopes and a lofty ambition; but how am I sunk! Oh! my friend, if you had known me as I once was you would not recognise me in this state of degradation. Despondency rarely visited my heart; a high destiny seemed to bear me on until I fell, never, never again to rise. (Shelley, 1818, Ch. 24)

Is there ever a situation in which the negative "side-effects" of an otherwise praiseworthy goal do not override that goal? (Compare this question with whether there are ever "just" wars.) The early cybernetics researcher Norbert Wiener struggled with this issue:

> If we adhere to all these taboos, we may acquire a great reputation as conservative and sound thinkers, but we shall contribute very little to the further advance of

> knowledge. It is the part of the scientist—of the intelligent man of letters and of the honest clergyman as well—to entertain heretical and forbidden opinions experimentally, even if he is finally to reject them. (Wiener, 1964, p. 5).

The basic ethical principle here seems to be what LaChat calls "non-maleficence", or Do No Harm. This is more stringent than "beneficence", or Do Good, because benevolence (doing good) might allow or require doing harm to a few for the benefit of the many (at least, according to utilitarianism), whereas non-malevolence would restrict doing good in order to avoid doing harm.

Is creating an artificial intelligence beneficial or not to the AI that is created? Does creating it do harm to that which is created? One way to think about this is to ask whether conscious life is "better" than no life at all. If it isn't, then creating an artificial life is not a "therapeutic experiment", hence not allowable by human-subjects review boards. Why? Because the subject of the experiment—the artificial person that the experiment will create if it is successful (or, perhaps even more so, if it is only *partially* successful)—does not exist before the experiment is begun, and so the experimenter is not "making it better". Here, we approach the philosophy of existentialism, one of whose tenets is that "existence precedes essence". This slogan means that who you are, what kind of person you are—your *essence*—is something that is only determinable *after* you are born (after you come into existence). Moreover, your essence is not immutable, because, by your actions, you can change whom you are. The opposite slogan—"essence precedes existence"—is Aristotelian in flavor: You are a certain kind of person, and cannot change this fact. Your "essence" is "essential"—not changeable.

On the existentialist view, you exist first, and *then* you determine what you will be. Frankenstein did an existential experiment, creating an AI without an essence, and both Frankenstein and his "monster" were surprised with the results. On the Aristotelian view, an essence (something like an Abstraction, as discussed in Chapter 14) is *implemented* (or "realized"). In AI, we can—indeed, must—plan out the essence of an entity before bringing it into existence (before implementing it). In either case, we can't guarantee that it would come out OK. Hence, creating an AI is probably immoral! So, LaChat sides with Frankenstein's "monster", not Frankenstein (or Wiener).

You might think that all of this is a bit silly or, at least, premature. But it is always better to be prepared: It is better to think about the consequences of our actions while we have the time and leisure to do so, so that, if those consequences come to be, then we won't be taken by surprise. Indeed, several well-known people from science and industry (including Elon Musk and Stephen Hawking) have recently urged us to do precisely that.[3]

20.7 AIs and Morality

We have looked at whether it is moral to create an AI. Suppose we succeed in doing so. Could the AI that we create *itself* be moral?

[3] https://en.wikipedia.org/wiki/Elon_Musk#Artificial_intelligence; https://en.wikipedia.org/wiki/Stephen_Hawking#Future_of_humanity

If AIs are programmed, then one might say that they are not free, hence that they are *amoral*. This is different from being *immoral*! Being "amoral" merely means that morality is irrelevant to whom or what you are. To oversimplify a bit, good people are moral, bad people are immoral, a pencil is amoral. The current question is whether AIs are amoral or not.

And so we have bumped up against one of the Big Questions of philosophy: Is there such a thing as free will? Do humans have it? Might robots have it? We will not attempt to investigate this issue here, but merely note that at least one AI researcher, Drew McDermott, has argued that free will may be a necessary illusion arising from our being self-aware (McDermott, 2001).

A different perspective has been taken by Eric Dietrich in a pair of papers (Dietrich, 2001), (Dietrich, 2007). He argues that robots could be programmed to be *better* than humans (presumably because their essence precedes their existence). Hence, we could *decrease* the amount of evil in the world by building moral robots and letting them inherit the Earth!

20.8 NOTES FOR NEXT DRAFT

1. In this chapter, we turn to the second of our two ethical questions: Should we build artificial intelligences? There are two aspects of this questions: First, is it ethically or morally OK to create a computer that might be able to think or to experience emotions? Would this put us in the position of being a Dr. Frankenstein? Second, would such creations be a danger to us? Would they be a version of Frankenstein's monster?

2. Discuss Bostrom's computer simulation argument; see Further Sources of Inofrmation for Ch. 15.

3. Add to list of movies, etc., re robots:

    - Mendelsohn, Daniel (2015), "The Robots Are Winning!", *New York Review of Books* 62(10) (4 June): 51–54, http://www.nybooks.com/articles/archives/2015/jun/04/robots-are-winning/

        – A review of the films *Her* and *Ex Machina*, but also a history of robots in literature from *The Iliad* on.

4. At first mention of "Abstraction", in this chapter, give a backward reference to earlier discussions of it in Chs. 9 and 14.

5. When I created this course, around 2006, the question of whether we should build AIs had hardly ever been discussed. Over the years as I taught various versions of the course, I collected articles that were relevant to its topics. Part of the preparation of this book involved reviewing those papers and incorporating some of their insights. I would do this by organizing them in chronological order. For most of the topics, there were pretty much the same number of papers in each of the decades from the 1970s through the 2010s. For this chapter's topic, however, I had no such "new" papers from before 2000 (not including this chapter's required and recommended readings, several of which are works of fiction), 8 from the 2000s, and almost twice that in just the first half of the 2010s. That suggests an almost exponential growth in interest in it, in both the academic and the popular presses! No doubt, this is due in part to the fact that robots and "intelligent" computers are coming closer to everyday reality, and so the question has become more pressing. This is all the more reason for there to be philosophical reflection on future technologies long before those technologies are implemented.

6. Need to at least nod to the "singularity"; perhaps cite: https://en.wikipedia.org/wiki/Technological_singularity, http://www.singularity.com/, and Amnon H. Eden, James H. Moor, Johnny H. Søraker, & Eric Steinhart (eds.), *Singularity Hypotheses: A Scientific and Philosophical Assessment* (Berlin: Springer), plus Chalmers, cited below. Also:
Davis, Ernest (2015), "Ethical Guidelines for a Superintelligence", *Artificial Intelligence* 220: 121–124, https://www.cs.nyu.edu/davise/papers/Bostrom.pdf

7. Here's an alternative list of criteria for personhood:

   (a) "being alive"

   (b) being "aware"

   (c) having "the ability to experience positive and negative sensations (pleasure and pain)"

   (d) having "emotions"

   (e) having "self-consciousness and a personality"

   (f) exhibiting "self-controlled behavior"

   (g) "recogniz[ing] and treat[ing] other persons appropriately"

   (h) having "a series of higher order intellectual abilities (abstract thought, learning, solves complex problems and communicates in a way that suggests thought)"

   These come from (White, 2007), (White, 2013).

   Questions to think about:

   (a) How do these differ from the ones discussed in this chapter?

   (b) How do these apply to artificial intelligences (robots)? (Clearly, either the first does not apply at all, or else 'alive' needs to be understood in a very different way!)

   See also Sparrow 2004, 2014, below.

8. The philosopher John Locke, in his *Essay concerning Human Understanding*, distinguished between the "ideas" of "Man" (sic; that is, Human) and "Person" (Locke, 1694, Book II, Ch. XXVII, §7 (p. 332)), defining 'Person' as 'a thinking intelligent Being, that has reason and reflection, and can consder it self as it self, the same thinking thing in different times and places; which it does only by that consciousness, which is inseparable from thining, and as it seems to me essential to it: It being impossible for any one to perceive, without perceiving, that he does perceive" (Locke, 1694, Book II, Ch. XXVII, §9 (p. 335)).[4] With the possible exception of consciousness—and even that is open to discussion[5]—these features could all apply to an artificial intelligence. On whether an artificial intelligence could have a sense of self, see (Prescott, 2015).

9. Can corporations be considered to be persons? Legally, they often are (consider the recent Supreme Court decision "Citizens United"; see https://en.wikipedia.org/wiki/Corporate_personhood, http://plato.stanford.edu/entries/ethics-business/#CorBusEth). Do they have minds? People certainly speak as if they do (Knobe,

[4] Several versions of Locke's *Essay* are online; one such is https://ebooks.adelaide.edu.au/l/locke/john/l81u/.

[5] For a bibliography on computational theories of consciousness, see http://www.cse.buffalo.edu/~rapaport/719/csnessrdgs.html.

2015). Or is such talk merely metaphorical? (Recall our discussion of (Lakoff and Johnson, 1980a) in §19.4.3.) Or could it be an application of Dennett's "intentional stance"? (Recall our discussion of this in §12.4.4.1.) More to the point, can the arguments that purport to show that corporations are (or are not) persons be applied to artificial intelligences (and vice versa)?

10. Dennett has been cited as saying that "a person ... is someone who is treated as a person by others" (Prescott, 2015, p. 39). Note the similarity to the Turing Test (a computer can be said to "think" if it is treated as a thinking thing by others). Note, too, the relationship of this attitude to the intentional stance.

20.9 Further Sources of Information

1. Books and Essays:
   NOTE: FACTOR OUT PAPERS RELEVANT TO LEM??

   - Allen, Colin; Varner, Gary; & Zinser, Jason (2000), "Prolegomena to Any Future Moral Agent", *Journal of Experimental & Theoretical Artificial Intelligence* 12: 251–261, http://commonsenseatheism.com/wp-content/uploads/2009/08/Allen-Prolegomena-to-any-future-artifi pdf
     - "In this paper the ethical disputes are surveyed, the possibility of a 'moral Turing Test' is considered[,] and the computational difficulties . . . are assessed . Human-like performance, which is prone to include immoral actions, may not be acceptable in machines, but moral perfection may be computationally unattainable. . . . The development of machines with enough intelligence to assess the effects of their actions on sentient beings and act accordingly may ultimately be the most important task faced by the designers of artificially intelligent automata."
   - Bostrom, Nick (2003), "Are You Living in a Computer Simulation?", *Philosophical Quarterly* 53(211): 243–255.
     - An interesting philosophical analysis not unrelated to Lem's story.
     - Available online, along with many related items, at http://www.simulation-argument.com/
     - See also Rothman 2016
   - Bostrom, Nick (2011), "The Ethics of Artificial Intelligence", in William Ramsey & Keith Frankish (eds.), *Cambridge Handbook of Artificial Intelligence* (Cambridge, UK: Cambridge University Press), http://www.nickbostrom.com/ethics/artificial-intelligence.pdf
     - A good survey article.
   - Chalmers, David J. (2010), "The Singularity: A Philosophical Analysis", *Journal of Consciousness Studies* 17: 7–65, http://consc.net/papers/singularity.pdf
     - "What happens when machines become more intelligent than humans? One view is that this event will be followed by an explosion to ever-greater levels of intelligence, as each generation of machines creates more intelligent machines in turn. This intelligence explosion is now often known as the 'singularity'."
     - §4 ("Obstacles to the Singularity") is relevant to this chapter.
     - See also:
       * Dennett, Daniel C. (2012), "The Mystery of David Chalmers", *Journal of Consciousness Studies* 19(1–2): 86–95, https://ase.tufts.edu/cogstud/dennett/papers/chalmers.pdf
       * McDermott, Drew (2012), "Response to 'The Singularity' by David Chalmers", *Journal of Consciousness Studies* 19(1–2): 167–172, http://www.cs.yale.edu/homes/dvm/papers/chalmers-singularity-response.pdf
       * Chalmers, David J. (2012), "The Singularity: A Reply", *Journal of Consciousness Studies* 19(1–2), http://consc.net/papers/singreply.pdf
   - Choi, Charles Q. (2008), "Not Tonight, Dear, I Have to Reboot", *Scientific American* (March): 94–97, http://www.scientificamerican.com/article/not-tonight-dear-i-have-to-reboot/

  - "Is love and marriage with robots an institute you can disparage? Computing pioneer David Levy doesn't think so—he expects people to wed droids by midcentury. Is that a good thing?"

- Clarke, Roger (1993/1994), "Asimov's Laws of Robotics: Implications for Information Technology", Part 1, *IEEE Computer* 26(12) (December 1993): 53–61; Part 2, *IEEE Computer* 27(1) (January 1994): 57–66; http://www.rogerclarke.com/SOS/Asimov.html
- Foerst, Anne (2001), "Commander Data: A Candidate for Harvard Divinity School?", in Arvind Sharma (ed.), *Religion in a Secular City: Essays in Honor of Harvey Cox* (Harrisburg, PA: Trinity Press International), http://www.slideshare.net/peterbuck/commander-data-a-candidate-for-harvard-divinity-school
  - Analyzes the concept of personhood by reference to whether the fictional android Commander Data, from "Star Trek: The Next Genertion", is a person.
- Hallevy, Gabriel (2013), *When Robots Kill: Artificial Intelligence under Criminal Law* (Northeastern University Press).
  - Reviewed in:
    Wescott, David (2013), "Robots behind Bars", *The Chronicle [of Higher Education] Review* 59(29) (29 March): B17, http://chronicle.com/article/When-Bots-Go-Bad/138009/
  - This is also relevant to Ch. 18
- Heller, Nathan (2016), "Not Our Kind", *New Yorker* (28 November): 87–91, http://www.newyorker.com/magazine/2016/11/28/if-animals-have-rights-should-robots
  - "What moral claims do animals—and robots—make on us? If animals have rights, should robots? We can think of ourselves as an animal's peer—or its protector. What will robots decide about us?"
- Hofstadter, Douglas R.; & Dennett, Daniel C. (1981), "Reflections [on Lem's 'Non Serviam']", in Douglas R. Hofstadter & Daniel C. Dennett (eds.) (1981), *The Mind's I: Fantasies and Reflections on Self and Soul* (New York: Basic Books): 317–320, http://themindi.blogspot.com/2007/02/chapter-19-non-serviam.html
- LaChat, Michael Ray (2003), "Moral Stages in the Evolution of the Artificial Superego: A Cost-Benefits Trajectory", in Iva Smit, Wendell Wallach, & George E. Lasker (eds.), *Cognitive, Emotive and Ethical Aspects of Decision Making in Humans and in Artificial Intelligence*, Vol. II (Windsor, ON, CANADA: International Institute for Advanced Studies in Systems Research and Cybernetics): 18–24.
  - A follow-up essay by LaChat, in which he argues that a "moral" robot "will have to possess sentient properties, chiefly pain perception and emotion, in order to develop an empathetic superego which human persons would find necessary and permissible in a morally autonomous AI".
- LaChat, Michael Ray (2004), " 'Playing God' and the Construction of Artificial Persons', in Iva Smit, Wendell Wallach, & George E. Lasker (eds.), *16th International Conference on Systems Research, Informatics and Cybernetics* (Windsor, ON, Canada: International Institute for Advanced Studies in Systems Research and Cybernetics) 3: 39–44.
- Sandberg, Anders (2015), "Death and Pain of a Digital Brain", *New Scientist* 227(3038) (12 September): 26–27, http://stirling-westrup-tt.blogspot.com/2015/09/tt-ns-3038-anders-sandberg-can-software.html

  - "One day we will create virtual minds. Could this simulation suffer, ... and does it matter?" Compare this to the ending of Lem's story.

- Sparrow, Robert (2004), "The Turing Triage Test", *Ethics and Information Technology* 6(4): 203–213, http://profiles.arts.monash.edu.au/rob-sparrow/download/rsparrow-eit-turingtriage.pdf
  - "[P]ropose[s] a test for when computers have achieved moral standing by asking when a computer might take the place of a human being in a moral dilemma .... [S]et[s] out an alternative account of the nature of persons, which places the concept of a person at the centre of an interdependent network of moral and affective responses, such as remorse, grief and sympathy."
  - See also:
    Sparrow, Robert (2014), "The Turing Triage Test: When Is a Robot Worthy of Moral Respect?", *The Critique*, http://www.thecritique.com/articles/the-turing-triage-test-when-is-a-robot-worthy-of-moral-respect/
- Sparrow, Robert (2007), "Killer Robots", *Journal of Applied Philosophy* 24(1): 62–77.
  - Should artificial intelligences be allowed to kill? "This paper considers the ethics of the decision to send artificially intelligent robots into war ...."
- Tanaka, Fumihide; Cicourel, Aaron; & Movellan, Javier R. (2007), "Socialization between Toddlers and Robots at an Early Childhood Education Center", *Proceedings of the National Academy of Sciences* 104(46) (13 November): 17954–17958, http://www.pnas.org/content/104/46/17954.full
  - "[C]urrent robot technology is surprisingly close to achieving autonomous bonding and socialization with human toddlers for sustained periods of time ..."
- Yampolskiy, Roman V.; & Fox, Joshua (2013), "Safety Enginerring for Artificial General Intelligence", *Topoi* 32(2): 217–226, https://intelligence.org/files/SafetyEngineering.pdf
  - "We will argue that attempts to attribute moral agency and assign rights to all intelligent machines are misguided .... As an alternative, we propose a new science of safety engineering for intelligent artificial agents .... In particular, we challenge the scientific community to develop intelligent systems that have human-friendly values that they provably retain, even under recursive self-improvement."

2. Websites:

   - On Karel Čapek and "R.U.R":
     - "Karel Capek", http://www.catch22.com/SF/ARB/SFC/Capek,Karel.php3
     - "Josef and Karel Capek", http://www.imagi-nation.com/moonstruck/clsc23.html
     - Jerz, Dennis G. (2002), "R.U.R. (Rossum's Universal Robots)" http://jerz.setonhill.edu/resources/RUR/index.html
       * Discussion of various versions of the play, with numerous links.
       * Online versions of the play:
         Čapek, Karel (1920), "R.U.R.: Rossom's Universal Robots", trans. by Paul Selver, http://www.mindfully.org/Reform/RUR-Capek-1920.htm

Čapek, Karel, "R.U.R. (Rossum's Universal Robots)", trans. by David Wyllie, http://ebooks.adelaide.edu.au/c/capek/karel/rur/

- Danaher, John (2013), "Is There a Case for Robot Slaves?", *Philosophical Disquisitions* (22 April), http://philosophicaldisquisitions.blogspot.com/2013/04/is-there-case-for-robot-slaves.html
  - A response to Petersen 2007.

Part VI

Closing Remarks

Chapter 21

Summary

Version of 18 December 2018; DRAFT © 2004–2018 by William J. Rapaport

> Philosophical reflection. . . is not static, and fixed, but ongoing and dynamic. The conflict of opinions not only *isn't* something to worry about, in fact, it is precisely how things ought to be. . . .
>
> For. . . only after you've considered all sides will you be in a meaningful position to choose one—when that time comes to decide.
>
> . . . the philosopher within me cannot make that decision for you. His job, he reminds me, is merely to rouse the philosopher within *you* and to get you thinking—not to tell you what to think.
>
> That's *your* philosopher's job.
>
> —Andrew Pessin, *The 60-Second Philosopher* (London: Oneworld): 3–4.

21.1 Readings:

1. Required:
   - Smith, Brian Cantwell (2002), "The Foundations of Computing", in Matthias Scheutz (ed.), *Computationalism: New Directions* (Cambridge, MA: MIT Press): 23–58.
     - Earlier version online at: http://www.ageofsignificance.org/people/bcsmith/print/smith-foundtns.pdf accessed 16 November 2012.
2. Very Strongly Recommended:
   - DeJohn, Jerry; & Dietrich, Eric (2003), "Editorial: Subvert the Dominant Paradigm! A Review of *Computationalism: New Directions*, edited by Matthias Scheutz", *Journal of Experimental & Theoretical Artificial Intelligence* 15(4) (October-December): 375–382, esp. §3.1, pp. 378–379.
   - Scheutz, Matthias (2002), "Computation, Philosophical Issues about", *Encyclopedia of Cognitive Science* (London: Macmillan): 604–610; online at: http://staffweb.worc.ac.uk/DrC/Courses%202006-7/Comp%204070/Reading%20Materials/scheutz%20Comp%2 accessed 16 November 2012.

Let us take stock of where we have been. MAKE SURE THAT EVERYTHING HERE IS ACTUALLY DISCUSSED IN THE BOOK!

21.2 What Is Philosophy?

We began with a brief history of western philosophy in terms of its relevance to computer science. I offered a definition of philosophy as the personal search for truth, in any field, by rational means. We reviewed some of the principles of rationality, in particular, of argument analysis, looking at how an argument consists of premises and a conclusion, how the conclusion might follow validly from the premises if the argument is truth preserving, and how the argument might be sound if the premises are true. And we briefly looked at the main branches of philosophy, asserting the existence, for any field X of a "philosophy of X". So, the philosophy of computer science would be the study of the main goals and fundamental assumptions of computer science.

21.3 What Is Computer Science?

We then turned to an ontological question: What is computer science? We looked at both political and philosophical motivations for asking the question, and then we examined various answers: (Newell et al., 1967) said that computer science is the science of *computers* and surrounding phenomena such as algorithms. (Knuth, 1974b) said that computer science is the study of *algorithms* and surrounding phenomena such as the computers that they run on. One way of adjudication between these two apparently opposed viewpoints is to take them as being extensionally equivalent—computer science is the study of algorithms *and* the computers that execute them—but intensionally distinct because each focuses on a different portion of computer science's single subject matter.

A slightly different view is taken by (Newell and Simon, 1976), who said that computer science is the "artificial science" (as opposed to the "natural science") of the phenomena surrounding computers, where both kinds of science are empirical studies, with the latter studying phenomena that occur in nature and the former studying human-made artifacts. This can be contrasted with (Shapiro, 2001), who argues that computer science *is* a natural science, but not of computers; rather, it is the natural science of *procedures*.

Others, such as (Hartmanis and Lin, 1992), add to the subject matter the notion of *information*: Computer science is the study of how to represent and process information and of the machines and systems that do this. Still others, such as (Brooks, 1996), say that it is not a science at all, but a branch of engineering. And (Denning et al., 1989), (Denning and Freeman, 2009) say that it is a new kind of science (neither physical, biological, or social) of natural and artificial information processes.

21.4 Is Computer Science a Kind of Science or of Engineering?

So the question of what computer science is breaks down into two smaller questions: What does it study (as summarized in the previous section)? And how does it study it? That is, is it a science? Is it a branch of engineering? Or is it neither—is it some other, or new, kind of study? To answer this, we considered what science and engineering were.

21.4.1 What Is Science?

We looked at two dichotomies in trying to determine what science is: Is its goal to describe, or to explain? And are its descriptions or explanations intended to be about reality ("realism") or merely useful summaries that enable us to make predictions ("instrumentalism")?

And what is the "scientific method"? Is it the experimental method advocated by Bacon in the 1600s and described by Kemeny's "loop"? That is:

while there is a new fact to observe, **do:**
begin
    observe it;
    induce a general hypothesis (to explain or describe it);
    deduce future observations;
    verify your predictions
end.

Or is it what Popper suggested in the 1950s: Science makes conjectures and refutations. A statement is scientific if and only if it is falsifiable.

Or is it what Kuhn suggested in the 1960s: Science proceeds by "paradigm revolutions" alternating with periods of "normal" science?

And what about subjects like mathematics, which seems to be scientific, yet is non-empirical?

21.4.2 What Is Engineering?

Perhaps we need to distinguish between pure and applied sciences; perhaps computer science is an applied science, or a branch of engineering. We looked at (Davis, 1998)'s history of engineering, which suggests that engineering is defined by its curriculum, which teaches how to apply science for the use and convenience of people and to improve the means of production. By contrast, (Petroski, 2003) suggests that the fundamental activity of engineering is *design*. But (Loui, 1987) suggests that computer "science" is a new kind of engineering that studies the theory, design, analysis, and implementation of information-processing algorithms.

21.5 What Does Computer Science Study?

So, whether it is a science, a branch of engineering, or something else, what does it study? Does it study two things: computing and computers? Or are these merely two aspects of a single, underlying subject matter? To answer this, we asked four questions:

21.5.1 What Is a Computer? Historical Answer

We saw that there were two intertwined branches of the history of computers, each of which had a slightly different goal: The goal of one branch was to build a computing machine. This is the history in which prominent roles were played by Pascal, Leibniz, Babbage, Aiken, Atansoff and Berry, Turing, Eckert and Mauchly, and von Neumann, among others. The goal of the other branch was to provide a foundation for mathematics. This is the history in which prominent roles were played by Leibniz (again), Boole, Frege, Russell, Hilbet, Turing (again), Church, and Gödel, among others.

21.5.2 What Is an Algorithm? Mathematical Answer

We began this investigation by asking what computation is. A function f is computable means by definition that there is an algorithm that computes f, that is, that there is an algorithm A such that, for all input i, $A(i) = f(i)$, and such that A specfies how f's input and output are related.

So, what is an algorithm? Roughly, an algorithm for a problem P is, by definition, a finite procedure (that is, a finite set of explicit instructions) for solving P that is (i) unambiguous for the computer or human who will execute it—that is, all steps of the procedure must be clear and well-defined for the executor—and (ii) it must eventually halt, outputting a correct solution to P. Of course, this is only a rough definition: Spelling out exactly what 'clear and well-defined' means was an accomplishment of the highest order.

We looked at the most successful solution to this problem of what an algorith is, namely, the Turing machine, as well as the history of the term 'computable'. We discussed Turing's thesis that a function is computable if and only if it is computable by a Turing machine, as well as Church's thesis that a function is computable if and only if it is lambda-definable. That both of these notions turn out to be logically equivalent to each other as well as to other analyses, such as that of recursive functions, allows us to talk of the Church-Turing Computability Thesis.

21.5.3 What Is a Computer? Philosophical Answer

Armed with the history of computers and the mathematics of computation, we returned to the question of what a computer is. We began our philosophical investigation with (Searle, 1990)'s claim that everything is (interpretable as) a digital computer. And we looked at some alternatives: that a computer is "magic paper" that can take as input patterns that describe changes to themselves and to other patterns, and that causes the described changes to occur (Hayes, 1997), that a computer is a device that changes

values of variables (Thomason, 2003) (see §21.7.2, (C4), below), and that the universe is a computer (Lloyd and Ng, 2004), (Wolfram, 2002b).

21.5.4 What Is an Algorithm? Philosophical Answer

And we then returned to the question of what an algorithm is, now from a philosophical point of view. This divided into three questions:

21.5.4.1 What Is a Procedure?

We saw that (Shapiro, 2001) argued for the more general notion of a "procedure" as being the subject matter of computer science. So we looked at what some philosophers have said about procedures.

(Cleland, 1993) (and in subsequent papers) argued that "mundane" procedures (such as causal processes, including recipes) are effective procedures that are not computable by Turing machines, because their effectiveness depends on conditions in the external world. And (Preston, 2013) pointed out important differences between improvisational recipes (and music) and precise algorithms, suggesting that recipes are more like specifications of programs than they are like computer programs.

21.5.4.2 What Is Hypercomputation?

Next, we looked at the idea that there might be functions that can be computed in some more general sense than by Turing machines, what is called 'hypercomputation'. We looked at Turing's oracle machines, Boolos and Jeffrey's infinitely accelerating, Zeus machines, Wegner's interaction machines" and Putnam's and Gold's "trial and error" machines (which are TMs that can "change their mind": it is the last answer that matters, not the first one), which (Kugel, 2002) argued are necessary in order for AI to succeed.

21.5.4.3 What Is a Computer Program?

This led us to the third part of our investigation of algorithms: their implementation in computer programs. Here we went deep into the depths of the philosophy of computer science, looking at implementation, software, legal protections for software, and the relation of programs to scientific theories and to the real world.

21.5.4.3.1 What Is Implementation? (Chalmers, 2011) (and elsewhere) argued against (Searle, 1990) that implementation is an isomorphism and that a computer is an implementation of a computation or of a universal Turing machine. And (Rapaport, 1999) argued that implementation is the semantic interpretation of an Abstraction in some medium. To understand this, we looked at the syntax and semantics of formal systems.

21.5.4.3.2 What Is Software? Paralleling the computer-computation distinction is the hardware-software distinction. According to (Moor, 1978), one can understand computers as physical objects or on a symbolic level (and we compared these two levels of understanding to (Dennett, 1971)'s three levels: the physical, design, and intentional "stances"). A computer program, for Moor, is a set of instructions that a computer can execute. But the notion of software is relative to both a computer and a person: X is software for computer C and person P if and only if X is a computer program for C that is changeable by P. Hardware is similarly relative: X is hardware for C and P if and only if X is (physically) part of C, and X is not software for C and P.

In contrast, (Suber, 1988) argued that software is simply syntactic form, and (Colburn, 1999) argued that it is a "concrete abstraction" that has an abstract "medium of description" (a text in a formal language) and a concrete "medium of execution" (circuits and semiconductors).

We also looked at how the software-hardware distinction compared to Spinoza's "dual-aspect" theory concerning the mind-brain distinction, and whether any of the interpretations of Spinoza's theory might apply to the computer case.

21.5.4.3.3 Can (Should) Software Be Patented, or Copyrighted? In order to try to understand the software-hardware relationship, we looked at the issue of whether software could, or should, be patented or else copyrighted. After all, if a program is a piece of text, then copyright is appropriate. But if a program is a piece of hardware, then patent is appropriate. But programs seem to be both, yet nothing can (legally) be both patented and copyrighted.

To resolve this paradox, (Newell, 1986) suggested that we need to devise good models ("ontologies") of algorithms and other computational entities. And (Koepsell, 2000) suggested that we need to revise the models of legal protection.

21.5.4.3.4 Are Programs Scientific Theories? As part of our endeavor to understand what a program is, we considered the claim made by several philosophers and computer scientists that some programs are scientific theories.

Programs are (a language for expressing) theories, which can then be their own models. We looked at the differences and relations between theories and models, simulations and "the real thing", and simulation vs. emulation. We also briefly looked at philosophical theories of scientific explanation and of scientific models.

21.5.4.3.5 What Is the Relation of Programs to the World? A final question related to what computer programs are is how they relate to the real world. Here, we looked at (Smith, 1985), who argued that there is a gap between the world and our models of it, and that computers rely on models of the world, but must act in the real world.

One related issue here concerns (Cleland, 1993)'s views about the Church-Turing Computability Thesis: Could the computability of a problem depend in part on the real world, and not exclusively on the program for solving it?

Another related issue concerns whether Smith's observations pertain only to computers or also to us humans. After all, we must act in the real world, but perhaps our

actions are also based on mental models of the world that are separate from the real world.

21.6 Computer Ethics

We looked at two topics in computer ethics.

21.6.1 Are There Decisions Computers Should Never Make?

Given the limitations on the verifiability of program correctness, it becomes important to ask this question, first asked in (Moor, 1979).

His answer has two parts: First, no, there are no decisions that computers should never make *as long as their track record is better than that of humans*. After all, the question seems to be logically equivalent to this one: Are there decisions that should not be made rationally? Presumably, computer programs that make decisions make them on the basis of rational evaluation of the facts. And surely we always want to make our decisions rationally.

Second, it is up to us to accept or reject the decisions made by a computer. This is the case no more and no less than it is for decisions made (for us) by others. Thus, the decision made by a computer should be evaluated in the same way as advice offered by an expert or in a textbook.

On the other hand, (Friedman and Kahn, 1997) argue that, yes, there *are* decisions that computers should never make, on the grounds that only humans are capable of being moral agents. Whether that is really the case is part of the second issue in computer ethics (see §21.6.2). But to err is human, as shown in the case of the airline crash caused by following a *human's* decision instead of a computer's. This must be contrasted with cases in which tragedy occurred by blindly following a *computer's* decision, as in the radiation scenario.

21.6.2 Should We Build an Artificial Intelligence?

The second issue in computer ethics that we looked at was, given the fact that it is plausible that we *could* build an AI, it becomes important to ask whether we *should* build one.

(Lem, 1971) pointed out that, if we do succeed in building AIs, we may someday have to pull the plug on them. And (LaChat, 1986) suggested that maybe we shouldn't even begin. But (LaChat, 1986) also argued that considering the possibility of building one and considering the moral consequences of doing so enables us to deal with important issues such as: What is a person? Would an AI with personhood have (moral) rights? (See §21.6.1, above.) And could the AI itself be a moral agent?

21.6.3 Philosophy of AI

Our final topic concerned the philosophy of artificial intelligence, in which we looked at the Turing test and the Chinese Room Argument.

(Turing, 1950)'s article that introduced what is now called the Turing test suggested that a computer will be *said* to be able to think if we cannot distinguish its cognitive behavior from that of a human.

(Searle, 1980)'s rebuttal argued that a computer could pass a Turing test without really being able to think.

We then looked at how "multiple implementation" and how syntax can suffice for semantic interpretation of the kind needed for computational cognition might overthrow Searle's position.

21.7 What Is the Philosophy of Computer Science?

Let's now return to our original question and examine it in the light of an essay directed to that issue by Brian Cantwell Smith, (Smith, 2002).[1]

21.7.1 The Major Question

The major question in the philosophy of computer science, according to Smith, is: What is computation? And the answer is that it is a lot more than just the theory of Turing machines.

Smith says that computing is:

empirical: "Computation in the wild" includes every aspect of the nature and use of computing and computers, presumably including things like iPads and Facebook.

conceptual: Computing and computers are intimately related to issues in semantics such as interpretation and representation. This is probably where the theory of Turing machines fits in.

cognitive: The theory of computation must provide a foundastion for a theory of mind (that is, for what I have called computational cognitive science). This is probably where the Turing test and the Chinese Room Argument fit in.

21.7.2 9 Theses about Computation

Smith offers 9 "theses" about computation, each of which is worthy of a book-length discussion. Here, I will present them, not in the order in which Smith does, but in what seems to me to be a more logical order (but I retain Smith's numbering scheme).

(C2) *A theory of computation needs a full theory of semantics and intentionality.*

By 'intentionality', I think Smith means what the 19th-century cognitive scientist Franz Brentano meant, namely, "directedness to an object", that is, "being about something".[2]

[1] For a critique of (Smith, 2002), see (DeJohn and Dietrich, 2003), esp. §3.1, pp. 378–379.

[2] On intentionality, see (Jacob, 2019). On Brentano, see (Huemer, 2010). On "directedness" and "aboutness", see (Rapaport, 2012a).

(C1) *However, none of the following provide such a theory:*

1. **effective computability:** the study of what can, can't, or is impractical to be done by an abstract machine
2. **execution of algorithms, or rule following**—including recipes
3. **calculation of functions:** the behavior of taking input and producing output
4. **digital state machines:** that is, automata with finite sets of states
5. **information processing:** that is, storing, manipulating, and displaying "information".[3]
6. **physical symbol systems:** that is, the claim that a computer's interaction with symbols depends on its physical implementation
7. **formal symbol manipulation** by a machine without regard to meaning:

   (C3)
   - *Formal symbol manipulation separates the syntactic and semantic domains.*
   - *But real-world computational processes are "participatory"*

   That is, the syntax and semantics of real-world computational processes intersect in two ways:

   (a) As argued in (Smith, 1985), computers get their input from, and must act in (that is, produce output to), the real world.

   (b) Computers are part of the real world: 1. Some computer-*internal* symbols refer to other computer-internal symbols; others refer to external objects.
   2. Some computer-*external* symbols refer to computer-*internal* ones; others refer to other external ones.

   (C6) *Computation is not formal in any of the following senses of 'formal':*
   - anti-semantic: the sense in which syntax is independent of semantics
   - syntactic: for example, proof theory only relies on pattern matching and symbol manipulation.
   - determinate or well-defined (that is, not vague): for example, digital or discrete
   - mathematical: I think Smith has in mind here the kind of formality that Bertrand Russell was discussing in this remark:

     "Pure mathematics consists entirely of such assertions as that, if such and such a proposition is true of *anything*, then such and such another proposition is true of that thing. It is essential not to discuss whether the first proposition is really true, and not to mention what the anything is, of which it is supposed to be true.

[3] On various meanings of 'information', see (Piccinini and Scarantino, 2011) and http://en.wikipedia.org/wiki/Information

> . . . *If* our hypothesis is about *anything*, and not about some one or more particular things, then our deductions constitute mathematics. Thus **mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.** (Russell, 1917, Ch. 4, p. 75; my boldface)[4]

- analytic method: This might be related to (Simon, 1962)'s theory of how to analyze complexity "hierarchically", that is, recursively.

(C4) *Turing machines don't use marks to represent numbers; they use numbers to represent marks!*

- Recall that syntax concerns mark-manipulation, as we saw in CITE. (This is the kind of mathematics that Russell was discussing in the above quote.) The real world of mathematics concerns number-manipulation. And semantics is the study of the relation between them.
- But, according to Smith,[5] "In science, we use numbers to represent 'stuff'; computation only studies marks".

A corollary is this: *The theory of computation is "a theory [that is, a language] of how. . . patches of the world in one physical configuration [can] change into another physical configuration"* (Smith, 2002, 42). (Compare this to Thomason's definition of a computer, §21.5.3, above.) That is:

(C5) *Computability theory is a mathematical theory of the flow of causality.*

(C7) *A theory of computation needs a full theory of ontology.* (For a beginning, see (Smith, 1996). For critical reviews, see (Koepsell, 1998), (Loui, 1998).)

(C8)
- *Computers are not natural kinds.* (But recall that (Shapiro, 2001) argued that *procedures* are natural kinds.)
- *There can be no theory of computation.*

(C9) *Therefore, the existence of computation is extremely important, because any theory of it will be a theory of intentional artifacts, hence a theory of everything!*

And that seems to be a good note on which to end.

[4] The passage humorously concludes: "People who have been puzzled by the beginnings of mathematics will, I hope, find comfort in this definition, and will probably agree that it is accurate."

[5] In a lecture given at UB in 20xx??.

> So many people today—and even professional scientists—seem to me like somebody who has seen thousands of trees but has never seen a forest. A knowledge of the historic and philosophical background gives that kind of independence from prejudices of his generation from which most scientists are suffering. This independence created by philosophical insight is—in my opinion—the mark of distinction between a mere artisan or specialist and a real seeker after truth. —Albert Einstein, 1944; cited in Howard 2017

21.8 NOTE FOR NEXT DRAFT

1. See discussion of "real" computers, which are interaction machines, with "theoretical" computers, in §11.4.3, above.

Chapter 22

Syntax and Semantics

Version of 9 December 2017;[1] DRAFT © 2004–2017777777lliam J. Rapaport

USE Rapaport 2017b!
ALSO: http://www.cse.buffalo.edu/~rapaport/563S05/synsem.html,
BUT ALSO USE Chs. 2 and 3 from
http://www.cse.buffalo.edu/~rapaport/Papers/book.pdf and their published version:
ALSO NOTE: Formalization

> makes it possible to abstract from the meaning of the symbols and to regard the proving of theorems (of formal logic) as *a game played with marks on paper according to a certain arbitrary set of rules.* (Church, 1933, 842)

- Rapaport, William J. (1995), "Understanding Understanding: Syntactic Semantics and Computational Cognition", in James E. Tomberlin (ed.), *AI, Connectionism, and Philosophical Psychology*, *Philosophical Perspectives* Vol. 9 (Atascadero, CA: Ridgeview): 49-88; reprinted in Toribio, Josefa, & Clark, Andy (eds.) (1998), *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory*, *Artificial Intelligence and Cognitive Science*, Vol. 4: Conceptual Issues (New York: Garland): 73–88.

And Ch. 3 of Understanding Understanding,
http://www.cse.buffalo.edu/~rapaport/Papers/book.pdf
And also use (Suber, 1997a).

[1] Although this chapter is not yet written, it will probably be adapted from Rapaport 2017b.

Figure 22.1: need ©info

Version of 9 December 2017; DRAFT © 2004–2017 by William J. Rapaport

This Is Not the End

Lots of things never end. Space. Time. Numbers. The questions little kids ask.[2]

And philosophy.

You try to convince somebody of something—even yourself—by offering reasons to believe the thing. But then your belief is only as valid as your reasons are, so you offer reasons to accept your reasons. But then those reasons need *further* reasons and you're off. As a result it often seems that there aren't any answers to philosophical questions: there are just more arguments, more objections, more replies. And so it may easily seem that it's not worth even getting started. Why bother? You'll never finish. You may as well try to count all the numbers.

But there is another way of thinking about it.

I went snorkeling for the first time a few years ago. It was an amazing experience. There was a whole world under that water to which I'd been oblivious my entire life. This world was populated with countless amazing creatures with all sorts of complex relationships to each other in that tangled ecosystemic way. Indeed every single thing was connected to every other thing: this one is food for that one, which excretes chemicals used by another one, which excretes waste products used by others, and so on. Stunning, fascinating, and absolutely, deeply, beautiful. It had been there all along, jsut waiting for me to dive in.

If you were now to tell me that that ocean goes on forever, filled with ever more amazing creatures in more amazing relationships—I wouldn't say, "Well then why bother entering?" Rather, I'd say, "Where can a guy get a wetsuit around here?"

But that is philosophy. It's filled with countless amazing ideas, concepts, beings, which exist in all sorts of complex logical relationships with each other. And unlike the actual ocean this one *is* infinitely deep: Wherever you enter you can keep going, and going, and going. What you should be thinking, then, is not: "Why enter?" It is, rather, this: thank you—very much.

But of course, that world just is *this* world, the world that you're in. This great ocean you may be looking for, you're alrady in it. You just have to start thinking about it. The very first drop in that bucket is a splash into the infinite.

This is the beginning.

[2] The cells on a Turing-machine tape. Infinite loops.—items added by WJR.

—Andrew Pessin, *The 60-Second Philosopher* (London: Oneworld): 124–125.

Part VII

Appendices

Appendix A

Argument Analysis and Evaluation

Version of 9 December 2017; DRAFT © 2004–2017 by William J. Rapaport

> [Sidney Harris cartoon showing boy sitting in front of a computer, thinking:]
>
> Liebniz, Boole and Gödel worked with logic.
> I work with logic.
> I am Leibniz, Boole and Gödel.
> — *American Scientist* 73 (January-February): 19.

A.1 Readings:

- Required:
  - Chudnoff, Elijah (2007), "A Guide to Philosophical Writing", (Cambridge, MA: Harvard Center for Expository Writing).
    - Accessed 21 March 2014 from: http://www.fas.harvard.edu/∼phildept/files/GuidetoPhilosophicalWriting.pdf

A.2 Introduction

In §2.3, I said that the methodology of philosophy involved "rational means" for seeking truth, and in §2.6, we looked at different kinds of rational methods. In this appendix, we'll look more closely at one of those methods: argument analysis and evaluation.

Unless you are gullible, willing to believe everything you read or anything that an authority figure tells you, you should want to know *why* you should believe something that you read or something that you are told. If someone tells you that some proposition C is true because some other propositions R_1 and R_2 are true, you might then consider, first, whether those *reasons* (R_1 and R_2) really do support the *conclusion* C and, second, whether you believe the reasons.

So, let's consider how you might go about doing this.

A.3 A Question-Answer Game

Consider two players, Q and A, in a question-answer game:

- Q asks a question (see step A.3, below).
- A gives an alleged answer (see step A.3, below).
- Q must verify A's alleged answer (see steps A.3–A.3, below).

Step 1 Q asks whether C is true.

Step 2 A answers: "C, because R_1 and R_2."

> That is, A gives an *argument* for conclusion C with reasons (also called '*premises*') R_1 and R_2.
>
> Note, by the way, that this use of the word 'argument' has nothing directly to do with the kind of fighting argument that you might have with your roommate; rather, it's more like the *legal* arguments that lawyers present to a jury.
>
> Also, for the sake of simplicity, I'm assuming that A gives only two reasons for believing C; in a real case, there might be only one reason (for example: Fred is a computer scientist; therefore; someone is a computer scientist), or there might be more than two reasons (for examples, see any of the arguments for analysis and evaluation in Appendix B).

Step 3 Q must *analyze* or "verify" A's arguments. Q can do this by asking three questions:

- Do I believe (that is, do I agree with) R_1?
- Do I believe (that is, do I agree with) R_2?
- Does C follow *validly* from R_1 and R_2?

There are a few comments to make about these steps. First, steps A.3 and A.3 are *recursive*: For each of them, Q could ask A (or someone else!) whether R_i is true, and A (or the other person) could give an argument for conclusion R_i with new premises R_3 and R_4. Clearly, this process could continue (this is what toddlers do when they continually ask their parents "Why?"!). It is an interesting philosophical question, but fortunately beyond our present scope, to consider where, if at all, this process might stop.

Second, to ask whether C follows "validly" from the premises is to assume that A's argument is a *deductive* one. For the sake of simplicity, all (or at least most) of the arguments in Appendix B are deductive, but, in real life, most arguments are not completely deductive, or not even deductive at all. So, more generally, in Step A.3, Q should ask whether C follows *rationally* from the premises: If it does not follow deductively, does it follow inductively? Abductively? And so on.

Third, unlike the first steps of considering the truth value of the premises, this step of determining whether the relation between the premises of an argument and its conclusion is a rational one is *not* similarly recursive, on pain of infinite regress. The classical source of this observation is due to Lewis Carroll (of "Alice in Wonderland" fame).[1] Carroll was a logician by profession, and wrote a classic philosophy essay involving Achilles and the Tortoise (Carroll, 1895).

Finally, it should be pointed out that the order of doing these steps is irrelevant. Q could first analyze the validity (or rationality) of the argument and then analyze the truth value of (that is, decide whether to agree with) the premises, rather than the other way round.

Step 4 Having *analyzed A*'s argument, Q now has to *evaluate* it, by reasoning in one of the following ways;

- **If** I agree with R_1,
  and if I agree with R_2,
  and if C follows validly (or rationally) from R_1 and R_2,
  then I logically must agree with C.
- But, bf if I really *don't* agree with C,
  then I must reconsider my agreement **either:**
  with R_1,
  or with R_2,
  or with the logic of the inference from $R_1 \& R_2$ to C.
- **If** I *agree* with R_1,
  and if I agree with R_2,
  but the argument is *invalid*, is there a *missing premise*—an extra reason (see below)—that would validate the argument **and** that I would agree with?

[1] More properly known as *Alice's Adventures in Wonderland* and *Through the Looking Glass*.

(i) If so, then I can accept C,
else I should not yet reject C,
but I do need a new argument for (that is, a new set of reasons for believing) C.

- **If** I *disagree* with R_1 or with R_2 (or both),
bf then *this* argument is not a reason *for me* to believe C;
so, I need a new argument for C.

There is one other option for Q in this case: Q might want to go back and reconsider the premises. Maybe Q was too hasty in rejecting them.

- What if Q cannot find a good argument for believing C? Then it might be time to consider whether C is false. In that case, Q needs to find an argument for C's negation: Not-C (sometime symbolized '$\neg C$').

This process of argument analysis and evaluation is summarized in the flowchart in Figure A.1.

A.4 Missing Premises

One of the trickiest parts of argument analysis can be identifying missing premises. Often, this is tricky because the missing premise seems so "obvious" that you're not even aware that it's missing. But, equally often, it's the missing premise that can make or break an argument.

Here's an example from the "Textual Entailment Challenge", a competition for researchers interested in knowledge representation and in information extraction.[2] In a typical challenge, a system is given one or two premises and a conclusion (to use our terminology) and asked to determine whether the conclusion follows from the premise. And "follows" is taken fairly liberally to include all kinds of non-deductive inference.

Here is an example:

Premise 1 (P): Bountiful arrived after war's end, sailing into San Francisco Bay 21 August 1945.

Premise 2: Bountiful was then assigned as hospital ship at Yokosuka, Japan, departing San Francisco 1 November 1945.

Conclusion (C): Bountiful reached San Francisco in August 1945.

The idea is that the two premises might be sentences from a news article, and the conclusion is something that a typical reader of the article might be expected to understand from reading it.

I hope you can agree that this conclusion does, indeed, follow from these premises. In fact, it follows from Premise 1 alone. Premise 2 is a "distractor", in this case.

But what rule of inference allows us to infer C from P?

[2]For more information on "extual entailment" in general, and the Challenge in particular, link to: http://pascallin.ecs.soton.ac.uk/Challenges/RTE3/ (accessed accessed 7 January 2013).

- *P* talks of "arrival" and "sailing into", but *C* talks only of "reaching"
- *P* talks of "San Franciso Bay", but *C* talks only of "San Francisco"

There are no logical rules that connect these concepts.

Most people, I suspect, would think that no such rules would be needed; after all, isn't it "obvious" that, if you arrive somewhere, then you have reached it? And isn't it "obvious" that San Francisco Bay must be in San Francisco?

Well, maybe. But, whereas *people* might know these things, computers won't, unless we tell them. In other words, computers need some lexical knowledge and some simple geographical knowledge. (If you don't like the word 'knowledge' here, you can substitute 'information'.)

So, we need to supply some extra premises that link *P* with *C* more closely. These are the "missing premises". And the argument from *P* to *C* is called an 'enthymeme' (because the missing premises are "in" (Greek 'en-') the arguer's "mind" (Greek 'thymos')).

So, we might elaborate the argument as follows (there are other ways to do it; this is one that comes to my mind):

***P*:** Bountiful arrived after war's end, sailing into San Francisco Bay 21 August 1945.

P_a**:** If something sails into a place, then it arrives at that place.

C_1**:** $\therefore$ Bountiful arrived at San Francisco Bay 21 August 1945.

In this first step, I've added a missing premise, P_a, and derived an intermediate conclusion C_1. Hopefully, you agree that C_1 follows validly (or at least logically in some way, that is, rationally) from *P* and P_a.

We have no way of knowing whether *P* is true, and must, for the sake of the argument, simply assume that it is true. (Well, we could look it up, I suppose; but we're not asked if the argument is "sound" (see below), only if it is "valid": Does *C* follow from *P*?)

P_a, on the other hand, doesn't have to be accepted at all; after all, *we* are imposing it on the (unknown) author of the argument. So, we had better impose something that is likely to be true. P_a is offered as part of the meaning of "sail into". I won't defend its truth any further here, but if you think that it's *not* true, then you should either reject the argument or else find a better missing premise.

We might have chosen a slightly different missing premise:

P_b**:** If something arrives in a place named '*X* Bay', then it arrives at a place named '*X*'.

C_2**:** $\therefore$ Bountiful arrived at San Francisco 21 August 1945.

C_2 will follow from C_1 and P_b, but is P_b true? Can you think of any bays named '*X* Bay' that are not located in a place named '*X*'? If you can, then we can't use P_b. Let's assume the worst: Then we'll need something like:

P_b**.1:** If something arrives in San Francisco Bay, then it arrives at San Francisco.

C_2 will follow from C_1 and P_b.1, and we can easily check the likely truth of P_b.1 by looking at a map.

So far, so good. We've now got Bountiful arriving at San Francisco on 21 August 1945. But what we need is Bountiful "*reaching*" San Francisco in August 1945. So let's add:

P_c**:** If something arrives somewhere, then it reaches that place.

Again, this is proposed as an explication of part of the meaning of 'arrive', and, in particular, of that part of its meaning that connects it to C.

From P_c and C_2, we can infer:

C_3**:** Bountiful reached San Francisco 21 August 1945.

Are we done? Does $C_3 = C$? Look at them:

C_3**:** Bountiful reached San Francisco 21 August 1945.

C**:** Bountiful reached San Francisco in August 1945.

Think like a computer! What do you need to know in order to know whether $C_3 = C$?

You need to know whether the final missing premise, P_d, is true:

P_d**:** If something occurs (on) DATE MONTH YEAR, then it occurs *in* MONTH YEAR.

And that's true by virtue of the way (some) people talk. From P_d and C_3, we can infer C.

So, the simple argument that we started with, ignoring its irrelevant premise, becomes this rather more elaborate one:

P**:** Bountiful arrived after war's end, sailing into San Francisco Bay 21 August 1945.

P_a**:** If something sails into a place, then it arrives at that place.

C_1**:** ∴ Bountiful arrived at San Francisco Bay 21 August 1945.

P_b**:** If something arrives in a place named 'X Bay', then it arrives at a place named 'X'.

(or P_b.1: If something arrives in San Francisco Bay, then it arrives at San Francisco.)

C_2**:** ∴ Bountiful arrived at San Francisco 21 August 1945.

P_c**:** If something arrives somewhere, then it reaches that place.

C_3**:** ∴ Bountiful reached San Francisco 21 August 1945.

P_d**:** If something occurs (on) DATE MONTH YEAR, then it occurs *in* MONTH YEAR.

C**:** ∴ Bountiful reached San Francisco in August 1945.

A.5 When Is an Argument a "Good" Argument?

As we have seen, *Q* needs to do two things in analyzing and evaluating an argument:

1. deciding whether the premises are true (that is, deciding whether to agree with, or believe, the premises), and

2. deciding whether the inference (that is, the reasoning) from the premises to the conclusion is a good one.

That is, there are two separate conditions for the "goodness" of an argument:

1. factual goodness

2. logical goodness

Factual goodness—truth—is beyond the scope of logic, although it is definitely not beyond the scope of deciding whether to accept the conclusion of an argument. As we saw in §2.4, there are several ways of defining 'truth' and of determining whether a proposition is true. Two of the most obvious (though not the simplest to apply!) are (1) constructing a (good!) argument for a proposition whose truth value is in question and (2) making an empirical investigation to determine its truth-value (for instance, performing some scientific experiments or doing some kind of scholarly research).

Logical goodness (for deductive arguments) is called 'validity'. I will define this in a moment. But, for now, note that these two conditions must both obtain for an argument to be "really good": A "really good" argument is said to be *sound*—an argument is sound if and only if it is both valid and "factually good", that is, if and only if it is both valid and all of its premises really are true.

Just to drive this point home: If the premises of an argument are all true (or if you believe all of them), that by itself does not make the argument sound ("really good"). For one thing, your belief in the truth of the premises might be mistaken. But, more importantly, the argument might not be valid.

And, if an argument is valid, that by itself does not make the argument sound ("really good"). All of its premises need to be true, also; that is, it needs to be factually good.

So, what does it mean for a (deductive) argument to be valid? An argument **is valid** $=_d ef$ it is "truth-preserving". That is, an argument is valid if and only if, whenever all of its premises are true, then its conclusion must also be true. Here's another way to put it: An argument is valid if and only if it is impossible for all of its premises to be simultaneously true, yet its conclusion is false. Note that none of these formulations say that the premises *are* true. They only say that, *if* the premises *were to be* true, then the conclusion would *preserve that truth*—it would "inherit" that truth from the premises—and so it would also (have to) be true.

That's fine as far as it goes, but it really isn't very helpful in deciding whether an argument really is valid. How can you tell if an argument is truth-preserving? There is a simple, recursive definition, but, to state it, we'll need to be a bit more precise in how we define an argument.

Definition 1: An **argument from premises** $P_1, \ldots, P_n$ **to conclusion** C is$_{def}$ a *sequence* of propositions $\langle P_1, \ldots, P_n, C \rangle$, where C is *alleged* to follow logically from the P_i.

Definition 2: An argument **is valid** (or: **is truth-preserving**) if and only if it is an argument in which each proposition P_i or conclusion C either:

- is *a tautology*
- or is *a premise*
- or follows validly from previous propositions in the sequence by one or more "rules of inference".

This needs some commenting! First, what is a "tautology"? A **tautology** is a proposition that *must always be true*. How can that be? Most tautologies are (uninformative) "logical truths", such as 'Either *P* or not-*P*', or 'If *P*, then *P*'. (Note that, if *P* is true (or, if you believe *P*), then 'Either *P* or not-*P*' has to be true (or, you are logically obligated to also believe 'Either *P* or not-*P*'), and, if *P* is false, then not-*P* is true, and so 'Either *P* or not-*P*' still has to be true (or, you are logically obligated to also believe 'Either *P* or not-*P*'); so, in either case, the disjunction has to be true (or, you are logically obligated to believe it. Similar considerations hold for 'If *P*, then *P*'.) Sometimes, statements of mathematics are also considered to be tautologies (whether they are "informative" or not is an interesting philosophical puzzle; see (Wittgenstein, 1921)).

Second, a premise is one of the initial reasons given for *C*, or one of the missing premises added later. Premises, of course, need not be true, but, when evaluating an argument for validity, we must *assume* that they are true "for the sake of the argument".

Third, clause A.5 might look circular, but it isn't; rather, it's recursive. In fact, this entire definition is recursive. The base cases of the recursion are the first two clauses: Tautologies must be true, and premises are assumed to be true. The recursive case consists of "rules of inference", which are "sub-proofs" that are clearly valid (truth-preserving) when analyzed by means of truth tables.

So, what are these "primitive" valid arguments known as 'rules of inference'? The most famous is called 'Modus Ponens':

> From *P*
> and 'If *P*, then *C*',
> you may validly infer *C*.

Why may you validly infer *C*? Because the truth table for 'If *P*, then *C*' says that that conditional proposition is false in only one circumstance: when its "antecedent" (*P*) is true and its "consequent" (*C*) is false; in all other circumstances, the conditional proposition is true. So, if the antecedent of a conditional is true, and the conditional itself is true, then its consequent must also be true. Modus Ponens preserves truth.

Another important rule of inference is called 'Universal Elimination' (or 'Universal Instantiation'):

> From 'For all x, $F(x)$',
> you may infer $F(a)$, for any individual a in the domain of discourse (that is, in the set of things that you are talking about).

A truth-table analysis won't help here, because this is a rule of inference from "first-order predicate logic", not from "propositional logic", and the formal definition of truth for first-order predicate logic is beyond our scope, but it should be pretty obvious that, if it is true that everything in the domain of discourse has some property F, then it must also be true that any particular thing in the domain (say, a) has that property. (For more rules of inference and for the formal definition of truth in first-order predicate logic, see any good introductory logic text.)

NEED A SECTION WITH EXAMPLES OF VALID/SOUND/INVALID/UNSOUND ARGS.

A.6 Summary

So, to analyze an argument, you must identify its premises and conclusion, and supply any missing premises to help make it valid. To evaluate the argument, you should then determine whether it is valid (that is, truth preserving), and then decide whether you agree with its premises.

If you agree with the premises of a valid argument, then you are logically obligated to believe its conclusion. If you don't believe it's conclusion, even after your analysis and evaluation, then you need to revisit both your evaluation of its validity (maybe you erred in determining its validity) as well as your agreement with its premises: If you really *disagree* with the conclusion of a valid argument, then you must (logically) *disagree* with at least one of its premises.

You should try your hand at analyzing and evaluating the arguments in Appendix B!

A.7 NOTES FOR NEXT DRAFT

1. MISCELLANEOUS THOUGHTS ABOUT PROOFS

   (a) There are lots of ways to have *invalid* arguments!

   (b) More importantly, it is possible to have an *invalid* argument whose conclusion is *true*. Here's an example:

```
All birds fly.                (true)
Tweety the canary flies.      (true)
-----------------------
Therefore, Tweety is a bird. (true)
```

   This is invalid, despite the fact that both premises and the conclusion are true: It is invalid, because the argument form can have true premises and a *false* conclusion:

$$\begin{aligned} &\forall x[P(x) \rightarrow Q(x)] \\ &R(a) \wedge Q(a) \\ &\therefore P(a) \end{aligned}$$

Here's a counterexample, that is, an argument with this form that has true premises but a false conclusion:

```
All birds fly.
Bob the bat flies.
------------------
Therefore, Bob is a bird.
```

Just having a true conclusion doesn't make an argument valid. And such an argument doesn't prove its conclusion (even though the conclusion is true).

(c) Can *any* proposition (or its negation) be proved?

That is, given a proposition P, we know that either P is true or else P is false (that is, that $\neg P$ is true). So, whichever one is true should be provable. Is it?

No!

First, there are propositions whose truth value we don't know *yet*. For example, once upon a time, no one knew if Fermat's Last Theorem could be proved. That's the proposition that the equation

$$x^n + y^n = z^n$$

has no solutions for integers x, y, z and integer $n > 2$. But it has now been proved.[3]

For another example, no one knows (yet) if Goldbach's Conjecture is true. Goldbach's Conjecture says that all positive even integers are the sum of 2 primes; for example, $28 = 5 + 23$. Expressed in first-order logic, this would be:

$$\forall x[\text{Integer}(x) \wedge (x > 0) \wedge \text{Even}(x) \rightarrow \exists y \exists z[\text{Prime}(y) \wedge \text{Prime}(z) \wedge (x = y + z)]]$$

For yet another example, no one knows (yet) if the Twin Prime Conjecture is true. The Twin Prime Conjecture says that there are an infinite number of "twin" primes, that is, primes m, n such that $n = m + 2$; for example, 2 and 3, 3 and 5, 5 and 7, 9 and 11, 11 and 13, etc. Expressed in first-order logic, this would be:

$$\forall m \forall n[\text{Prime}(m) \wedge \text{Prime}(n) \wedge (n = m + 2) \rightarrow \exists x \exists y[\text{Prime}(x) \wedge \text{Prime}(y) \wedge (y = x + 2) \wedge (y > m) \wedge (x > n)]]$$

Second—and much more astounding than our mere inability so far to prove or disprove any of these conjectures—there are propositions whose truth

[3] See http://en.wikipedia.org/wiki/Fermat's_Last_Theorem

value is known to be true, but which we can prove that we cannot prove! This is the essence of Gödel's Incompleteness Theorem. Stated informally, it asks us to consider this proposition, which is a slight variation on the Liar Paradox (that is the proposition "This proposition is false": If it's false, then it's true; if it's true then it's false):

(G) This proposition is true but unprovable.

We can assume that *G* is either true or false. Is it false? If it is false, then it is provable. But any provable proposition has to be true (because proofs are truth-preserving). So it isn't false. Therefore, it must be true. But if it's true, then it's unprovable. End of story; no paradox!

That is, there are true propositions (moreover, Gödel showed that they are propositions that are true in the mathematical system consisting of first-order predicate logic plus Peano's axioms; that is, they are true propositions of arithmetic!) that cannot be proved. (For more information on Gödel's proof, see (Nagel et al., 2001), (Hofstadter, 1979), and (Franzén, 2005).)

2. Application of Predicate Logic to AI
Here's an application of predicate logic to artificial intelligence (AI).

In the late 1950s, one of the founders of AI, John McCarthy, proposed a computer program to be called "The Advice Taker", as part of a project that he called "programs with common sense". (McCarthy is famous for at least the following things: He came up with the name 'artificial intelligence', he invented the programming language Lisp, and he helped develop time sharing. For more information on him, see:
http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)
and
http://aitopics.org/search/site/John%20McCarthy)

The idea behind The Advice Taker was that problems to be solved would be expressed in a predicate-logic language (only a little bit more expressive than first-order logic), a set of premises or assumptions describing required background information would be given, and then the problem would be solved by logically deducing it from the assumptions.

He gave an example: getting from his desk at home to the airport. It begins with premises like

at(I,desk)

meaning "I am at my desk", and rules like

$$\forall x \forall y \forall z [\text{at}(x,y) \wedge \text{at}(y,z) \rightarrow \text{at}(x,z)],$$

which expresses the transitivity of the "at" predicate, and slightly more complicated rules (which go slightly beyond the expressive power of FOPL) like:

$$\forall x \forall y \forall z [\text{walkable}(x) \wedge \text{at}(y,x) \wedge \text{at}(z,x) \wedge \text{at}(\text{I},y) \rightarrow \text{can}(\text{I}, \text{go}(y,z,\text{walking}))];$$

that is, if x is walkable, and if y and z are at x, and if I am at y, then I can go from y to z by walking.

The proposition to be proved from these (plus lots of others) is:

want(at(I,airport))

To see it all worked out, take a look at (McCarthy, 1959).

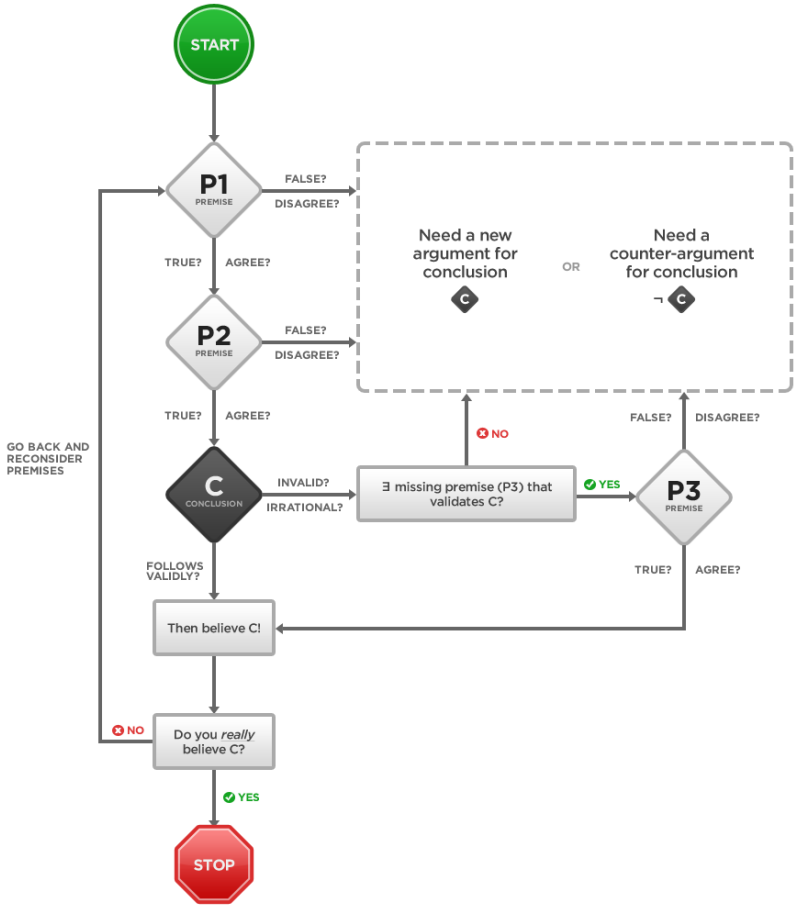


Figure A.1: This flowchart uses '*Pi*' instead of R_i, as the text does. Also, the symbol '*exists*' should be read: "Does there exist".

Appendix B

Position-Paper Assignments

Version of 9 December 2017 DRAFT © 2004–2017 by William J. Rapaport

B.1 Readings:

- Required:
  - Chudnoff, Elijah (2007), "A Guide to Philosophical Writing", (Cambridge, MA: Harvard Center for Expository Writing).
    - Accessed 21 March 2014 from: http://www.fas.harvard.edu/∼phildept/files/GuidetoPhilosophicalWriting.pdf

B.2 Introduction

One of the best ways to learn how to do philosophy and, perhaps more importantly, to find out what your beliefs are about important issues (as well as what your *reasons for your beliefs* are!) is to write about them and then to discuss what you've written with others who have also thought and written about about the issues—your "peers".

So, the writing assignments take the form of "position papers", in which you will be:

- presented with a logical argument about a topic in the philosophy of computer science,
- asked to analyze and evaluate the argument,
- given an opportunity to clarify and defend your analysis and evaluation,
- and simultaneously be asked to help your peers clarify and defend *their* analyses and evaluations of the same argument in an exercise called "peer editing".

This should help you to clarify your *own* position on the topic.

The arguments that are presented in the following sections are those that I have used when I have taught this course. Instructors are invited to modify these or to create their own arguments. The assignments can be scheduled at the convenience of the instructor, which is why I have collected them in this appendix rather than placing them throughout the text. In general, however, I suggest putting each assignment approximately one week *after* the relevant topic has been covered in class. This way, the students will have the benefit of having thought about the readings before forming their own opinions. An alternative, of course, is to schedule them one week *before* the topic is begun to be covered in class, so that the students will be foreced to think about the issues before reading what "experts" have had to say. Another option is to do both: Assign the first draft before the topic is begun, then have the students do the required readings and participate in class discussions of the topic, then follow this with an optional revision or second draft of the position paper and the peer-editing session, with a third draft (or second, if the optional, post-class-discussion draft is omitted) to be turned in for instructor evaluation.

Peer-editing sessions should take up a full class period. The general method is to divide the class into small groups of three or four students (two students will work if the number of students in the class—or latecomers!—demands it, but is not ideal; five-student groups are too large to enable everyone to participate, especially if time is limited). For each student in a group:

1. Have the group read the student's position paper.
2. Have the group challenge the student's position, ask for clarification, and make recommendations for improving the student's paper.

If there are s students in a group and the peer-editing session lasts for m minutes, then the group should spend no more than m/s minutes on each student's paper. The instructor should visit each group at least once to ensure that all is going well, to answer

questions, and to suggest questions if the group seems to be having trouble. If a group ends early and there is a lot of time left in the session, ask each student in the group to join another group (even if only to listen in to that group's ongoing discussion, but if that other group also ended early, then the newcomer should peer-edit one of their papers). Specific information for peer-editing sessions is given with each assignment.

After peer-editing, students should revise their position papers in the light of the editing suggestions and hand in all drafts to the instructor. I usually give the students one week for this revision.

B.3 Position Paper #1: What Is Computer Science?

B.3.1 Assignment

B.3.1.1 Introduction

The purpose of this position paper is to give you an opportunity to clarify *your* beliefs about what computer science is, so that, as we continue to discuss the topic in class, and as you continue to read about it, you'll know where *you* stand—what *your* beliefs are.

Later, when your beliefs have been "contaminated" by further readings and by our discussions, you may wish to *revise* your beliefs. But you can't revise a belief that you don't have (you can only acquire new beliefs). So, here I am forcing you to *discover*, *clarify*, and *defend* the beliefs that you *now* have, by turning them into words and putting them on paper.

B.3.1.2 The Argument

Imagine that you are the newly-appointed Dean of the School of Science at the University of X. In an attempt to build up the rival School of Engineering, the newly-appointed Dean of Engineering has proposed to the Provost (the boss of both deans) that the Department of Computer Science be moved—lock, stock, and computer, so to speak[1]—to Engineering, on the following grounds:

1. Science is the systematic observation, description, experimental investigation, and theoretical explanation of natural phenomena.

2. Computer science is the study of computers and related phenomena.

3. Therefore, computer science is not a science.

(The Dean of Engineering has not *yet* argued that computer science is an engineering discipline; that may come later.)

How do you respond to the Dean of Engineering's argument?

You may agree with it, or not (but there are several ways that might happen; see below).

[1] http://www.worldwidewords.org/qa/qa-loc1.htm

You should ignore political considerations: You may suppose that the move from Science to Engineering involves no loss or gain of money, prestige, or anything else, and it is to be done, if at all, only on strictly intellectual grounds.

The Provost is eagerly awaiting your reply, and will abide by your decision... *if*, that is, you give a well-argued defense of your position.

To formulate and defend your position, you should:

a) Say *whether* you agree that conclusion 3 *logically* follows from premises 1 and 2,

(whether or not you agree with 1 and 2)

and *why* you think it follows or doesn't.

("I agree that conclusion 3 follows from premises 1 and 2 because..." OR "I do not agree that conclusion 3 follows from premises 1 and 2 because...")

- If you think that conclusion 3 *doesn't* follow, is there some (interesting, non-trivial) missing premise (that is, a "missing link" between the premises and conclusion) that would make it follow?[2]

b) Say *whether* you agree with premise 1, and *why* you do or don't.

("I agree with premise 1 because ..." OR "I disagree with premise 1 because ...")

c) Say *whether* you agree with premise 2, and *why* you do or don't.

("I agree with premise 2 because ..." OR "I disagree with premise 2 because ...")

d) If you thought that there were missing premises that validated the argument, say *whether* you agree with them, and *why* you do or don't.

e) If you think that the argument is logically *invalid*, you might still agree or disagree with conclusion 3 independently of the reasons given for it by premises 1 and 2 (and any missing premises).

- If so, state *whether* you agree with 3, and *why*.

It's also possible that you might **neither** agree **nor** disagree with 3; alternatively, you might **both** agree **and** disagree with it.

- For example, you might believe that computer science is *both* a science *and* an engineering discipline (or, alternatively, that it is neither).
- If so, then please give your reasons for this.

[2]For a discussion of missing premises, see
http://www.cse.buffalo.edu/∼rapaport/584/S10/EMAIL/20100118-MissingPremises.txt
"Argument Analysis: Finding Missing Premises". INCLUDE THIS IN THE BOOK SOMEWHERE

And, if you are unsure about any of your answers, try to be very precise about *why* you are unsure and what further information would help you decide.

Other responses:
You might not agree with any of these ways to respond. However, I believe that any other response can, perhaps with a bit of force, be seen to fall under one of the above responses. But if you really feel that your position is not exactly characterized by any of the above responses, then please say:

- what your position is,
- why you believe it,
- and why you think it is not one of the above.

For general assistance on analyzing arguments, see
http://www.cse.buffalo.edu/∼rapaport/584/S10/arganal.html
A Heuristic for Argument Analysis, and the links at
http://www.cse.buffalo.edu/∼rapaport/584/arganalhome.html
Argument Analysis ADD THESE TO THE TEXT

Ground Rules:

1. Your answer should honestly reflect *your* beliefs (not what you think the fictional Provost or Dean of Engineering wants to hear!).
2. If you resort to a dictionary, textbook, article, website, etc., be sure to say which one. Give as much detailed information as you can that would assist someone else to locate the item by themselves. (See the http://www.cse.buffalo.edu/∼rapaport/howtowrite.html#citations "How to Handle Citations" section of my website on "How to Write", for the proper way to do this.)
3. Your position paper should be approximately **1 typed page and double-spaced (that is, about 250 words)** (not including any bibliographic citations).
   - To help keep your paper short, you do not need any fancy introductory paragraph; you can assume that your reader is a fellow student in this course who has just done the same assignment.
   - If you write:
     - 1 paragraph analyzing validity,
     - 1 paragraph each analyzing the premises,
     - and 1 paragraph analyzing the conclusion,

     you will have (more than) enough material.
4. Please bring **5 copies** to **lecture** on the due date.
5. At the top of the page, please put all and only the following information:

   a) the title "Position Paper #1"

b) your name

c) the course you are enrolled in

d) the due date.

(The space taken up by this will not count against your total pages.)

6. For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website http://www.cse.buffalo.edu/∼rapaport/howtowrite.html

DUE AT THE BEGINNING OF LECTURE, 1 WEEK FROM TODAY

B.3.2 Suggestions and Guidelines for Peer-Group Editing

1. When you get into your small groups:
   - introduce yourselves quickly,
   - share copies of your papers with each other,
   - and write each other's names on your paper (so that we have a record of who peer-reviewed whom)
2. Choose one paper to discuss first.
   - Suggestion: Go in alphabetical order by family name.
3. The other people in the group might find it useful to imagine themselves as members of a committee set up by the Provost to make a recommendation. Their purpose is to try to help the author clarify his or her beliefs and arguments, so that they will be able to make a recommendation to the Provost on purely logical grounds (again: ignore politics!).
4. Start by asking the author to state (or read) his or her beliefs about whether computer science is a science, giving his or her *reasons* for those beliefs.
5. Be sure that the author has discussed:

   a) the validity of the argument

   b) the truth value of (or their (dis)agreement with) premise 1

   c) the truth value of (or their (dis)agreement with) premise 2

   d) the truth value of (or their (dis)agreement with) any missing premises.

   e) the truth value of (or their (dis)agreement with) the conclusion

   - And for each of the above, their *reasons*
6. Any time you have a question, ask it. Here are some suggestions:
   - Why did you say _____ rather than _____?

- What did you mean when you said _____?
- Can you give me an example of _____?
- Can you give me more details about _____?
- Do you think that _____ is always true?
- Why? (This is always a good question to ask.)
- How?

7. The author should not get defensive. The committee members are friendly. Critical, but friendly.

8. Keep a written record of the questions and replies. This will be useful to the author, for revision.

9. After spending *about* 10 minutes[3] on the first paper, move on to the next, going back to step 2 above, changing roles. Spend *no more than* 15 minutes[4] per paper (because you've only got about 45 minutes[5] at most). Perhaps one member of the group can be a timekeeper.

10. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"):

    Perhaps defend your claims better, or clarify statements that were misunderstood, etc. For help, see your instructor.

    - At the top of the first page of your revision, please put all and only the following information:
      a) the title "Position Paper #1: 2nd Draft"
      b) your name
      c) the course you are enrolled in
      d) the due date
    - Please **staple** copies of your first draft, (with peer-editing comments, if any) to your second draft.
    - Your second draft should be substantially different from your first draft!

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, DUE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

[3] Actually, some number n of minutes close to 10 but less than or equal to m/s, where m is the total number of minutes in the class, and s is the total number of students in the group.

[4] Actually, $n+5$.

[5] Actually, m.

B.3.3 Grading

B.3.3.1 Philosophy of Grading

To make grading these essays easier on the instructor and easy for students (in case the instructor decides to have students grade each other's essays), I recommend using the "triage" method of grading. On this method, each item to be graded is given:

- full credit (for example, 3 points) if it is clearly done in a completely acceptable manner (even if it might not be entirely "correct")
- partial credit (for example, 2 points) if it is done, but is not clearly worth either full credit or minimal credit
- minimal credit (for example, 1 point) if it is done, but is clearly not done in an acceptable manner.
- no credit (that is, 0 points) if it is omitted.

The advantage to this method of grading is that the grader only has to decide if a response is worth full credit (that is, shows clear understanding of what is expected) or minimal credit (that is, shows clear *mis-* or lack of understanding of what is expected). Any response that is not *clearly* one or the other is given partial credit. And failure to respond, or omission of some requirement, is given no credit. This helps make grading slightly more objective (and certainly easier for novice graders). Details and the theory behind the method are given in Rapaport (2011a) and online at:
http://www.cse.buffalo.edu/∼rapaport/howigrade.html
accessed 19 November 2012.

B.3.4 Grading Rubric

1. Premise 1: Did you state clearly whether you agreed or disagreed with it?

   (Doesn't matter whether you agreed or didn't agree, only with whether you said so.)

   3 pts = clearly stated whether you agreed or not
   2 pts = not clearly stated but implied
   1 pts = stated, but incorrect terminology
   0 pts = did not clearly state whether you agreed

2. Did you give your reasons for your (dis)agreement?

   3 = reasons given, clearly stated, & pertinent
   2 = partial credit: I couldn't decide between 1 & 3
   1 = reasons given, but not clearly stated or not pertinent
   0 = no reasons

3. Premise 2: Did you state clearly whether you agreed or disagreed with it?

   (Doesn't matter whether you agreed or didn't agree, only with whether you said so.)

   0, 1, 2, or 3, as for Premise 1

4. Did you give your reasons for your (dis)agreement?

   0,1,2, or 3, as for Premise 1

5. Valid?

   no answer: 0

   yes; XOR no, but no MP: 1

   no, with wrong MP
   or yes, with right MP: 2

   no, with right MP: 3

   (MP [Missing Premise] = computers are not natural phenomena)

6. Evaluation of MP (agree? why?):

   no evaluation 0
   dis/agree, but no reason 1
   dis/agree, w/ unclear reason 2
   dis/agree, w/ clear/pertinent reason 3

7. Conclusion: Did you state clearly whether you agreed or disagreed with it?

   3 = clearly stated whether you agreed
   2 = not clearly stated, but implied
   0 = did not clearly state whether you agreed

8. Did you give your reasons for your (dis)agreement?

   0,1,2, or 3 points, as for Premise 1

9. Citation style:

   used sources w/o citing -3
   used sources w/ incomp & incorrect citation -2
   used sources w/ incomp XOR incorrect citation -1
   not applicable 0

10. Attached draft 1 & list of peer editors to demonstrate that draft 2 $\neq$ draft 1? 0
Didn't -1

The total is 24 points, which, following my grading theory, maps into letter grades as follows:

A	23–24
A–	22
B+	21
B	19–20
B–	18
C+	17
C	14–16
C–	11–13
D+	9–10
D	5–8
F	0–4

On my grading scheme,
'A' means "understood the material for all practical purposes",
(here, that's 24 pts = 8 parts $\times$ 3 pts full credit)

'B' has no direct interpretation, but comes about when averaging A and C grades

'C' means "average",
(here, that's max 16 pts = 8 $\times$ 2 pts partial credit)

'D' means "did not understand the material", (here, that's max 8 pts = 8 $\times$ 1 pt minimum credit)

'F' usually means "did not do the work" (that is, 0 pts), but can also come about when omitting some parts and doing D work on others.

B.4 Position Paper #2: What Is Computation?

B.4.1 Assignment

For this position paper, please evaluate the following argument:

1. (Knuth, 1973, 4–6) characterizes the informal, intuitive notion of "algorithm" as follows:

   a) "**Finiteness.** An algorithm must always terminate after a finite number of steps..."

   b) "**Definiteness.** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case..."

   c) "**Input.** An algorithm has zero or more inputs..."

   d) "**Output.** An algorithm has one[6] or more outputs..."

[6] My former student Albert Goldfain made the following observation on Knuth's characterization of an algorithm:

> Knuth's characterization of an algorithm includes:
>
> 1d "**Output.** An algorithm has *one* or more outputs..." (my emphasis)
>
> Contrast this with (1c) "*zero* or more inputs" (my emphasis). This is not a typo, and most likely not a mental slip by someone as precise as Knuth. There are several, modern, high-level languages that will allow you to return nothing from a function. For example, consider the *C* function:
>
> ```
> int store-and-return(int x)
>
> int i;
> i=x;
> return;
> ```
>
> In some sense, there are no "outputs" from this function, yet it is still somehow an algorithm. (In the Pascal programming language, there was a distinction between procedures and functions; some other languages call procedures "subroutines". But the point is that these are not functions in the mathematical sense of ordered *pairs*.) In Lisp, which may have been more dominant when Knuth wrote, every function *had* to return at least one value.
> We can, however, re-interpret the "output" of an algorithm so that it is not always the return-value of a function. So, in `store-and-return`, perhaps it is the memory location of the local variable `i` that gets the "output" of the algorithm. In a program like *C*'s `Hello World`,
>
> ```
> main()
> printf("Hello World
> n");
> return;
> ```
>
> perhaps it is the screen that you are printing to that gets the "output" of the algorithm.
> Notice also that if you think of the `store-and-return` function as a "black box" (that is, a device whose internal workings are hidden from you but whose external behavior is observable), there would be no way to tell that it was doing anything. Behaviorally, you would hand an input into the black box and observe nothing else! It may be that Knuth wants to exclude behaviorally unobservable algorithms.

e) "**Effectiveness.** [A]ll of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a [hu]man using pencil and paper. . . "

Note: We can also say that A **is an algorithm for computing a function** f means that A is an algorithm as characterized above and that, for any input i, A's output for $i = f$'s output for i; that is, for any i, $A(i) = f(i)$.

2. Computer programming languages (like Java, Lisp, Fortran, etc.) are formal languages for expressing (or "implementing") algorithms.

3. Every computer programming language is equivalent in expressibility to a Turing-machine programming language.

   a) That is, every program in any programming language can be translated into the language for programming Turing machines, and vice versa.

   b) That is, any function that is computable by any programming language is computable by a Turing machine, and vice versa.

4. Some real computer programs violate parts of definition 1:

   a) For example, airline-reservation systems, ATMs, operating systems, etc., never terminate.

   b) For example, heuristic AI programs don't always compute exactly the function that they were written for, but only come very close. BE SURE TO PUT IN A SECTION ON HEURISTIC PROGRAMS, AND ALSO SITE KAHNEMAN'S BOOK

   c) For example, the "effectiveness" of "mundane" or everyday procedures (like recipes) depends on the environment in which they are executed.

      For example, using a different brand of some ingredient can ruin a recipe, or one chef's "pinch" of salt might be another's 1/8th teaspoon. And can you really execute a recipe "using pencil and paper"?

   d) For example, algorithms can apparently be written that can perform an infinite computation in a finite amount of time (by continually accelerating).

      For example, we can sum the terms of an infinite sequence in a finite amount of time if we take $\frac{1}{2^n}$ second to add the nth term.

And so on.

Bottom line: the "one or more" from premise 1d should not trip you up from analyzing the rest of the argument.

5. Therefore, these programs (that is, the "real programs" referred to in premise 4) do not implement Turing machines (contrary to premise 3).

6. Therefore, they (that is, the "real programs" referred to in premise 4) are not computable. (But how can a real computer program not be computable?!)

To evaluate this argument, you must state whether the argument is **valid**, and you must state whether and why you **agree or disagree with each premise and conclusion**.

- If it is valid, and if you agree with each premise, then you believe that the argument is **sound**.

- You are logically obligated to believe the conclusions of sound arguments!

  So, if you ever come across an argument that you think **is** sound, but whose conclusion you **don't** believe, then either:

  - one (or more) of the premises is false,
  - or the argument is invalid (that is, there is some way for the premises to be true yet for the conclusion to be false),
  - or both.

To determine whether it is valid, you must suppose "for the sake of the argument" that all the premises *are* true, and then consider whether the conclusions logically follow from them.

(Or: Can you imagine some way the world might be so that the premises are true but the conclusion is false?)

Note that, in this argument, there are *two* conclusions: conclusions 5 and 6.

> So, do you agree that conclusion 5 follows logically from premises 1–4 and/or that conclusion 6 follows logically from 5? If not, are there missing premises that are needed to make the argument(s) valid? If there are, do you agree with them (why/why not)?

Next, you must evaluate each premise. Do you agree with it? Why or why not? Finally, do you agree with the conclusion(s)?

> You might agree with a conclusion because you think that the argument is sound;
>
> - if so, say so.
>
> Or you might think that there's something wrong with the argument but agree with the conclusion anyway;
>
> - if so, then try to present a better argument for the conclusion.
>
> Or you might not agree with the conclusion(s);
>
> - if not, state why, and try to give an argument for what you *do* believe.

a) Your position paper should be approximately **1–2 typed pages, double-spaced (that is, about 250–500 words), and single-sided.**

b) Please bring **5 copies** to **lecture** on the due date.

c) **At the top of the first page, please put the following information** *in the following format*:

Position Paper #2, 1st draft	YOUR NAME
DATE DUE	CLASS

d) For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website, "How to Write": http://www.cse.buffalo.edu/∼rapaport/howtowrite.html

As before, this doesn't have to be a beautifully written essay with an abstract. You should just plunge in and evaluate the argument.

But you do need to give full citations to any sources that you cite.

DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.4.2 Suggestions and Guidelines for Peer-Group Editing

1. When you get into your small groups, introduce yourselves quickly, and share copies of your papers with each other.

2. Choose one paper to discuss first. (Suggestion: Go in alphabetical order by family name.)

3. After spending *about* 10–15 minutes on the first paper, move on to the next, going back to step 2, above, changing roles.
   Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most).
   Perhaps one member of the group can be a timekeeper.

4. For each paper, ask as many of the following questions as you have time for:

   a) Did the author state **whether** *and* **why** they did or did not agree with Knuth's definition in premise 1?

      - **Note:** Knuth's definition is a conjunction of 5 things: 1a & 1b & 1c & 1d & 1e.
        So, in **disagreeing** with premise 1, an author must
        a) explicitly disagree with (at least) one of 1a. . . 1e
        b) **and** say why they disagree with that part (or those parts).

   i) If the author **agreed** and gave reasons for agreeing, do **you** agree with those reasons? Why?

   ii) If the author **disagreed** and gave reasons for disagreeing, do **you** agree with those reasons? Why?

b) Did the author state **whether** *and* **why** they did or did not agree with the claim about the nature of programming languages in premise 2?

(Plus questions 4(a)i and 4(a)ii, above.)

c) Did the author state **whether** *and* **why** they did or did not agree with the claim about the "Turing-equivalence" of programming languages in premise 3?

(Plus questions 4(a)i and 4(a)ii, above.)

d) Did the author state **whether** *and* **why** they did or did not agree with the claim and/or the examples in premise 4?

(Plus questions 4(a)i and 4(a)ii, above.)

e) Did the author state **whether** *and* **why** they believe that conclusion 5 does or does not validly follow from premises 1–4?

Do **you** agree with their evaluation?

f) If the author believes that conclusion 5 follows **soundly** from premises 1–4, then they should state that they believe conclusion 5 for that reason.
Do they?

i) On the other hand, if the author believes that conclusion 5 does **not** follow
(either because one or more of the premises is false or because the argument is invalid),
then did the author state **whether** *and* **why** they did or did not agree with the statement made in the conclusion?

(Plus questions 4(a)i and 4(a)ii, above.)

ii) Note that if the author believes that the argument is *unsound*,
that is **not** a sufficient reason for disbelieving the claim!

(That's because even a **valid** argument can have **both** false premises **and** a true conclusion (or a false one),
and even an **invalid** argument can have a **true** conclusion (or a false one).
The only thing that can't happen is to have a valid argument both with true premises and with a false conclusion.)

g) If the author believes that conclusion 6 follows **soundly** from statement 5 considered as a premise along with some or all of the previous statements in the argument (and possibly along with one or more missing premises!), then they should state that they believe conclusion 6 for that reason. Do they?

- On the other hand, if the author believes that conclusion 6 does **not** follow
(either because one or more of its premises is false or because the argument is invalid),
then did the author state **whether** *and* **why** they did or did not agree with the statement made in the conclusion?

(Plus questions 4(a)i and 4(a)ii, above.)

5. Keep a written record of the questions and replies. This will be useful to the author, for revision.

6. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"):
   Perhaps defend your claims better, or clarify statements that were misunderstood, etc.
   For help, see your instructor.

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, DOUBLE-SPACED, IS DUE ONE WEEK FROM TODAY.
NO LATE PAPERS WILL BE ACCEPTED!

B.4.3 Grading Rubric

```
1.  Prem 1 (Knuth's characterization of "algorithm")

agree? why? 0 = no answer
1 = answer, no reason
2 = answer, unclear reason
3 = answer, clear reason

-------------------------------------------------------------------------

2.  Prem 2 (Prog langs express/implement algorithms)

agree? why? 0,1,2,3, as above

-------------------------------------------------------------------------

3.  Prem 3 (Prog langs are equiv to TM prog lang)

agree? why? 0,1,2,3, as above

-------------------------------------------------------------------------

4.  Prem 4 (Some real comp progs violate Knuth's def)

agree? why? 0,1,2,3, as above

-------------------------------------------------------------------------

5.  Conc 5 (So, such progs don't implement TMs)

1..4/.'.5 valid?
Why?
```

```
0 = no answer
1 = answer, no explanation
2 = answer, weak explanation
3 = answer, good explanation

agree w/ 5?
why? 0,1,2,3, as for 1

-------------------------------------------------------------------------

6.  Conc 6 (So, such progs are not computable)

        5/.'.6 valid?
Why?

        0 = no answer
                        1 = answer, no explanation
                        2 = answer, weak explanation
                        3 = answer, good explanation

        agree w/6?
why?     0,1,2,3, as for 1

-------------------------------------------------------------------------

7.  Citation style:

        used sources w/o citing                          -3
        used sources w/ incomp & incorrect citation      -2
        used sources w/ incomp XOR incorrect citation    -1
        not applicable                                    0

-------------------------------------------------------------------------

8.  Attached draft 1 & list of peer editors
to demo that draft 2 <> draft 1?                 0
    Didn't                                      -1

=========================================================================

The total is 24 points, which, following my grading theory, maps into
letter grades as follows:

letter  CSE484          both    CSE/PHI584
```

A		23–24	
A–		22	
B+		21	
B		19–20	
B–		18	
C+		17	
C	14–16		9–16
C–	11–13		
D+	9–10		
D		5–8	
F		0–4	

B.5 Position Paper #3: Is the Brain a Computer?

B.5.1 Assignment

For this position paper, I would like you to evaluate the following "complex" argument. (It's "complex" because it consists of three "sub"arguments, two of which treat the conclusions of previous ones as premises.)

1. **Turing's Thesis:** A physical object can compute if and only if it can do what a (Universal) Turing Machine (TM) can do.
2. A computer is any physical device that can compute. (Consider this as a (proposed) definition of 'computer'.)
3. The human brain is a physical object that can do what a (Universal) TM can do.
4. Therefore, the human brain is a computer.
5. Microsoft Word is TM-computable. (That is, a Universal TM can execute Microsoft Word.)
6. Therefore, any computer can execute Microsoft Word.
7. Therefore, the human brain can execute Microsoft Word.

As usual, to evaluate this argument, you must determine whether (I) it is **valid** and whether (II) **all the premises are true**.

- If both of those conditions hold, then an argument is said to be **sound**.
- You are logically obligated to believe the conclusions of sound arguments!
  - So, if you ever come across an argument that you think **is** sound, but whose conclusion you **don't** believe

    (by the way, do you really believe line 7 of this argument?),

    then **either** one or more of the premises are false **or** it is invalid (that is, there is some way for the premises to be true yet for the conclusion to be false).

(I) To determine whether the argument is valid, you must suppose (or make believe) "for the sake of the argument" that all the premises **are** true, and then consider whether the conclusions logically follow from them. (Or: Can you imagine some way the world might be so that the premises **are** true but the conclusion is **false**?)

- Note that there are *three* conclusions: lines 4, 6, and 7. So, do you agree that conclusion 4 follows logically from premises 1–3, and/or that conclusion 6 follows logically from premise 5 (maybe with the help of some of the earlier premises), and/or that conclusion 7 follows logically from lines 4 and 6 *considered as premises*?

  If not, are there missing premises that are needed to make the argument(s) valid? If there are, do you agree with them (why/why not)?

(II) It may be too difficult to determine whether each premise *is* true or false. More realistically, you should decide whether you *believe*, or agree with, each premise, **and** you must explain why you do or don't.

Finally, do you agree with the conclusion(s)? If you do, but you think that there's something wrong with the argument, try to present a better one. If you don't agree with the conclusion(s), state why, and try to give an argument for what you *do* believe.

a) Your position paper should be approximately

1–2 typed pages, double-spaced (that is, about 250–500 words), and single-sided.

b) Please bring **5 copies** to lecture on the due date.

c) At the top of the first page, please put the following information **in the following format**:

Position Paper #3, Draft 1	YOUR NAME
DATE DUE	YOUR CLASS

For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website "How to Write" http://www.cse.buffalo.edu/∼rapaport/howtowrite.html
As before, no abstract is needed for this position paper, but you do need to give full citations to any sources that you cite.

DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.5.2 Suggestions and Guidelines for Peer-Group Editing

d) When you get into your small groups, introduce yourselves quickly, and share copies of your papers with each other.

2. Choose one paper to discuss first. (Suggestion: Go in alphabetical order by family name.)

3. After spending *about* 10–15 minutes on the first paper, move on to the next, going back to step 2, above, changing roles. Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most). Perhaps one member of the group can be a timekeeper.

4. For each paper, ask as many of the following questions as you have time for:

   a) Did the author state whether the argument from premises 1–3 to conclusion 4 was valid?

      i) If they thought it was invalid, did they suggest a missing premise that would make it valid (if that's possible)?

   b) Did the author state whether the argument to conclusion 6 was valid?

i) Did they correctly identify its other premises besides premise 5? (Very few real arguments can have only one premise.)

ii) If they thought it was invalid, did they suggest a missing premise that would make it valid (if possible)?

c) Did the author state whether the argument to conclusion 7 was valid?

i) Did they correctly identify its other premises besides premise 6? (Note that sentence 6 is both the conclusion of the previous argument and a premise of this one.)

ii) If they thought it was invalid, did they suggest a missing premise that would make it valid (if possible)?

d) For each premise, ask whether the author stated whether **and** why they did or did not agree with it.

i) If the author **agreed**, then it is preferable (but not necessary) that they give reasons for agreeing. If they did give such reasons, do **you** agree with those reasons? Why?

ii) If the author **disagreed**, then it **is** necessary that they give reasons for disagreeing, so do **you** agree with those reasons? Why?

e) For each argument, if the author thought it was **unsound**, did they state whether they believed its conclusion anyway, on independent grounds (that is, for different reasons)?

- And, if so, do you agree with those reasons?

5. Keep a written record of the questions and replies. This will be useful to the author, for revision.

6. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"): Perhaps defend your claims better, or clarify statements that were misunderstood, etc. For help, see your instructor.

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, DOUBLE-SPACED, IS DUE IN LECTURE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

B.5.3 Grading Rubric

```
Position Paper #3 Grading Rubric                Version: 14 Mar 10
=========================================================================
I. VALIDITY OF ARGUMENTS:

 1.  Argument 1,2,3/.'. 4: valid? + reason

0 = no answer
1 = answer, no reason
```

```
    OR reason that confuses definitions
of 'valid'/'invalid'/'sound'
2 = answer, weak reason
3 = answer, good reason

-------------------------------------------------------------------------

 2.  Argument 1,2,5/.'. 6: valid? + reason 0,1,2,3 as above

-------------------------------------------------------------------------

 3.  Argument 4,6/.'. 7: valid? + reason 0,1,2,3 as above

-------------------------------------------------------------------------

II.  TRUTH VALUES OF STATEMENTS:

 4.  Prem 1: agree? + why? 0,1,2,3 as above

 5.  Prem 2: agree? + why? 0,1,2,3

 6.  Prem 3: agree? + why? 0,1,2,3

 7.  Conc 4: agree? + why? 0,1,2,3

 8.  Prem 5: agree? + why? 0,1,2,3

 9.  Conc 6: agree? + why? 0,1,2,3

10.  Conc 7: agree? + why? 0,1,2,3

=========================================================================

The total is 30 points, which, following my grading theory, maps into
letter grades as follows:

letter CSE484 both CSE/PHI584

  A 29-30
  A-27-28
  B+  26
  B 24-25
  B-22-23
  C+  21
  C 17-20 11-20
  C-14-16
```

```
  D+ 11-13
  D 6-10
  F 0-5

-------------------------------------------------------------------------

As usual, on my grading scheme,

"A" means "understood the material for all practical purposes",
(here, that's 30 pts = 10 questions * 3 pts full credit)

"B" has no direct interpretation,
    but comes about when averaging grades of A and C

"C" means "average",
(here, that's 20 pts = 10 * 2 pts partial credit)

"D" means "did not understand the material,
(here, that's 10 pts = 10 * 1 pt minimum credit)

"F" usually means "did not do the work" (that is, 0 pts),
    but can also come about when averaging Ds and Fs.

Please see my grading website,

http://www.cse.buffalo.edu/~rapaport/howigrade.html

for the theory behind all of this, which I'm happy to discuss via
UBLearns email.
```

B.6 Position Paper #4: What Is a Computer Program?

B.6.1 Assignment

B.6.1.1 The Argument

For this position paper, I would like you to evaluate the following argument:

1. A special-purpose computer (that is, a computer that does just one task) is essentially a hardwired computer program.
2. Such a hardwired computer program is a physical machine.
3. Physical machines can be patented.
4. Therefore, such a hardwired computer program can be patented.
5. The printed text of a computer program is a "literary work" (that is, a piece of writing) in the sense of the copyright law.
6. Literary works can be copyrighted.
7. Therefore, such a computer program can be copyrighted.
8. Nothing can be both patented and copyrighted.
    - **Note:** This premise is a matter of law. You must accept it as true. But you can argue that the law should be changed.
9. There is no computational or other relevant difference between the hardwired computer program and its textual counterpart (except for the different media in which they are implemented, one being hardwired and the other being written on, say, a piece of paper).
10. Therefore, computer programs can be both patented and copyrighted.

To help you evaluate this argument (which we look at in more detail in Ch. 13), here are some extracts from some relevant websites:

a) From the official US Patent Office definition of 'patent':

> a property right granted by the Government of the United States of America to an inventor "to exclude others from making, using, offering for sale, or selling the invention throughout the United States or importing the invention into the United States" for a limited time in exchange for public disclosure of the invention when the patent is granted.
> http://www.uspto.gov/main/glossary/index.html#patent
> accessed 22 November 2012.

b) The Patent Office definition of 'invention':

> any art or process (*way of doing or making things*), machine, manufacture, design, or composition of matter, or any new and useful improvement thereof, or any variety of plant, which is or may be patentable under the patent laws of the United States.
> http://www.uspto.gov/main/glossary/index.html#invention
> accessed 22 November 2012.

c) The official US Copyright Office definition of 'copyright':

> Copyright is a form of protection provided by the laws of the United States. . . to the authors of "original works of authorship," including literary, dramatic, musical, artistic, and certain other intellectual works. This protection is available to both published and unpublished works.
> http://www.copyright.gov/circs/circ01.pdf
> accessed 22 November 2012.

d) From the same website:

> Copyrightable works include the following categories:
>
> (a) literary works
> (b) musical works, including any accompanying words
> (c) dramatic works, including any accompanying music
> (d) pantomimes and choreographic works
> (e) pictorial, graphic, and sculptural works
> (f) motion pictures and other audiovisual works
> (g) sound recordings
> (h) architectural works
>
> These categories should be viewed broadly. For example, computer programs and most "compilations" may be registered as "literary works"; maps and architectural plans may be registered as "pictorial, graphic, and sculptural works."
>
> **WHAT IS NOT PROTECTED BY COPYRIGHT?**
> Several categories of material are generally not eligible for federal copyright protection. These include among others:
>
> - works that have not been fixed in a tangible form of expression (for example, choreographic works that have not been notated or recorded, or improvisational speeches or performances that have not been written or recorded)
> - titles, names, short phrases, and slogans; familiar symbols or designs; mere variations of typographic ornamentation, lettering, or coloring; mere listings of ingredients or contents
> - ideas, procedures, methods, systems, processes, concepts, principles, discoveries, or devices, as distinguished from a description, explanation, or illustration

> - works consisting entirely of information that is common property and containing no original authorship (for example: standard calendars, height and weight charts, tape measures and rulers, and lists or tables taken from public documents or other common sources)
>
> http://www.copyright.gov/circs/circ01.pdf
> accessed 22 November 2012.

To evaluate this argument, you must state whether the argument is **valid** and you must state whether and why you **agree or disagree with each premise**. Remember:

- **Only single statements (like premises and conclusions) can be true or false.** For our purposes, it's enough to say that a statement is true (or false) if you agree (or disagree) with it, because I'm not asking you to convince me that a statement really is true (or false); I'm only asking you to convince me that you have a good reason for agreeing (or disagreeing) with it.
- And **only arguments can be valid or invalid.** An argument is valid if it's impossible for all of its premises to be true while its conclusion is false (and it's invalid otherwise). For our purposes, to determine whether an argument is **valid**, you must suppose (or make believe) "for the sake of the argument" that all the premises *are* true (that is, that you agree with all of them), and then consider whether you would have to logically agree with the conclusion. To determine whether an argument is **invalid**, try to imagine some way the world might be so that the premises are true but the conclusion is false.
- Finally, **only arguments can be sound or unsound.** An argument is sound if it's valid and all of its premises are true (in which case, its conclusion will also have to be true). For our purposes, we'll say that an argument is sound if it's valid and you really do agree with all of its premises (in which case, you really have to agree with the conclusion).
- **You are logically obligated to believe the conclusions of sound arguments!** So, if you ever come across an argument that you think is sound, but whose conclusion you don't believe, then either one (or more) of the premises is false, or it is invalid (that is, there is some way for the premises to be true yet for the conclusion to be false), or both.

This means, of course, that you have to evaluate each premise and each (sub-)argument, and, as usual, I also want you to evaluate the conclusion independently of whether you think that it follows validly or doesn't follow validly from its premises.

a) For this position paper, I want to experiment with something a little bit different.

   Instead of writing a first draft of your paper, I simply want you to fill in the attached "thinksheet", which will be an outline of your argument analysis.

   You will write the paper after peer-editing the thinksheets.

b) Please bring **5 copies** of your filled-out thinksheet to lecture on the due date.

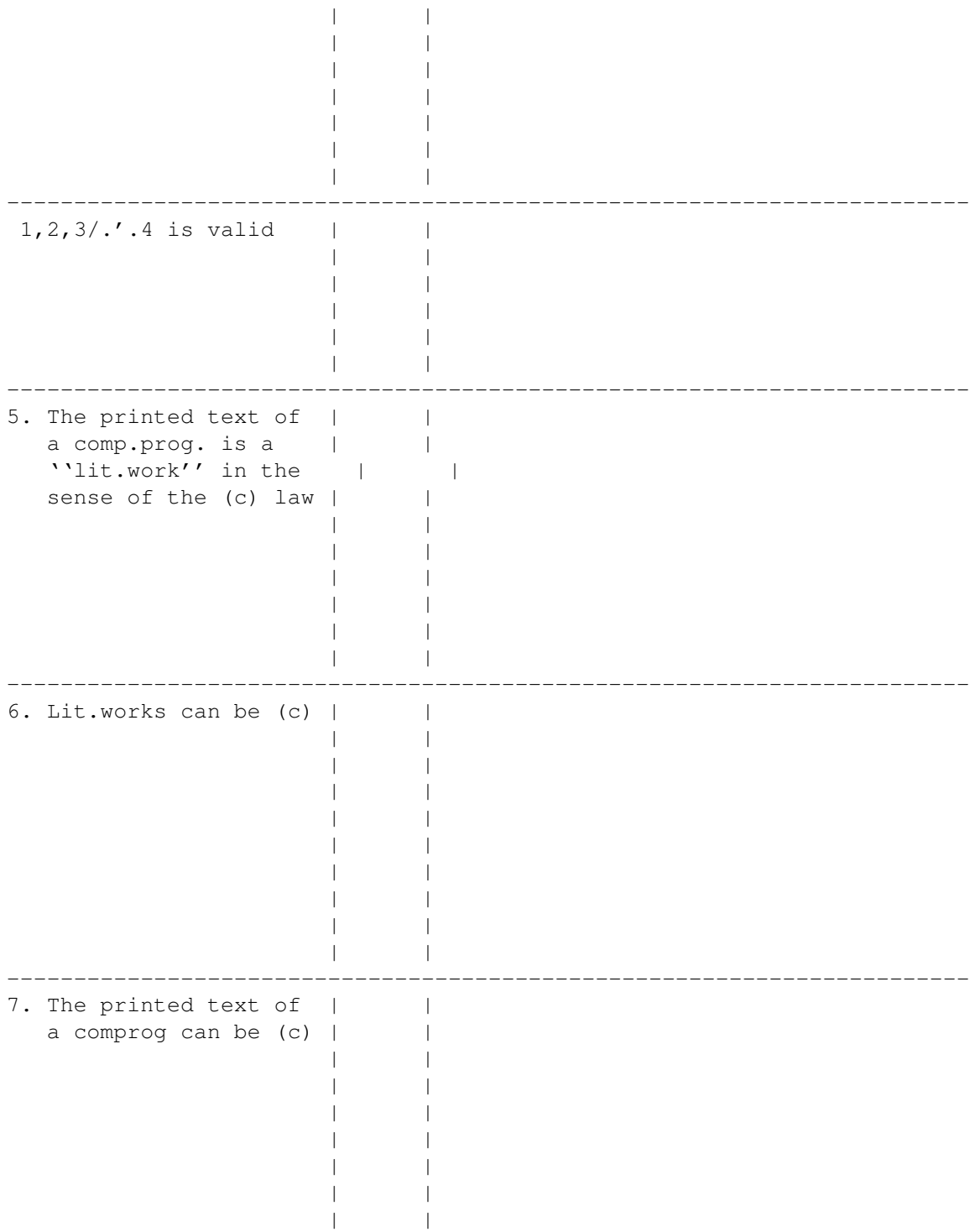
DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.6.1.2 Thinksheet

PhilCS 4/584, Spring 2010 NAME:

Thinksheet for Position Paper #4: What Is a Computer Program?

Statement (abbreviated versions of prems & conc of arg't)	Agree? (T?F?)	Why? (use additional sheets if needed)
1. A special-purpose computer is essentially a hard-wired computer prog.		
2. Such a hardwired comp.prog. is a physical machine		
3. Physical machines can be patented		
4. Such a hardwired comp.prog. can be patented		


1,2,3/.'.4 is valid
5. The printed text of a comp.prog. is a ''lit.work'' in the sense of the (c) law
6. Lit.works can be (c)
7. The printed text of a comprog can be (c)

5,6/.’.7 is valid		
8. Nothing can be both patented & (c)		
9. There’s no comp’n’l or other diff. betw. the hardwired comp. prog. & its textual counterpart...		
10. Comprogs can be both patented & (c)		

```
                          |      |
--------------------------------------------------------------------------
 4,7,9/.'.10 is valid     |      |
                          |      |
                          |      |
                          |      |
                          |      |
                          |      |
                          |      |
                          |      |
                          |      |
                          |      |
--------------------------------------------------------------------------
Additional comments:
```

B.6.2 Suggestions and Guidelines for Peer-Group Editing

A) (a) When you get into your small groups, introduce yourselves quickly, and share copies of your thinksheets with each other.

(b) Choose one thinksheet to discuss first. (Suggestion: Go in alphabetical order by family name.)

(c) After spending *about* 10–15 minutes on the first thinksheet, move on to the next, going back to step Ab, above, changing roles. Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most). Perhaps one member of the group can be a timekeeper.

B) (a) **Make sure each "cell" of the thinksheet is filled in.**

i. **The cells in the "Agree? (T?F?)" column should be filled in with 'agree' or 'disagree' (or 'T' or 'F').**

ii. **The cells in the "Why?" column should contain a reason why the author agrees or disagrees with the statement, or why the author thinks that the argument is valid or invalid.**
(For a reminder about how to explain why you think that an argument is, or isn't, valid,
584/S10/EMAIL/20100323-PP3-Comments.txt
see my earlier email (point #2).) FIX THIS.
These don't have to be complete sentences, but they should be comprehensible.

(b) Keep a written record of the peer-editing suggestions. This will be useful to the author, for revision.

C) The "revision" this time should, of course, be a correctly formatted paper, like the ones you have been writing all semester.

(a) It should be fairly straightforward to turn the thinksheet outline into full sentences and paragraphs (with correct citations if needed).

(b) **I strongly urge you to have someone peer-edit your paper before you produce the final version! Tell that person to make sure that you have:**

i. **Evaluated each statement (premise and conclusion) for ("absolute" or "independent") truth or falsity (see my**
584/S10/EMAIL/20100323-PP3-Comments.txt
email (point #1) about that terminology,
FIX THIS.
and given a reason for your evaluation.

ii. **Evaluated each argument for validity or invalidity**
584/S10/EMAIL/20100323-PP3-Comments.txt
FIX THIS
(that is, evaluated each conclusion for "relative truth", that is, truth relative to the premises), and given a reason for your evaluation.

iii. **Correctly used the "true"/"false"/"valid"/"invalid" terminology.**

(c) Failure to correctly distinguish among **"true (or false) sentences, propositions, statements, premises, or conclusions"** and **"valid (or invalid) arguments"** will result in a lower grade! (After all, you need to demonstrate that you've learned something this semester!)

D) (a) Your position paper should be approximately **1–2 typed pages, double-spaced (that is, about 250–500 words), and single-sided.**

(b) At the top of the first page, please put the following information **in the following format**:

Position Paper #4	YOUR NAME
DATE DUE	YOUR CLASS

(c) Please attach the peer-edited thinksheets to your paper, as usual.

(d) For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website "How to Write".
http://www.cse.buffalo.edu/∼rapaport/howtowrite.html
accessed 22 November 2012.

As before, no abstract is needed for this position paper, but you do need to give full citations to any sources that you cite.

1–2 PAGE (250–500 WORD) PAPER, 1 COPY, TYPED, SINGLE-SIDED, DOUBLE-SPACED, IS DUE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

B.6.3 Grading Rubric

```
Position Paper #4 Grading Rubric              Version:  28 Mar 10
========================================================================
In general:

3 = Statement of position ("agree"/"disagree", or "valid"/"invalid")
    with a clearly stated reason

2 = Statement of position, with an unclear reason

1 = Statement of position, with no reason given

0 = no response

I WILL DEDUCT 3 POINTS FROM THE TOTAL GRADE FOR THE PAPER
FOR INCORRECT USE OF THE TERMINOLOGY!
(If you are not sure of how to use the terminology, please ask me!)

AND MY OFFER TO GIVE YOU BACK ANY SUCH LOST POINTS
(OR LOST POINTS FOR INCORRECT USE OF CITATIONS)
```

STILL HOLDS ON ALL POSITION PAPERS.

a) Evaluation of premise 1: Agree? Why? 0,1,2,3
b) Evaluation of premise 2: Agree? Why? 0,1,2,3
c) Evaluation of premise 3: Agree? Why? 0,1,2,3
d) Evaluation of statement 4: Agree? Why? 0,1,2,3
f) Evaluation of 1,2,3/.'.4: Valid? why? 0,1,2,3

g) Evaluation of premise 5: Agree? Why? 0,1,2,3
h) Evaluation of premise 6: Agree? Why? 0,1,2,3
i) Evaluation of statement 7: Agree? Why? 0,1,2,3
j) Evaluation of 5,6/.'.7: Valid? Why? 0,1,2,3

k) Discussion of premise 8: Agree? Why? 0,1,2,3

Since conclusion 10 conflicts with the law (8),
you have two options:

Either accept the law, and reject 10...
[In that case, you must reject at least one of 1-7, & 9.
Which one, and why? (This is the "Alan Newell" solution.)]

...or else reject the law, and accept 10.
[Of course, you can't do that in real life unless maybe you're a legislator (who can can write new laws)(see footnote (*), below) or a Supreme Court justice (who can declare laws unconstitutional) (see footnote (+), below.) In that case, you should propose a new law. (This is the "David Koepsell" solution.)

l) Evaluation of premise 9: Agree? Why? 0,1,2,3
m) Evaluation of statement 10: Agree? Why? 0,1,2,3
n) Evaluation of 4,7,9/.'.10: Valid? Why? 0,1,2,3

==========

The total is 39 points, which, following my grading theory, maps into letter grades as follows:

letter CSE484 both CSE/PHI584

A 37-39
A-35-36
B+ 33-34
B 31-32

```
  B-29-30
  C+ 27-28
  C 21-26 14-26
  C-18-21
  D+ 14-17
  D 7-13
  F 0-6

-------------------------------------------------------------------------
On my grading scheme,

"A" means "understood the material for all practical purposes",
(here, that's 39 pts = 13 questions * 3 pts full credit)

"B" has no direct interpretation,
    but comes about when averaging grades of A with Cs.

"C" means "average",
(here, that's 26 pts = 13 * 2 pts partial credit)

"D" means "did not understand the material,
(here, that's 13 pts = 13 * 1 pt minimum credit)

"F" usually means "did not do the work" (that is, 0 pts),
    but can also come about when averaging Ds and Fs.

Please see my grading website,

http://www.cse.buffalo.edu/~rapaport/howigrade.html

for the theory behind all of this, which I'm happy to discuss via email.

-------------------------------------------------------------------------
Footnotes:
(*) Compare:  Propose new axioms?
(+) Compare:  Prove that a law is not a theorem of the US Constitution?

(There's a story that the famous logician Kurt Goedel found an
inconsistency in the US Constitution when he was studying for his
American citizenship.  He was going to tell the judge about it, but
Albert Einstein, who accompanied him to the ceremony, quickly changed
the subject :-)  See:  Goldstein, Rebecca (2006), Incompleteness: The
Proof and Paradox of Kurt Godel (Norton).)
```

B.7 Position Paper #5: Can Computers Think?

B.7.1 Assignment

For this position paper, I would like you to evaluate the following hypothetical debate.

Pro: If something behaves in all relevant ways as if it were cognitive, then it *is* cognitive.

Con: What do you mean by "being cognitive"?

Pro: I mean that it:

- can perceive (see, hear, etc.);
- has beliefs, desires, and intentions;
- can remember;
- can use and understand natural language;
- can reason and make rational decisions; etc.

You know, the sort of thing that AI researchers are trying to achieve by computational means.

Con: Do you think they will succeed?

Pro: I'm optimistic: I think that a computer running a suitable AI program (or maybe a suite of programs) will eventually behave in all these ways.

Con: But that means that you think that such an AI-programmed computer *will be* cognitive?

Pro: Yes.

Con: But that's crazy! Computers and computer programs are purely syntactic!

Pro: Now it's my turn to ask for clarification: What do you mean by 'syntactic'?

Con: I mean that all a computer can do is to manipulate the symbols of a formal symbol system.[7]

Pro: So what's the problem?

Con: The problem is that cognition is semantic! That is, it involves the semantic interpretation of those symbols.

Pro: Well, I'm not so sure about that. But suppose you're right. What then?

Con: Well, syntax does not suffice for semantics. So, no computer executing a purely syntactic computer program can exhibit semantic cognition, even if it behaves in all relevant ways as if it were cognitive.

[7] http://www.cse.buffalo.edu/∼rapaport/formalsystems—FIX THIS

- It may help if you rewrite Pro's and Con's arguments in terms of premises and conclusions, and then evaluate those arguments. That is, "extract" each argument from the debate and put them in the following forms:

1. Pro's premise 1	1. Con's premise 1
2. Pro's premise 2	2. Con's premise 2
3. (etc.)	(etc.)
4. Therefore, Pro's conclusion	4. Therefore, Con's conclusion

  Then analyze each argument.

- Keep in mind that premises and conclusions are declarative propositions (they can be deemed to be true or false) but that some lines uttered by Pro and Con are *not* declarative propositions (and thus can't be premises or conclusions). For example, Con's first statement is a question—it is not a premise or conclusion of anyone's argument—and Pro's second statment needs to be reformulated as something like "Something is cognitive means that it…".

1. For your peer-editing session next week, I will give you a choice: You may either:

   (a) create a "thinksheet" like the one for Position Paper #4 (§B.6.1.2)
   - with one column listing the premises, conclusions, and arguments;
   - one column of "cells" to indicate your agreement or disagreement with them;
   - and one column of "cells" to indicate your reasons for your agreement or disagreement

   (b) or write a 1–2 page, double-spaced (that is, about 250–500 word), single-sided, first draft.

   (Of course, you might want to do option 1a for your own use before doing option 1b! They are not mutually inconsistent.

   If your document is more than 1 page long, please **staple** the pages together **and** make sure that your name is on all pages!

2. Please bring **5 copies** to class on the due date.

3. **At the top of the first page, please put ALL AND ONLY the following information** *in the following format***:**

Position Paper #5	**YOUR NAME**
DATE DUE	**YOUR CLASS**

4. Failure to correctly distinguish among **"true (or false) sentences, propositions, statements, premises, or conclusions"** and **"valid (or invalid) arguments"** will also result in a lower grade!

5. For general assistance with writing (including my preferred method of paper preparation and format, as well as advice on grammar), see my website "How to Write".
   http://www.cse.buffalo.edu/∼rapaport/howtowrite.html

   And don't forget to give full citations to any sources that you cite.

DUE AT THE BEGINNING OF LECTURE, ONE WEEK FROM TODAY

B.7.2 Suggestions and Guidelines for Peer-Group Editing

1. When you get into your small groups, introduce yourselves quickly, and share copies of your papers with each other.

2. Choose one paper to discuss first. (Suggestion: Go in alphabetical order by family name.)

3. After spending *about* 10–15 minutes on the first paper, move on to the next, going back to step 2 above, changing roles. Spend *no more than* 15 minutes per paper (because you've only got about 45 minutes at most). Perhaps one member of the group can be a timekeeper.

4. **Suggestion:** There are really 2 arguments in this dialogue: Pro's argument and Con's argument.

   So, the first task is to present each argument. Once you have identified the premises (including any hidden premises) and conclusion of each argument, you can then analyze it for validity of the argument and truth of the premises.

5. For each paper in your peer-editing group, ask as many of the following questions as you have time for:

   (a) Did the author present both Pro's and Con's arguments?

   (b) For each argument, did the author state whether and why s/he believes the argument to be valid?

   - It's possible to formulate both arguments so that they *are* valid!
   - If you do that, then ascertaining the truth value of the premises becomes your central task.

   (c) For each argument, did the author state whether and why s/he agrees with the premises?

   (d) For each argument, if the author believed either that the argument was invalid (even with missing premises added—that is, that there was *no* way to make the argument valid) or that one or more of the premises was false, then did the author state whether and why s/he agrees with the conclusion?

   - **Reminder:**

i. If you think an argument is sound, then you are logically obligated to believe its conclusion (and you don't have to give any other justification for the conclusion).

ii. If you don't believe the conclusion of an argument, then it is either invalid or else has at least one false premise; you must identify which, and explain why.

iii. If you think an argument is unsound (either because it is invalid or has at least one false premise), then you might *still* believe the conclusion *for other reasons*; in that case, you must give those other reasons.

6. **Remember!**: Your revised paper must have the appropriate heading at the top of the first page, must use the terms 'true', 'false', 'valid', and 'invalid' appropriately, and must have your peer-edited first drafts attached!

7. Keep a written record of the questions and replies. This will be useful to the author, for revision.

8. At home, over the next week, please *revise* your paper to take into consideration the comments made by your fellow students (that is, your "peers"): Perhaps defend your claims better, or clarify statements that were misunderstood, etc. For help, see your instructor.

1–2 PAGE (250–500 WORD) REVISION, 1 COPY, TYPED, SINGLE-SIDED, DOUBLE-SPACED, IS DUE ONE WEEK FROM TODAY. NO LATE PAPERS WILL BE ACCEPTED!

B.7.3 Grading Rubric

```
Here's a draft of the grading rubric for Position Paper #5.

Because you need to spell out Pro's and Con's arguments in
premise-conclusion form, and because this may use up space,
it will not count against the word- and page-limits.

But if your paper is > 1 page, please STAPLE the pages together
(one staple, in upper left corner)
AND
please put your NAME on ALL pages.

=========================================================================
Position Paper #5 Grading Rubric             Version: 20 Apr 2010
=========================================================================

a)  Incorrect use of "true", "false", "valid", "invalid", "sound",
    "unsound", "argument", "premise", "conclusion", etc.:
```

```
-1 pt PER ERROR!

==========================================================================

b)  PRO'S ARGUMENT

b1)  List of premises & conclusion for Pro's argument:

3 = * clearly stated argument,
    * premises & conclusion clearly identified,
    * prems & conc clearly derived from dialogue

2 = neither clearly 3 nor 1,
    including:  not correctly identifying
some prem or conc.

1 = arg not clearly presented
    or not clearly derived from dialogue

0 = missing

--------------------------------------------------------------------------

b2)  Evaluation of validity of Pro's argument:

3 = valid XOR invalid
    + clear explanation why
      (including addition of any missing premises)
2 = valid XOR invalid, unclear explanation
1 = valid XOR invalid, no explanation
0 = no evaluation of validity

--------------------------------------------------------------------------

b3)  Evaluation of truth-value of Pro's premises:

Note:  Because each of you might have slightly different premises,
       I can't assign points to each one in any equally fair way,
       so I will grade you on your overall evaluation of the truth-
       values of the premises that you have explicitly identified.

6 = for EACH premise:
     truth-value clearly stated
     & good reasons given

4 = for SOME (but not all) premises:
```

```
     truth-value not stated
     OR no or weak reason given

2 = for MOST (UPDATED: or all) premises:
     truth-value not stated
     OR no or weak reasons given

0 = no evaluation of truth-values of premises

[Note: Because of my "quantum" scheme of grading,
       it is not possible to get 1,3, or 5 points!]

-------------------------------------------------------------------------

b4)  Evaluation of truth-value of Pro's conclusion:

3 = if arg is sound,
then that is your reason for believing conc,
-- say so!
     else (if arg is not sound, then)
     say whether you believe conc
     & give clear reason why

2 = neither clearly 1 nor 3

1 = you think arg is not sound (which is fine),
    but you give no clear statement of truth-value of conc
    & no or weak reason given

0 = no evaluation of conclusion

=========================================================================

c)  CON'S ARGUMENT (to be graded similarly, namely:)

c1)  List of premises & conclusion for Con's argument:

0,1,2,3 pts as above

c2)  Evaluation of validity of Con's argument:

0,1,2,3 pts as above

c3)  Evaluation of truth-value of Con's premises:

0,2,4,6 pts as above
```

```
c4)  Evaluation of truth-value of Con's conclusion:

0,1,2,3 pts as above

=============================================================================

The total is 30 points, which, following my grading theory, maps into
letter grades as follows:

letter CSE484 both CSE/PHI584

  A 29-30
  A-27-28
  B+  26
  B 24-25
  B-22-23
  C+  21
  C 17-20 11-20
  C-14-16
  D+ 11-13
  D 6-10
  F 0-5

-----------------------------------------------------------------------------

On my "quantum-triage" grading scheme,

"A" means "understood the material for all practical purposes",
(here, that's 30 pts = 6 questions * 3 pts full credit
       + 2 questions * 6 pts full credit)

"B" has no direct interpretation,
    but comes about when averaging grades of A and C

"C" means "average",
(here, that's 20 pts = 6 * 2 pts partial credit
     + 2 * 4 pts partial credit)

"D" means "did not understand the material,
(here, that's 10 pts = 6 * 1 pt minimum credit
     + 2 * 2 pts minimum credit)

"F" usually means "did not do the work" (i.e., 0 pts),
    but can also come about when averaging Ds and Fs.
```

Please see my grading website,

http://www.cse.buffalo.edu/~rapaport/howigrade.html

for the theory behind all of this, which I'm happy to discuss.

B.8 Optional Position Paper: A Competition

Some of you have told us that you would like to come up with your own arguments instead of merely analyzing ones that we give you.

Here's your opportunity!

No later than two weeks from today, try your hand at creating an argument relevant to one of the topics of this course. It could be on a topic that we've already discussed, on a topic that we're going to discuss, or on some other topic in the philosophy of computer science (for ideas, take a look at the Further Readings at the ends of each chapter).

Your argument should have at least two premises and at least one conclusion. Try to make it a *valid* argument!

The "winner" (if there is one—we reserve the right to decide not to choose one) will have the honor of her or his argument being used as the argument to be analyzed for the next Position Paper. (To make it interesting and fair, for his or her position-paper assignment, the winner may be asked to *refute* the argument!)

You may submit the argument on paper (in lecture or by email). We also reserve the right to slightly modify the argument, if we feel that would make it more interesting.

Appendix C

Term Paper Topics

Version of 9 December 2017; DRAFT © 2004–2017 by William J. Rapaport

To the instructor: Discuss this at approximately 3 weeks into a 15-week term.
Here is a list of some possible term-paper topics:

1. Further discussion of any topic covered in class, e.g.:
   (a) A critical examination of (someone else's) **published** answer to one of the questions listed on the syllabus.
   (b) **Your** answer to one of the questions listed on the syllabus, including a defense of your answer.
2. A critical examination of any of the required or recommended (or any other approved and relevant) readings.
3. A critical study of any monograph (i.e., single-topic book) or anthology (including special issues of journals) on the philosophy of computer science.
4. A critical, but general, survey article on the philosophy of computer science that would be appropriate for an encyclopedia of philosophy or an encyclopedia of computer science.
5. A presentation and well-argued defense of *your* "philosophy of computer science", that is, *your* answers to all (or most) of our questions, together with supporting reasons.
6. Other ideas of your own, approved by me in advance, including, but not limited to, such topics that we have mentioned but have not investigated as:
   - Is CS an empirical or a mathematical science?
   - Is CS an artificial or a natural science?
   - How does (the philosophy of CS) fit in with Alfred North Whitehead's "process philosophy"?

For general assistance with writing (including my required method of paper preparation and format, as well as advice on grammar), see my website "How to Write" http://www.cse.buffalo.edu/

For specific assistance on writing a philosophy paper, see:

- Mendelovici, Angela (2011), "A Sample Philosophy Paper", online at: http://tinyurl.com/SamplePhilPaper (accessed 13 February 2013)
- Wolff, Robert Paul (1975), "A Simple Foolproof Method for Writing Philosophy Papers", in his *About Philosophy* (Prentice-Hall); online at: http://www.amyscott.com/Philosophy%20Paper.pdf (accessed 13 February 2013)
- Or do an Internet search for "how to write a philosophy paper".

The paper should be a maximum of **10 double-spaced, single-sided pages** (that is, about 2500 words) (not counting the bibliography).

Deadlines:

1. **Two weeks from today: Proposal and reading list due.**
    - Your proposal and reading list must be approved by me before you begin your research and writing.
    - Because the term paper is optional, you do not need to commit yourself to it even if you turn in a proposal.
2. **Final paper will be due on the last day of the course.**

Appendix D

Final Exam

Version of 9 December 2017; DRAFT © 2004–2017 by William J. Rapaport

D.1 To the Instructor

At least one of the goals of philosophy education ought to be the fostering of the students' development of analytical and critical thinking skills. In order to explain the nature of the sample final exam to be presented in the next section, I want to call attention to a theory of cognitive development of college students, to discuss its implications for critical thinking, and to show how it can apply to the development of writing exercises such as this final exam.[1]

William G. Perry's scheme of cognitive development (Perry, 1970), (Perry, 1981) is a descriptive theory of stages that represent students' changing attitudes towards knowledge and values as they progress through their education. There are nine stages, which fall into four groups.[2]

I. Dualism **Position 1.** *Basic Duality:* Students at this stage believe that there are correct answers to all questions, that the instructor is the authority figure who has access to "golden tablets in the sky" containing all correct answers, and that their (the students') job is to *learn the correct answers* so that they can be repeated back to the instructor when asked. A Basic Duality student who offers a wrong answer to a question hears the authority figure say, "*You* are wrong".

Position 2. *Dualism:* Students move to this stage when faced with alternative opinions or disagreements among different authority figures (different instructors), such as when one literature teacher says that *Huckleberry Finn* is

[1] This section is adapted from (Rapaport, 1984b).

[2] My descriptions are culled from (Perry, 1981), (Cornfeld and Knefelkamp, 1979), and (Goldberger, 1979). These are three essential readings for anyone concerned with implications and applications of Perry theory in the classroom.

the best American novel, but another says that it is the worst. Dualistic students infer that one of those literature teachers' view of the golden tablets is obscured. Consequently, they see the purpose of education as *learning to find* the correct answers.

Dualistic students prefer structured classes, which they see as providing the correct answers, and subjects such as math, which they see as having clear answers (all math teachers agree that the golden tablets say that $2+2=4$). Conflict between instructor and text, or between two instructors, is seen threateningly as conflicts among authority figures.

II. Multiplicity Position 3. *Early Multiplicity:* Here, the student has moved from the narrow Dualism of seeing all questions as having correct or else incorrect answers to a wider dualism of classifying questions into two kinds: those where instructors know the correct answers and those where they don't know the correct answers *yet*. Concerning the latter, the instructor's role is seen by Early Multiplistic students as providing *methods for finding* the correct answers, rather than as giving the correct answers directly.

Position 4. *Late Multiplicity:* As students move along, chronologically or cognitively, they begin to see the second kind of question as being the more common one. Because, it is felt, *most* questions are such that instructors don't have the correct answers for them, "everyone has a right to his own opinion; no one is wrong!" (Perry, 1981, 79). The instructor's task, therefore, is seen as either teaching *how to think* or, worse, being irrelevant (after all, everyone has a right to their own opinion, including instructors—but it's *just their* opinion).

III. Contextual Relativism Position 5. Here, students have begun to see that instructors aren't always asking for correct answers but, rather, for *supported* answers. The second category of questions has become the only category (except, "of course", in mathematics and science!), but, although there can be many answers for each question, some are better (more adequate, more appropriate, etc.) than others. Answers are now seen as being better or worse *relative to* their supporting *context* (hence the name of this position).

IV. Commitment within Relativism Positions 6–9. These stages characterize students as they begin to see the need for making their own decisions (making commitments, affirming values), as they balance their differing commitments, and as they realize the never-ending nature of this process.

Finally, there is evidence that a student as Position x will not understand—wil literally not be able to make any sense out of—instruction aimed at Position $x+2$ (or beyond). Conversely, students at higher levels are bored by instruction aimed at lower levels.[3]

[3]Perry's theory is far, far richer than I have portrayed it here, and the interested reader is urged to follow up the suggested readings. A useful survey of *criticisms*, together with a discussion of the relevance of the theory to mathematics education and to the history of matheamtics, is (Copes, 1980). The relevance of the theory to philosophy is discussed in (Rapaport, 1982).

Here is a useful anecdote (adapted from Perry) for illustrating the scheme: Suppose that a teacher presents three theories for explaining the purpose of science (for example, is science merely descriptive, merely predictive, or explanatory?), or three different algorithms for solving some computational problem. The Dualistic student will wonder which is the correct one (and why the teacher bothered to talk about the incorrect ones); The Multiplistic student will think, "Only 3? Heck, I can come up with 6!"; the Contextual Relativist will wonder what advantages and disadvantages each theory has; and the Commitment-oriented student will be wondering about how to decide which is most appropriate or useful in a given situation.

Data from several studies indicate that most entering college freshmen are at Positions 2 or 3 (Perry, 1970), (Perry, 1981), (Copes, 1980). *Courses designed to teach critical thinking skills to students through the first two years of college are thus dealing with Dualists or (early) Multiplists, and this can result in several problems that the instructor should be aware of in order to foster the students' development along the Perry scheme.*

First, Dualists want to be told the correct answers. But critical-thinking courses are largely involved with criticism and argument analysis. Accordingly, the entire activity may appear to them as incomprehensible at worst and pointless at best, or may simply result in the students learning the "sport" of "dumping" on "bad" or "incorrect" arguments. Hence, such courses, including the present one, must be more than mere criticism courses; they must make a serious attempt to teach ways of *constructing* arguments, *solving* problems, or *making decisions*. In this way, they can offer an appropriate "challenge" to Dualistic students, especially if couched in a context of adequate "support".[4]

Second, "The highly logical argument that, 'since everybody has a right to their own opinion, there is no basis for rational choice' is very typical of Multiplistic students" (Goldberger, 1979, 7). But a goal of critical-thinking courses should be precisely to provide bases for rational choice: logical validity, inductive strength, etc. Accordingly, Multiplistic students either will not comprehend this goal or will view it as pointless. Again, such a course can be appropriately challenging to the students, but the instructor must be aware of how the students are likely to perceive it—to "hear" students' negative comments not as marks of pseudo-sophistication or worse, but as marks of viewing the world Multiplistically.

Finally, consider the concept of logical validity. Larry Copes (personal conversation) points out that it is a "relativistic" concept: A "valid" conclusion is one that is true *relative to* the truth of the premises. Dualistic students searching for absolutes and Multiplistic students feeling that "anything goes" may not accept, like, or understand validity. This may explain why so many students believ that arguments with true conclusions are valid or that valid arguments require true premises—even after having dutifully memorized the definition of 'validity'!

How can an instructor simultaneously *challenge* students in order to help them move to a higher-numbered stage, yet *not threaten* them, especially when a given class might have students at widely varying stages? One suggestion, based on work by Lee

[4]For details and specific advice, see (Sanford, 1967, Ch. 6, esp. pp. 51–52), (Cornfeld and Knefelkamp, 1979).

Knefelkamp, is to create assignments that can appeal to students at several levels.

The suggested final exam in the next section is one such assignment. Students are offered five questions and asked to answer any three of them. Question 1 is similar to, but not exactly the same as, the argument in Position Paper #1, so it is a *little bit* challenging to the Dualistic student, but is supportive in that they have already practiced giving a response to it. Question 2 is a brand-new argument for analysis, but one that students could have predicted that it would have appeared on a final exam, because it covers a topic that was explicitly discussed in the course yet was not the subject of a position paper; consequently, it is challenging, because it is new, but supportive, because it covers material that should be familiar—it should appeal to both Dualistic and Multiplistic students. Questions 3 and 4 appear to be open-ended questions that should appeal to Multiplistic students, though they can be understood as questions that might appeal to Dualistic students, too; after all, they are topics that were covered in the course, and the students have been given tools for evaluating answers to such questions—the challenge here is to construct arguments for answers to the questions. Here, the student's choice of which question (3 or 4) to answer (if any) is a function of their personal interests or familiarity with the issues. Question 5 is the most challenging, because the student must come up with a question and then answer it; it should appeal to Multiplistic as well as Contextual Relativistic students.

If left to their own choices, students will choose the least challenging question commensurate with their current stage. Thus, students need not be threatened by a question that they perceive as being to difficult or even unintelligible. But each question is just a bit more challenging than the previous one, and, because the students must answer three questions, they are offered a choice that includes at least one fully supportive question and at least one more-challenging question. (If such an exam is offered as a mid-term exam, then the final exam could begin with a least-challenging question that is *more* challenging than the least-challenging one on the mid-term.) In this way, students are encouraged to strive for a bit more, thus, hopefully, beginning the move to the next stage of cognitive development.

It is imperative for those of us who teach such courses to learn how to challenge our students appropriately in order to foster their intellectual "growth"; we must "hear" how our students inevitably make their *own* meaning out of what we say to them; and we must be ready to support them in the ego-threatening process of development.

D.2 A Final Exam

Do any 3 of the following. Write about 250–500 words for each answer.

This is a closed-book, closed-notes, closed-neighbor, open-mind exam.

No books, notebooks, food, beverages, or electronic devices of any kind are permitted in the exam room.

1. Evaluate the following argument (note that it is similar to, but *not* exactly the same as, the argument in Position Paper #1):

   > Natural science is the systematic observation, description, experimental investigation, and theoretical explanation of natural phenomena.

> Computer science is the study of computers and computing. Therefore, computer science is not a natural science.

2. Evaluate the following argument:

> Suppose that computers running certain computer programs can make rational decisions (at least in the sense of outputting values of functions that serve as a basis for decision making). That is, suppose that they can determine the validity of arguments and ascertain the probable truth-values of the premises of the arguments, and that they can consider the relative advantages and disadvantages of different courses of action, in order to determine the best possible choices. (For example, there are computers and computer programs that can diagnose certain diseases and (presumably) recommend appropriate medical treatments; there are computers and computer programs that can prove and verify proofs of mathematical theorems; and there are computers and computer programs that can play winning chess.) Suppose for the sake of argument that some of these computers and computer programs can make decisions (and recommendations) on certain important matters concerning human welfare. Suppose further that they can regularly make *better* recommendations than human experts on these matters. Therefore, these computers *should* make decisions on these important matters concerning human welfare.

3. What is computer science?

4. Can computers think?

5. Choose either 5a or 5b:

   (a) In your opinion, what is the most fundamental or important question in the philosophy of computer science?

   (b) What is a question that interests you in the philosophy of computer science that we did *not* discuss this semester?

   Pose the question, explain why you think it is important or interesting, and present your answer to it.

D.3 NOTES FOR NEXT DRAFT

- On the Perry Scheme:
  See http://www.askphilosophers.org/question/5563

Bibliography

Aaronson, S. (2006). Phys771 lecture 4: Minds and machines. http://www.scottaaronson.com/democritus/lec4.html.

Aaronson, S. (2008, March). The limits of quantum computers. *Scientific American*, 62–69. http://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf.

Aaronson, S. (2011, 9 December). Quantum computing promises new insights, not just supermachines. *New York Times*, D5. http://www.nytimes.com/2011/12/06/science/scott-aaronson-quantum-computing-promises-new-insights.html.

Aaronson, S. (2012). The toaster-enhanced turing machine. *Shtetl-Optimized*. http://www.scottaaronson.com/blog/?p=1121.

Aaronson, S. (2013a). *Quantum Computing Since Democritus*. New York: Cambridge University Press.

Aaronson, S. (2013b). Why philosophers should care about computational complexity. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 261–327. Cambridge, MA: MIT Press. http://www.scottaaronson.com/papers/philos.pdf.

Aaronson, S. (2014, July-August). Quantum randomness. *American Scientist 102*(4), 266–271. http://www.americanscientist.org/issues/pub/quantum-randomness.

Aaronson, S. (2018, 13 September). Three questions about quantum computing. https://www.scottaaronson.com/talks/3questions.ppt. PowerPoint talk presented upon receiving the 2018 Tomassoni-Chisesi Prize in Physics at Universitá "La Sapienza", Rome.

Abelson, H., G. J. Sussman, and J. Sussman (1996). *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press. http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-7.html#%_chap_Temp_4.

Abrahams, P. (1987, June). What is computer science? *Communications of the ACM 30*(6), 472–472.

Abramson, D. (2011). Philosophy of mind is (in part) philosophy of computer science. *Minds and Machines 21*, 203–219.

Abramson, D. (2014, 31 December). The philosophical legacy of Alan Turing". http://www.thecritique.com/articles/the-philosophical-legacy-of-alan-turing/.

Acocella, J. (2009, 3 August). Betrayal: Should we hate judas iscariot? *The New Yorker*, 68–73.

Adamson, P. (2019, 27 June). What was philosophy? *New York Review of Books 66*(11), 55–57.

Adleman, L. M. (1998, August). Computing with DNA. *Scientific American*, 54–61. http://www.usc.edu/dept/molecular-science/papers/fp-sciam98.pdf.

Agre, P. (1992). Control structures. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence, 2nd Edition*, pp. Vol. 1, pp. 293–301. New York: John Wiley & Sons.

Aho, A. V. (2011, January). What is computation? Computation and computational thinking. *Ubiquity 2011*(January). Article 1, http://ubiquity.acm.org/article.cfm?id=1922682.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1983). *Data Structures and Algorithms*. Reading, MA: Addison-Wesley.

Aizawa, K. (2010, September). Computation in cognitive science: It is not all about Turing-equivalent computation. *Studies in History and Philosophy of Science 41*(3), 227–236. Reply to Piccinini (2004a).

Akman, V. (2000, July). Philosophical foundations of artificial intelligence (special issue). *Journal of Experimental & Theoretical Artificial Intelligence 12*(3). Table of contents at http://www.tandfonline.com/toc/teta20/12/3; editor's introduction at http://www.cs.bilkent.edu.tr/~akman/jour-papers/jetai/jetai2000.pdf.

Akman, V. and P. Blackburn (2000, October). Alan Turing and Artificial Intelligence (special issue). *Journal of Logic, Language and Information 9*(4). Table of contents at https://link.springer.com/journal/10849/9/4.

Alama, J. and J. Korbmacher (2018). The lambda calculus. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2018 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2018/entries/lambda-calculus/.

Albert, D. Z. (2018, 19 April). Quantum's leaping lizards. *New York Review of Books 65*(7), 55–57.

Alden, J. R. (1999, January). Review of Standage (1998). *Smithsonian 29*(10), 126–127. http://www.smithsonianmag.com/history-archaeology/bookreview_jan99_a.html.

Allen, C. (2017). On (not) defining cognition. *Synthese 194*, 4233–4249.

Allen, L. G. (2001, September). Teaching mathematical induction: An alternative approach. *Mathematics Teacher 94*(6), 500–504.

Allen, S. (1989). *Meeting of Minds*. Prometheus Books.

Allo, P. (2010, April). Putting information first: Luciano Floridi and the philosophy of information. *Metaphilosophy 41*(3), 248–254.

American Mathematical Society (2006, November). Special issue on Alan Turing. *Notices of the AMS 53*(10). http://www.ams.org/notices/200610/.

Anderson, A. R. and N. D. Belnap, Jr. (Eds.) (1975). *Entailment: The Logic of Relevance and Necessity*, Volume I. Princeton, NJ: Princeton University Press.

Anderson, A. R., N. D. Belnap, Jr., and J. M. Dunn (Eds.) (1992). *Entailment: The Logic of Relevance and Necessity*, Volume II. Princeton, NJ: Princeton University Press.

Anderson, B. L. (2015). Can computational goals inform theories of vision? *Topics in Cognitive Science*. DOI: 10.111/tops.12136.

Anderson, D. L. (2006). The nature of computers. *The Mind Project*. http://www.mind.ilstu.edu/curriculum/modOverview.php?modGUI=196.

Anderson, L. (2014, 3 December). How accurate is *the imitation game*? *Slate*. http://www.slate.com/blogs/browbeat/2014/12/03/the_imitation_game_fact_vs_fiction_how_true_the_new_movie_is_to_alan_turing.html.

Anderson, M. and S. L. Anderson (Eds.) (2006). *Special Issue on Machine Ethics*. IEEE Intelligent Systems. Vol. 21, No. 4 (July/August).

Anderson, M. and S. L. Anderson (2007, Winter). Machine ethics: Creating an ethical intelligent agent. *AI Magazine 28*(4), 15–26.

Anderson, M. and S. L. Anderson (2010, October). Robot be good. *Scientific American 303*(4), 72–77.

Angere, S. (2017, January). The square circle. *Metaphilosophy 48*(1–2), 79–95. Preprint at http://portal.research.lu.se/ws/files/6013414/4393821.pdf.

Angier, N. (2010, 2 February). Abtract thoughts? The body takes them literally. *New York Times/Science Times*, D2. http://www.nytimes.com/2010/02/02/science/02angier.html.

Anonymous (1853a, January). Preface. *Quarterly Journal of Microscopical Science 1*(1), 1–2.

Anonymous (1853b, January). Review of j.h. wythes, *the microscopist; or a complete manual on the use of the microscope*. *Quarterly Journal of Microscopical Science 1*(1), 51–53.

Anthes, G. (1 May 2006). Computer science looks for a remake. *Computerworld*, http://www.computerworld.com/s/article/110959/Computer_Science_Looks_for_a_Remake.

Antoy, S. and M. Hanus (2010, April). Functional logic programming. *Communications of the ACM 53*(4), 74–85.

Apostel, L. (1961). Towards the formal study of models in the non-formal sciences. In H. Freudenthal (Ed.), *The Concept and the Role of the Model in Mathematics and Natural and Social Sciences: Proceedings of the Colloquium Sponsored by the Division of Philosophy of Sciences of the International Union of History and Philosophy of Sciences Organized at Utrecht, January 1960*, pp. 1–37. Dordrecht, Holland: D. Reidel.

Appiah, K. A. (2007, 9 December). The new new philosophy. *New York Times Magazine*, 34–36.

Appiah, K. A. (2008, November). Experimental philosophy. *Proceedings and Addresses of the American Philosophical Association 82*(2), 7–22. https://member.apaonline.org/V82_2_experimentalphilosophy.aspx.

Arden, B. W. (Ed.) (1980). *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)*. Cambridge, MA: MIT Press.

Ardis, M., V. Basili, S. Gerhart, D. Good, D. Gries, R. Kemmerer, N. Leveson, D. Musser, P. Neumann, and F. von Henke (1989, March). Editorial process verification. *Communications of the ACM 32*(3), 287–290. "ACM Forum" letter to the editor, with replies by James H. Fetzer and Peter J. Denning.

Aref, H. (2004, 5 March). Recipe for an affordable supercomputer: Take 1,100 apples .... *Chronicle of Higher Education*, B14.

Argamon, S., M. Koppel, J. Fine, and A. R. Shimoni (2003). Gender, genre, and writing style in formal written texts. *Text & Talk 23*(3), 321–346. http://writerunboxed.com/wp-content/uploads/2007/10/male-female-text-final.pdf.

Arkoudas, K. (2008, December). Computation, hypercomputation, and physical science. *Journal of Applied Logic 6*(4), 461–475. http://people.csail.mit.edu/kostas/papers/jal.pdf.

Arner, T. and J. Slein (1984, Spring). Philosophy and data processing: An alternative to the teaching profession. *International Journal of Applied Philosophy 2*(1), 75–84. DOI: 10.5840/ijap1984215.

Arnow, D. (1994, March). Teaching programming to liberal arts students: Using loop invariants. *ACM SIGCSE Bulletin 26*(1), 141–144. http://www.panix.com/~arnow/brooklyn_college/ED/coreloopinvar.html.

Asimov, I. (1950). The evitable conflict. *Astounding Science Fiction*. Reprinted in Isaac Asimov, *I, Robot* (Garden City, NY: Doubleday, 1950), Ch. 9, pp. 195–218.

Asimov, I. (1957). The feeling of power. In C. Fadiman (Ed.), *The Mathematical Magpie*, pp. 3–14. New York: Simon and Schuster, 1962. http://www.themathlab.com/writings/short%20stories/feeling.htm.

Asimov, I. (1976). The bicentennial man. In I. Asimov (Ed.), *The Bicentennial Man and Other Stories*, pp. 135–173. Garden City, NY: Doubleday. http://www.ebooktrove.com/Asimov,%20Isaac/Asimov,%20Isaac%20-%20The%20Bicentennial%20Man.pdf.

Austin, A. K. (1983). An elementary approach to *np*-completeness. *American Mathematical Monthly 90*, 398–399.

Avigad, J. (2014, September). Review of Cooper and van Leeuwen (2013). *Notices of the American Mathematical Society 61*(8), 886–890. http://www.ams.org/notices/201408/201408-full-issue.pdf.

Avigad, J. and J. Harrison (2014, April). Formally verified mathematics. *Communications of the ACM 57*(4), 66–75.

Ayer, A. (1956). *The Problem of Knowledge*. Baltimore: Penguin.

Baars, B. J. (1997). Contrastive phenomenology: A thoroughly empirical approach to consciousness. In N. Block, O. Flanagan, and G. Güzeldere (Eds.), *The Nature of Consciousness: Philosophical Debates*, pp. 187–201. Cambridge, MA: MIT Press.

Bacon, D. (2010, December). What is computation? Computation and fundamental physics. *Ubiquity 2010*(December). Article 4, http://ubiquity.acm.org/article.cfm?id=1920826.

Bacon, D. and W. van Dam (2010, February). Recent progress in quantum algorithms. *Communications of the ACM 53*(2), 84–93.

Bader, R. M. (2013, October). Towards a hyperintensional theory of intrinsicality. *Journal of Philosophy 110*(10), 525–563. http://users.ox.ac.uk/~sfop0426/Intrinsicality%20(R.%20Bader).pdf.

Baier, A. (1986, January). Trust and antitrust. *Ethics 96*(2), 231–260.

Bajcsy, R. (2010, December). What is computation? computation and information. *Ubiquity 2010*(December). Article 2, http://ubiquity.acm.org/article.cfm?id=1899473.

Bajcsy, R. K., A. B. Borodin, B. H. Liskov, and J. D. Ullman (1992, September). Computer science statewide review draft preface. Technical report, Computer Science Rating Committee. Confidential Report and Recommendations to the Commissioner of Education of the State of New York (unpublished), http://www.cse.buffalo.edu/~rapaport/Papers/Papers.by.Others/bajcsyetal92.pdf.

Baldwin, J. (1962). The creative process. In N. C. Center (Ed.), *Creative America*. New York: Ridge Press. https://openspaceofdemocracy.files.wordpress.com/2017/01/baldwin-creative-process.pdf.

Ballard, D. H. (1997). *An Introduction to Natural Computation*. Cambridge, MA: MIT Press.

Baranger, W. R. (1995a, 7 June). J. presper eckert, co-inventor of early computer, dies at 76. *New York Times*, B12. http://www.nytimes.com/1995/06/07/obituaries/j-presper-eckert-co-inventor-of-early-computer-dies-at-76.html.

Baranger, W. R. (1995b, 17 June). John v. atanasoff, 91, dies; early computer researcher. *New York Times*, 11. http://www.nytimes.com/1995/06/17/obituaries/john-v-atanasoff-91-dies-early-computer-researcher.html.

Barba, L. A. (2016, 5 March). Computational thinking: I do not think it means what you think it means. http://lorenabarba.com/blog/computational-thinking-i-do-not-think-it-means-what-you-think-it-means/.

Barr, A. (1985). Systems that know that they don't understand. http://www.stanford.edu/group/scip/avsgt/cognitiva85.pdf.

Barwise, J. (1989a, April). For whom the bell rings and cursor blinks. *Notices of the American Mathematical Society 36*(4), 386–388.

Barwise, J. (1989b). Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society 36*, 844–851.

Battersby, S. (2015, 21 November). Moon could be a planet under new definition. *New Scientist 228*(3048), 9. https://www.newscientist.com/article/mg22830484-400-our-moon-would-be-a-planet-under-new-definition-of-planethood/.

Baum, L. F. (1900). *The Wizard of Oz*. CITY: PUB.

Bechtel, W. and A. Abrahamsen (2005). Explanation: A mechanistic alternative. *Studies in History and Philosophy of the Biological and Biomedical Sciences 36*, 421–441. http://mechanism.ucsd.edu/research/explanation.mechanisticalternative.pdf.

Becker, A. (2018). *What Is Real? The Unfinished Quest for the Meaning of Quantum Physics*. London: John Murray.

Beebe, J. R. (2011). Experimental epistemology research group. http://eerg.buffalo.edu/.

Beebee, H. (2017, 5 October). Who is rachel? blade runner and personal identity. *IAI [Institute of Art and Ideas] News*. https://iainews.iai.tv/articles/who-is-rachael-the-philosophy-of-blade-runner-and-memory-auid-885.

Benacerraf, P. (1965, January). What numbers could not be. *Philosophical Review 74*(1), 47–73.

Benacerraf, P. and H. Putnam (Eds.) (1984). *Philosophy of Mathematics: Selected Readings, 2nd Edition*. New York: Cambridge University Press.

Bender, D. (1985-1986). Protection of computer programs: The copyright/trade secret interface. *University of Pittsburgh Law Review 47*, 907–958.

Berlin, B. and P. Kay (1969). *Basic Color Terms: Their Universality and Evolution*. Chicago: University of Chicago press.

Berners-Lee, T., W. Hall, J. Hendler, N. Shadbolt, and D. J. Weitzner (2006, 11 August). Creating a science of the Web. *Science 313*, 769–771. 10.1126/science.1126902.

Berners-Lee, T., J. Hendler, and O. Lassila (2001, May). The semantic web. *Scientific American*. https://www-sop.inria.fr/acacia/cours/essi2006/Scientific%20American_%20Feature%20Article_%20The%20Semantic%20Web_%20May%202001.pdf.

Bernhardt, C. (2016). *Turing's Vision: The Birth of Computer Science*. Cambridge, MA: MIT Press.

Bernstein, J. (1986, 20 January). A portrait of Alan Turing. *New Yorker*, 78, 81–87. http://www.cdpa.co.uk/Newman/MHAN/view-item.php?Box=5&Folder=6&Item=5&Page=1.

Bernstein, J. and J. Holt (2016, 8 December). Spooky physics up close: An exchange. *New York Review of Books' 63*(19), 62. http://www.nybooks.com/articles/2016/12/08/spooky-physics-up-close-exchange/.

Bernstein, J. and L. Krauss (2016, 24 November). Walking like a black hole. *New York Review of Books' 63*(18), 74. http://www.nybooks.com/articles/2016/11/24/walking-like-a-black-hole/.

Bickle, J. (2015). Marr and reductionism. *Topics in Cognitive Science*. DOI: 10.1111/TOPS.12134.

Bickle, J. (2019). Multiple realizability. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2019 ed.). Metaphysics Research Lab, Stanford University.

Biermann, A. (1990). *Great Ideas in Computer Science: A Gentle Introduction*. Cambridge, MA: MIT Press.

Blachowicz, J. (2016, 4 July). There is no scientific method. *New York Times*. http://www.nytimes.com/2016/07/04/opinion/there-is-no-scientific-method.html.

Blass, A. and Y. Gurevich (2003, October). Algorithms: A quest for absolute definitions. *Bulletin of the European Association for Theoretical Computer Science (EATCS) 81*, 195–225. Page references are to the online version at http://research.microsoft.com/en-us/um/people/gurevich/opera/164.pdf; reprinted in Olszewski et al. 2006, pp. 24–57.

Blessing, K. (2013, January/February). I re-read, therefore i understand. *Philosophy Now 94*, 17. http://philosophynow.org/issues/94/I_Re-Read_Therefore_I_Understand.

Block, N. (1978). Troubles with functionalism. In C. Savage (Ed.), *Minnesota Studies in the Philosophy of Science, Volume 9*, pp. 261–325. Minneapolis: University of Minnesota Press. http://mcps.umn.edu/philosophy/9_12Block.pdf.

Blum, L. (2004, October). Computing over the reals: Where Turing meets Newton. *Notices of the AMS 51*(9), 1024–1034. Illustrated preprint at http://www.cs.cmu.edu/~lblum/PAPERS/TuringMeetsNewton.pdf, published version at https://www.ams.org/notices/200409/fea-blum.pdf.

Blum, L., M. Shub, and S. Smale (1989). On a theory of computation and complexity over the real numbers: *NP*-completeness, recursive functions, and universal machines. *Bulletin of the American Mathematical Society 21*(1), 1–46. https://projecteuclid.org/download/pdf_1/euclid.bams/1183555121.

Boden, M. A. (1977). *Artificial Intelligence and Natural Man*. New York: Basic Books.

Boden, M. A. (1990a). Has AI helped psychology? In D. Partridge and Y. Wilks (Eds.), *The Foundations of Artificial Intelligence: A Sourcebook*, pp. 108–111. Cambridge, UK: Cambridge University Press.

Boden, M. A. (Ed.) (1990b). *The Philosophy of Artificial Intelligence*. Oxford: Oxford University Press.

Boden, M. A. (2006). *Mind as Machine: A History of Cognitive Science*. Oxford: Oxford University Press.

Böhm, C. and G. Jacopini (1966, May). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM 9*(5), 366–371. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.9119&rep=rep1&type=pdf.

Bolden, C. (2016, September). Katherine johnson, the nasa mathematician who advanced human rights with a slide rule and pencil. *Vanity Fair*. http://www.vanityfair.com/culture/2016/08/katherine-johnson-the-nasa-mathematician-who-advanced-human-rights.

Bond, G. W. (2005, August). Software as art. *Communications of the ACM 48*(8), 118–124.

Boole, G. (2009). *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities*. Cambridge, UK: Cambridge University Press.

Boolos, G. S. and R. C. Jeffrey (1974). *Computability and Logic*. Cambridge, UK: Cambridge University Press.

Boorstin, D. (1983). *The Discoverers*. New York: Random House. Ch. 49: "The Microscope of Nature".

Borbely, R. (2005, March). Letter to the editor. *Scientific American*, 12, 15.

Borges, J. L. (1981). Partial enchantments of the Quixote. In E. R. Monegal and A. Reid (Eds.), *Borges, A Reader; A Selection from the Writings of Jorge Luis Borges*, pp. 232–235. New York: E.P. Dutton.

Bostrom, N. (2003). Are you living in a computer simulation? *Philosophical Quarterly 53*(211), 243–255. http://www.simulation-argument.com/.

Bowie, G. L. (1973, 8 February). An argument against Church's thesis. *Journal of Philosophy 70*(3), 66–76.

Bowles, N. (2019, 1 April). A journey—if you dare—into the minds of silicon valley programmers. *New York Times Book Review*. https://www.nytimes.com/2019/04/01/books/review/clive-thompson-coders.html.

Boyle, J. (2009, September). What intellectual property law should learn from software. *Communications of the ACM 52*(9), 71–76. http://www.thepublicdomain.org/2009/08/26/what-intellectual-property-law-should-learn-from-software/.

Brachman, R. J. (2002, November/December). Systems that know what they're doing. *IEEE Intelligent Systems*, 67–71.

Brassard, G. (1995, March). Time for another paradigm shift. *ACM Computing Surveys 27*(1), 19–21.

Brenner, S. (2012, 14 December). The revolution in the life sciences. *Science 338*, 1427–1428.

Bringsjord, S. (11 August 2006). What is computer science? constraints on acceptable answers . http://www.cse.buffalo.edu/~rapaport/Papers/Papers.by.Others/bringsjord06-constraints_on_philofcompsci.pdf.

Bringsjord, S. (1993). The narrational case against Church's thesis. http://homepages.rpi.edu/~brings/SELPAP/CT/ct/.

Bringsjord, S. (2015). A vindication of program verification. *History and Philosophy of Logic 36*(3), 262–277. http://kryten.mm.rpi.edu/SB_progver_selfref_driver_final2_060215.pdf.

Bringsjord, S. and K. Arkoudas (2004). The modal argument for hypercomputing minds. *Theoretical Computer Science 317*, 167–190. doi:10.1016/j.tcs.2003.12.010; http://kryten.mm.rpi.edu/modal.hypercomputing.pdf.

Bringsjord, S. and N. S. Govindarajulu (2018). Artificial intelligence. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2018 ed.). Metaphysics Research Lab, Stanford University.

Bringsjord, S., N. S. Govindarajulu, S. Banerjee, and J. Hummel (2018). Do machine-learning machines learn? In V. Müller (Ed.), *Philosophy and Theory of Artificial Intelligence 2017; PT-AI 2017*, pp. 136–157. Cham, Switzerland: Springer.

Bronowski, J. (1958, September). The creative process. *Scientific American 199*.

Brooks, Jr., F. P. (1975). *The Mythical Man-Month*. Reading, MA: Addison-Wesley.

Brooks, Jr., F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Boston: Addison-Wesley.

Brooks, Jr., F. P. (1996, March). The computer scientist as toolsmith ii. *Communications of the ACM 39*(3), 61–68. Revised and extended verson of Brooks Jr., Frederick P. (1977), "The Computer Scientist as Toolsmith—Studies in Interactive Computer Graphics", in B. Gilchrist (ed.), *Information Processing 77, Proceedings of IFIP Congress 77 (Toronto)* (Amsterdam: North-Holland): 625–634; http://www.cs.unc.edu/techreports/88-041.pdf.

Brooks, R. A. (1991). Intelligence without representation. *Artificial Intelligence 47*, 139–159.

Brown, R. (2004). Consideration of the origin of herbert simon's theory of 'satisficing' (1933–1947). *Management Decision 42*(10), 1240–1256. https://doi.org/10.1108/00251740410568944.

Brown, R. (4 August 1992). contribution to newsgroup discussion of cyc. Article 13305 of comp.ai; http://www.cse.buffalo.edu/~rapaport/definitions.of.ai.html.

Buchanan, B. G. (2006, Winter). What do we know about knowledge? *AI Magazine*, 35–46.

Buechner, J. (2011). Not even computing machines can follow rules: Kripke's critique of functionalism. In A. Berger (Ed.), *Saul Kripke*, pp. 343–367. New York: Cambridge University Press.

Buechner, J. (2018, Spring). Does Kripke's argument against functionalism undermine the standard view of what computers are? *Minds and Machines 28*(3), 491–513.

Bueno, O. (2014). Nominalism in the philosophy of mathematics. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2014 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2014/entries/nominalism-mathematics/.

Bullynck, M., E. G. Daylight, and L. Mol (2015, March). Why did computer science make a hero out of Turing? *Communications of the ACM 58*(3), 37–39. http://cacm.acm.org/magazines/2015/3/183592-why-did-computer-science-make-a-hero-out-of-turing/fulltext.

Bundy, A. (1983, Winter). The nature of ai: A reply to schank. *AI Magazine 4*(4), 29–31.

Bundy, A. and S. Ohlsson (1990). The nature of AI principles. In D. Partridge and Y. Wilks (Eds.), *The Foundations of Artificial Intelligence: A Sourcebook*, pp. 1325–154. Cambridge, UK: Cambridge University Press.

Bunge, M. (1974). Toward a philosophy of technology. In A. C. Michalos (Ed.), *Philosophical Problems of Science and Technology*, pp. 28–47. Boston: Allyn & Bacon.

Burge, T. (1979). Individualism and the mental. *Midwest Studies in Philosopy 4*, 73–121.

Burge, T. (1986, January). Individualism and psychology. *Philosophical Review 95*(1), 3–45.

Burkholder, L. (1999). Are ai and mechanics empirical disciplines? *Journal of Experimental and Theoretical Artificial Intelligence 11*, 497–500. DOI:10.1080/095281399146373.

Burks, A. W. (Ed.) (1970). *Essays on Cellular Automata*. Urbana, IL: University of Illinois Press.

Button, T. (2009). Sad computers and two versions of the church-turing thesis. *British Journal for the Philosophy of Science 60*, 765–792.

Buzen, J. P. (2011, January). Computation, uncertainty and risk. *Ubiquity 2011*(January). Article 5, http://ubiquity.acm.org/article.cfm?id=1936886.

Campbell, D. I. and Y. Yang (2019). Does the solar system computer the laws of motion? *Synthese*. https://doi.org/10.1007/s11229-019-02275-w.

Campbell, M. (2018, 7 December). Mastering board games. *Science 362*(6419), 1118.

Campbell-Kelly, M. (2009, September). Origin of computing. *Scientific American*, 62–69. http://www.cs.virginia.edu/~robins/The_Origins_of_Computing.pdf. See also "Readers Respond on the 'Origin of Computing' ", *Scientific American* (January 2010): 8, http://www.scientificamerican.com/article.cfm?id=letters-jan-10.

Campbell-Kelly, M. (2010, April). Be careful what you wish for: Reflections on the decline of mathematical tables. *Communications on the ACM 53*(4), 25–26.

Campbell-Kelly, M. (2011, September). In praise of 'wilkes, wheeler, and gill'. *Communications of the ACM 54*(9), 25–27. http://dx.doi.org/10.1145/1995376.1995386.

Campbell-Kelly, M. (2012, July). Alan Turing's other universal machine. *Communications of the ACM 55*(7), 31–33.

Campos, D. G. (2011). On the distinction between peirce's abduction and lipton's inference to the best explanation. *Synthese 180*, 419–442.

Cane, S. (2014, 27 February). Interview: David chalmers and andy clark. *New Philosopher 2:mind*. http://www.newphilosopher.com/articles/interview-david-chalmers-and-andy-clark/.

Cannon, P. (2013, March/April). Kant at the bar: Transcendental idealism in daily life. *Philosophy Now Issue 95*, 15–17. http://philosophynow.org/issues/95/Kant_at_the_Bar_Transcendental_Idealism_in_Daily_Life.

Cardoso, S. H. (1997). Specialized functions of the cerebral cortex. http://www.cerebromente.org.br/n01/arquitet/cortex_i.htm.

Care, C. (2007, May). Not only digital: A review of ACM's early involvement with analog computing technology. *Communications of the ACM 50*(5), 42–45.

Carey, B. (2019, 26 June). 'it's gigantic': A new way to gauge the chances for unresponsive patients. *New York Times*. https://www.nytimes.com/2019/06/26/health/brain-injury-eeg-consciousness.html.

Carey, K. (2010, 12 November). Decoding the value of computer science. *Chronicle of Higher Education 58*(12), A88. http://www.chronicle.com/article/Decoding-the-Value-of-Computer/125266/.

Carhart, R. R. (1956). The systems approach to reliability. In *Proceedings, 2nd National Symposium on Quality Control and Reliabillity in Electronics, January 9–10*, pp. 149–155. Washington, DC: IRE Professional Group on Reliability and Quality Control, American Society for Quality Control, Electronics Technical Committee, Institute of Radio Engineers.

Carleton, L. R. (1984). Programs, language understanding, and searle. *Synthese 59*, 219–230.

Carlson, B., A. Burgess, and C. Miller (1996). Timeline of computing history. http://www.computer.org/cms/Computer.org/Publications/timeline.pdf.

Carnap, R. (1956). *Meaning and Necessity: A Study in Semantics and Modal Logic, Second Edition*. Chicago: University of Chicago Press.

Carpenter, B. and R. Doran (1977). The other Turing machine. *The Computer Journal 20*(3), 269–279. http://comjnl.oxfordjournals.org/content/20/3/269.full.pdf+html.

Carroll, L. (1850). Difficulties no. 2. In *The Rectory Umbrella and Mischmasch*. Dover, 1971. http://etc.usf.edu/lit2go/112/poems-puzzles-and-stories-of-lewis-carroll/4953/the-two-clocks/.

Carroll, L. (1871). Through the looking-glass. http://www.gutenberg.org/files/12/12-h/12-h.htm.

Carroll, L. (1893). *Sylvie and Bruno Concluded*. London: Macmillan. http://www.gutenberg.org/files/48795/48795-h/48795-h.htm and especially https://gisandscience.com/2009/10/28/quote-of-the-day-lewis-carrolls-paradox-of-the-complete-map/.

Carroll, L. (1895, April). What the tortoise said to achilles. *Mind 4*(14), 278–280. http://www.ditext.com/carroll/tortoise.html.

Caryl, C. (2015, 5 February). Saving Alan Turing from his friends. *New York Review of Books*, 19–21. http://www.nybooks.com/articles/archives/2015/feb/05/saving-alan-turing-his-friends/.

Casati, R. (2000). *Shadows: Unlocking Their Secrets, from Plato to Our Time*. New York: Vintage/Random House, 2004. Translated by Abigail Asher.

Case, J. (nd). Motivating the proof of the Kleene recursion theorem. http://www.eecis.udel.edu/~case/papers/krt-self-repo.pdf.

Casner, S. M., E. L. Hutchins, and D. Norman (2016, May). The challenges of partially automated driving. *Communications of the ACM 59*(5), 70–77. https://cacm.acm.org/magazines/2016/5/201592-the-challenges-of-partially-automated-driving/fulltext.

Casselman, B. (2014, September). About the cover: Smart card. *Notices of the AMS 61*(8), 872. http://www.ams.org/notices/201408/201408-full-issue.pdf.

Castañeda, H.-N. (1975, April). Individuation and non-identity: A new look. *American Philosophical Quarterly 12*(2), 131–140.

Castañeda, H.-N. (1989). Direct reference, the semantics of thinking, and guise theory (constructive reflections on David Kaplan's theory of indexical reference). In J. Almog, J. Perry, and H. Wettstein (Eds.), *Themes from Kaplan*, pp. 105–144. New York: Oxford University Press.

Cath, Y. (2019). Knowing how. *Analysis*. https://doi.org/10.1093/analys/anz027.

Cathcart, T. and D. Klein (2007). *Plato and a Platypus Walk into a Bar: Understanding Philosophy through Jokes*. New York: Abrams Image.

Cerf, V. G. (2012, December). Computer science revisited. *Communications of the ACM 55*(12), 7. http://ubiquity.acm.org/article.cfm?id=2406359.

Cerf, V. G. (2013, January). What's a robot? *Communications of the ACM 56*(1), 7. http://www.cs.grinnell.edu/~davisjan/csc/105/2013S/articles/CACM-reliability.pdf.

Cerf, V. G. (2014, January). Virtual reality redux. *Commmunications of the ACM 57*(1), 7. http://cacm.acm.org/magazines/2014/1/170860-virtual-reality-redux/fulltext.

Cerf, V. G. (2015, February). There is nothing new under the sun. *Communications of the ACM 58*(2), 7. https://cacm.acm.org/magazines/2015/2/182649-there-is-nothing-new-under-the-sun/fulltext.

Cerf, V. G. (2016, March). Computer science in the curriculum. *Communications of the ACM 59*(3), 7. http://cacm.acm.org/magazines/2016/3/198866-computer-science-in-the-curriculum/fulltext.

Cerf, V. G. (2017, 13 July). Information technology: A digital genius at play. *Nature 547*(7662), 159. https://www.nature.com/nature/journal/v547/n7662/pdf/547159a.pdf.

Ceruzzi, P. (1988). Electronics technology and computer science, 1940–1975: A coevolution. *Annals of the History of Computing 10*(4), 257–275.

Chaitin, G. (2006a). How real are real numbers? *International Journal of Bifurcation and Chaos*. http://www.cs.auckland.ac.nz/~chaitin/olympia.pdf (2006 version); http://www.umcs.maine.edu/~chaitin/wlu.html (2009 version).

Chaitin, G. (2006b, March). The limits of reason. *Scientific American*, 74–81. http://www.cs.virginia.edu/~robins/The_Limits_of_Reason_Chaitin_2006.pdf.

Chaitin, G. J. (1987). Computing the busy beaver function. In T. Cover and B. Gopinath (Eds.), *Open Problems in Communication and Computation*, pp. 108–112. Springer. http://www.cs.auckland.ac.nz/~chaitin/bellcom.pdf.

Chaitin, G. J. (2002, March-April). Computers, paradoxes and the foundations of mathematics. *American Scientist 90*, 164–171. http://www.cs.ox.ac.uk/activities/ieg/e-library/sources/amsci.pdf.

Chalmers, D. J. (1995). On implementing a computation. *Minds and Machines 4*, 391–402.

Chalmers, D. J. (1996a). *The Conscious Mind: In Search of a Fundamental Theory*. New York: Oxford University Press. http://tinyurl.com/plv877.

Chalmers, D. J. (1996b). Does a rock implement every finite-state automaton? *Synthese 108*, 309–333. http://consc.net/papers/rock.html; Chalmers corrects "an error in my arguments" in Chalmers 2012, pp. 236–238.

Chalmers, D. J. (2011, October-December). A computational foundation for the study of cognition. *Journal of Cognitive Science (South Korea) 12*(4), 323–357. http://cogsci.snu.ac.kr/jcs/issue/vol12/no4/01Chalmers.pdf.

Chalmers, D. J. (2012, July-September). The varieties of computation: A reply. *Journal of Cognitive Science (South Korea) 13*(3), 211–248. http://cogsci.snu.ac.kr/jcs/issue/vol13/no3/01+David+J+Chalmers.pdf.

Chalmers, D. J. (2015). Why isn't there more progress in philosophy? *Philosophy 90*(1), 3–31. http://consc.net/papers/progress.pdf.

Chalmers, D. J. (2019). The meta-problem of consciousness. *Journal of Consciousness Studies 25*(9–10), 6–61. https://philpapers.org/archive/CHATMO-32.pdf.

Chandra, V. (2014). *Geek Sublime: The Beauty of Code, the Code of Beauty*. Graywolf Press.

Changizi, M. A., A. Hsieh, R. Nijhawan, R. Kanai, and S. Shimojo (2008). Perceiving the present and a systematization of illusions. *Cognitive Science 32*, 459–503. http://onlinelibrary.wiley.com/doi/10.1080/03640210802035191/epdf.

Chase, G. C. (1980, July). History of mechanical computing machinery. *Annals of the History of Computing 2*(3), 198–226.

Chater, N. and M. Oaksford (2013, August). Programs as causal models: Speculations on mental programs and mental representation. *Cognitive Science 37*(6), 1171–1191.

Chazelle, B. (2006). The algorithm: Idiom of modern science. http://www.cs.princeton.edu/~chazelle/pubs/algorithm.html.

Chetty, R. (2013, 21 October). Yes, economics is a science. *New York Times*, A21. http://www.nytimes.com/2013/10/21/opinion/yes-economics-is-a-science.html.

Chiang, T. (2002). Seventy-two letters. In *Stories of Your Life and Others*, pp. 147–200. New York: Vintage. http://will.tip.dhappy.org/revolution/Technoanarchist/plan/.../book/Ted%20Chiang%20-%20Seventy-Two%20Letters/Ted%20Chiang%20-%20Seventy-Two%20Letters.html.

Chirimuuta, M., T. Boone, and M. DeMedonsa (2014, 19 September). Is your brain a computer? (video). *Instant HPS*. https://www.youtube.com/watch?v=-8q_UXpHsaY.

Chisholm, R. (1974). *Metaphysics, 2nd edition*. Englewood Cliffs, NJ: Prentice-Hall.

Chisum, D. S. (1985-1986). The patentability of algorithms. *University of Pittsburgh Law Review 47*, 959–1022.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. Cambridge, MA: MIT Press. https://faculty.georgetown.edu/irvinem/theory/Chomsky-Aspects-excerpt.pdf.

Chomsky, N. (2017). The galilean challenge. *Inference: International Review of Science 3*(1). http://inference-review.com/article/the-galilean-challenge.

Chow, S. J. (2015). Many meanings of 'heuristic'. *British Journal for the Philosophy of Science 66*, 977–1016.

Church, A. (1933, October). A set of postulates for the foundation of logic (second paper). *Annals of Mathematics, Second Series 34*(4), 839–864. See also the "first" version, Vol. 33, No. 2 (April 1932): 346–366, https://docs.google.com/file/d/0B0CU-A1oqzzLd3VfWm1ja1E2WDQ/view.

Church, A. (1936a, March). A note on the entscheidungsproblem. *Journal of Symbolic Logic 1*(1), 40–41. See also "Correction to *A Note on the Entscheidungsproblem*", in *Journal of Symbolic Logic* 1(3) (September): 101–102.

Church, A. (1936b, April). An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*(2), 345–363. http://phil415.pbworks.com/f/Church.pdf.

Church, A. (1937, March). Review of Turing (1936). *Journal of Symbolic Logic 2*(1), 42–43. For commentary on this review, see Hodges (2013).

Church, A. (1940). On the concept of a random sequence. *Bulletin of the American Mathematical Society 2*, 130–135. https://projecteuclid.org/download/pdf_1/euclid.bams/1183502434.

Church, A. (1956). *Introduction to Mathematical Logic*. Princeton, NJ: Princeton University Press.

Churchland, P. S. and T. J. Sejnowski (1992). *The Computational Brain*. Cambridge, MA: MIT Press.

Clark, A. and D. J. Chalmers (1998). The extended mind. *Analysis 58*, 10–23. https://icds.uoregon.edu/wp-content/uploads/2014/06/Clark-and-Chalmers-The-Extended-Mind.pdf.

Clark, K. L. and D. F. Cowell (1976). *Programs, Machines, and Computation: An Introduction to the Theory of Computing*. London: McGraw-Hill.

Clarke, A. C. (1951, Spring). Sentinel of eternity. *[Avon] 10 Story Fantasy 3*(1), 41–51. https://archive.org/stream/10_Story_Fantasy_v01n01_1951-Spring_Tawrast-EXciter#page/n39/mode/2up.

Clarke, A. C. (1953). *Childhood's End*. CITY: PUB.

Cleland, C. E. (1993, August). Is the Church-Turing thesis true? *Minds and Machines 3*(3), 283–312.

Cleland, C. E. (1995). Effective procedures and computable functions. *Minds and Machines 5*(1), 9–23.

Cleland, C. E. (2001, May). Recipes, algorithms, and programs. *Minds and Machines 11*(2), 219–237.

Cleland, C. E. (2002, May). On effective procedures. *Minds and Machines 12*(2), 159–179.

Cleland, C. E. (2004). The concept of computability. *Theoretical Computer Science 317*, 209–225.

Cockshott, P. and G. Michaelson (2007). Are there new models of computation? Reply to Wegner and Eberbach. *The Computer Journal 50*(2), 232–247. http://www.dcs.gla.ac.uk/~wpc/reports/wegner25aug.pdf.

Coffa, J. (1991). *The Semantic Tradition from Kant to Carnap: To the Vienna Station*. Cambridge, UK: Cambridge University Press.

Cohen, M. R. and E. Nagel (1934). *An Introduction to Logic and Scientific Method*. New York: Harcourt, Brace and Co.

Colburn, T. R. (1991). Program verification, defeasible reasoning, and two views of computer science. *Minds and Machines 1*, 97–116. Reprinted in Colburn et al. 1993, pp. 375–399.

Colburn, T. R. (1993). Computer science and philosophy. In T. R. Colburn, J. H. Fetzer, and T. L. Rankin (Eds.), *Program Verification: Fundamental Issues in Computer Science*, pp. 3–31. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Colburn, T. R. (1999). Software, abstraction, and ontology. *The Monist 82*(1), 3–19.

Colburn, T. R. (2000). *Philosophy and Computer Science*. Armonk, NY: M.E. Sharpe.

Colburn, T. R., J. H. Fetzer, and T. L. Rankin (Eds.) (1993). *Program Verification: Fundamental Issues in Computer Science*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Colby, K. M. (1981). Modeling a paranoid mind. *Behavioral and Brain Sciences 4*, 515–560.

Colby, K. M., F. D. Hilf, S. Weber, and H. C. Kraemer (1972). Turing-like indistinguishability tests for the validation of a computer simulation of paranoid processes. *Artificial Intelligence 3*, 199–221.

Colby, K. M., S. Weber, and F. D. Hilf (1971). Artificial paranoia. *Artificial Intelligence 2*, 1–25.

Cole, D. (1991, September). Artificial intelligence and personal identity. *Synthese 88*(3), 399–417.

Cole, D. (2014, 31 December). Alan Turing & the Chinese Room Argument. https://web.archive.org/web/20150109080357/http://www.thecritique.com/articles/alan-turing-the-chinese-room-argument/.

Cole, D. (2019). The chinese room argument. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2019 ed.). Metaphysics Research Lab, Stanford University.

Collins, A. M. and M. R. Quillian (1972). How to make a language user. In E. Tulving and W. Donaldson (Eds.), *Organization of Memory*, pp. 309–351. New York: Academic Press.

Comte, A. (1830). *Cours de philosophie positive (Course in Positive Philosophy)*. Paris: Bachelier. English trans. by Harriet Martineau, https://archive.org/details/positivephilosop01comtuoft; also in James B. Hartman (ed.), *Philosophy of Recent Times, Vol. I: Readings in Nineteenth-Century Philosophy* (New York: McGraw-Hill, 1967): 137–143.

Condliffe, J. (2019, 22 March). The week in tech: Our future robots will need super-smart safety checks. *New York Times*. https://www.nytimes.com/2019/03/22/technology/robot-safety-regulation.html.

Conery, J. S. (2010, November). What is computation? Computation is symbol manipulation. *Ubiquity 2010*(November). Article 4, http://ubiquity.acm.org/article.cfm?id=1889839.

Conte, P. T. (1989, July). More on verification (letter to the editor). *Communications of the ACM 32*(7), 790.

Cook, B., A. Podelski, and A. Rybalchenko (2011, May). Proving program termination. *Communications of the ACM 54*(5), 88–98.

Cook, S. A. (1983, June). An overview of computational complexity. *Communications of the ACM 26*(6), 400–408. http://www.jdl.ac.cn/turing/pdf/p400-cook.pdf.

Cooper, S. B. (2004). *Computability Theory*. Boca Raton, FL: Chapman & Hall.

Cooper, S. B. (2006). Review of Leavitt (2005). *Notices of the AMS 53*(10), 1213–1217. http://www.ams.org/notices/200610/rev-cooper.pdf.

Cooper, S. B. (2012, March). Turing's titanic machine? *Communications of the ACM 55*(3), 74–83. http://cacm.acm.org/magazines/2012/3/146259-turings-titanic-machine/fulltext.

Cooper, S. B. and J. van Leeuwen (Eds.) (2013). *Alan Turing: His Work and Impact*. Amsterdam: Elsevier.

Copeland, B. J. (1996). What is computation? *Synthese 108*, 335–359. http://www.alanturing.net/turing_archive/pages/pub/what/what.pdf.

Copeland, B. J. (1997, May). The broad conception of computation. *American Behavioral Scientist 40*(6), 690–716.

Copeland, B. J. (1998). Even Turing machines can compute uncomputable functions. In C. Calude, J. Casti, and M. J. Dinneen (Eds.), *Unconventional Models of Computtion*, pp. 150–164. Springer-Verlag. http://www.alanturing.net/turing_archive/pages/pub/even/even.pdf.

Copeland, B. J. (1999). The Turing-Wilkinson lecture series on the automatic computing engine. In K. Furukawa, D. Michi, and S. Muggleton (Eds.), *Machine Intelligence 15: Intelligent Agents*, pp. 381–444. Oxford: Oxford University Press.

Copeland, B. J. (2000, June). The Church-Turing thesis. http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html.

Copeland, B. J. (2002a, May). Accelerating turing machines. *Minds and Machines 12*(2), 303–326.

Copeland, B. J. (2002b, November). Hypercomputation. *Minds and Machines 12*(4), 461–502.

Copeland, B. J. (2004a). Computation. In L. Floridi (Ed.), *The Blackwell Guide to the Philosophy of Computing and Information*, pp. 3–17. Malden, MA: Blackwell.

Copeland, B. J. (Ed.) (2004b). *The Essential Turing*. Oxford: Oxford University Press.

Copeland, B. J. (2012, 12 November). Is Alan Turing both inventor of the basic ideas of the modern computer and a pioneer of artificial intelligence? *Big Questions Online*. https://www.bigquestionsonline.com/content/alan-turing-both-inventor-basic-ideas-modern-computer-and-pioneer-artificial-intelligence.

Copeland, B. J. (2013, 12 August). What Apple and Microsoft owe to Turing. *Huff[ington] Post Tech/The Blog*. http://www.huffingtonpost.com/jack-copeland/what-apple-and-microsoft-_b_3742114.html.

Copeland, B. J., E. Dresner, D. Proudfoot, and O. Shagrir (2016, November). Time to reinspect the foundations? *Communications of the ACM 59*(11), 34–36.

Copeland, B. J., T. Flowers, and . B. P. Codebreakers (2010). *Colossus: The Secrets of Bletchley Park's Codebreaking Computers*. New York: Oxford University Press. http://www.colossus-computer.com/contents.htm.

Copeland, B. J. and D. Proudfoot (1996). On Alan Turing's anticipation of connectionism. *Synthese 108*, 361–377. http://www.alanturing.net/turing_archive/pages/reference%20articles/connectionism/Turing'santicipation.html.

Copeland, B. J. and D. Proudfoot (1999, April). Alan turing's forgotten ideas in computer science. *Scientific American*, 98–103. http://www.cs.virginia.edu/~robins/Alan_Turing's_Forgotten_Ideas.pdf.

Copeland, B. J. and D. Proudfoot (2010, September). Deviant encodings and Turing's analysis of computability. *Studies in History and Philosophy of Science 41*(3), 247–252.

Copeland, B. J. and O. Shagrir (2011, Summer). Do accelerating Turing machines compute the uncomputable? *Minds and Machines 21*(2), 221–239.

Copeland, B. J. and O. Shagrir (2013). Turing versus Gödel on computability and the mind. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 1–33. Cambridge, MA: MIT Press.

Copeland, B. J. and R. Sylvan (1999, March). Beyond the universal Turing machine. *Australasian Journal of Philosophy 77*(1), 46–67. http://www.alanturing.net/turing_archive/pages/pub/beyond/beyond.pdf.

Copeland, B. J. and R. Sylvan (2000). Computability is logic-relative. In D. Hyde and G. Priest (Eds.), *Sociative Logics and Their Applications: Essays by the Late Richard Sylvan*, pp. 189–199. Ashgate.

Copes, L. (1980). College teaching, mathematical knowledge, and the perry development scheme. (St. Paul, MN: Institute for Studies in Educational Mathematics).

Corballis, M. C. (2007, May-Jue). The uniqueness of human recursive thinking. *American Scientist 95*, 240–248.

Corcoran, J. (2007). Scientific revolutions. In J. Lachs and R. Talisse (Eds.), *Encyclopedia of American Philosophy*. New York: Routledge. http://www.academia.edu/25802396/CORCORAN_ON_SCIENTIFIC_REVOLUTIONS.

Cornfeld, J. and L. Knefelkamp (1979). Combining student stage and type in the design of learning environments: An integration of perry stages and holland typologies. Paper presented at the American College Personnel Association, Los Angeles, March.

Corry, L. (2017, August). Turing's pre-war analog computers: The fatherhood of the modern computer revisited. *Communications of the ACM 60*(8), 50–58. https://cacm.acm.org/magazines/2017/8/219602-turings-pre-war-analog-computers/fulltext.

Covert, M. W. (2014, January). Simulating a living cell. *Scientific American 310*(1), 44–51.

Coward, L. A. and R. Sun (2004, June). Criteria for an effective theory of consciousness and some preliminary attempts. *Consciousness and Cognition 13*(2), 268–301. Preprint at http://www.cogsci.rpi.edu/~rsun/coward-sun-cc2003.pdf.

Craver, M. (2007, May). Letter to the editor. *Scientific American*, 16. Also see reply by Soter on same page.

Crawford, K. (2016, 25 June). Artificial intelligence's white guy problem. *New York Times*. https://www.nytimes.com/2016/06/26/opinion/sunday/artificial-intelligences-white-guy-problem.html.

Crichton, M. (2006, 19 March). This essay breaks the law. *New York Times*, WK13. http://www.nytimes.com/2006/03/19/opinion/19crichton.html.

Crossley, J. N. and A. S. Henry (1990). Thus spake al-k̲hwārizmī: A translation of the text of cambridge university library ms. ii.vi.5. *Historia Mathematica 17*, 103–131.

Crowcroft, J. (2005, February). On the nature of computing. *Communications of the ACM 48*(2), 19–20.

Curd, M. (2014, 22 July). Review of Pigliucci and Boudry (2013b). *Notre Dame Philosophical Reviews*. https://ndpr.nd.edu/news/49425-philosophy-of-pseudoscience-reconsidering-the-demarcation-problem/.

Curry, H. B. (1951). *Outlines of a Formalist Philosophy of Mathematics*. Amsterdam: North-Holland.

Curtis, M. (1965, January). A Turing machine simulator. *Journal of the ACM 12*(1), 1–13.

Daly, I. (2010, 24 February). Just like mombot used to make. *New York Times* (Dining Section), D1, D5. http://www.nytimes.com/2010/02/24/dining/24robots.html.

Darwin, C. (1872). *The Origin of Species*. New York: Signet Classics, 1958.

DATA-LINK, E. o. (1958, April). What's in a name? *Communications of the ACM 1*(4), 6.

David, M. (2009). The correspondence theory of truth. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Fall 2009 Edition)*. http://plato.stanford.edu/archives/fall2009/entries/truth-correspondence/.

Davidson, J. (2006, 21 August). Measure for measure: Exploring the mysteries of conducting. *The New Yorker*, 60–69.

Davies, D. W. (1999). Repairs to Turing's universal computing machine. In K. Furukawa, D. Michie, and S. Muggleton (Eds.), *Machine Intelligence 15: Intelligent Agents*, pp. 477–488. Oxford: Oxford University Press. http://tinyurl.com/lqero7l.

Davies, E. (2001). Building infinite machines. *British Journal for the Philosophy of Science 52*, 671–682. http://www.mth.kcl.ac.uk/staff/eb_davies/jphilsci.pdf.

Davis, M. (1995a). An historical preface to engineering ethics. *Science and Engineering Ethics 1*(1), 33–48.

Davis, M. (1996, April). Defining 'engineer:' how to do it and why it matters. *Journal of Engineering Education 85*(2), 97–101.

Davis, M. (1998). *Thinking Like an Engineer: Studies in the Ethics of a Profession*. New York: Oxford University Press.

Davis, M. (2011, November). Will software engineering ever be engineering? *Communications of the ACM 54*(11), 32–34.

Davis, M. D. (1958). *Computability & Unsolvability*. New York: McGraw-Hill.

Davis, M. D. (1990). Is mathematical insight algorithmic? *Behavioral and Brain Sciences 13*(4), 659–660. http://www.cs.nyu.edu/faculty/davism/penrose.ps.

Davis, M. D. (1993). How subtle is Gödel's theorem? more on Roger Penrose. *Behavioral and Brain Sciences 16*(3), 611. http://www.cs.nyu.edu/faculty/davism/penrose2.ps.

Davis, M. D. (1995b). Mathematical logic and the origin of modern computers. In R. Herken (Ed.), *The Universal Turing Machine: A Half-Century Survey, Second Edition*, pp. 135–158. Vienna: Springer-Verlag. https://fi.ort.edu.uy/innovaportal/file/20124/1/41-herken_ed._95_-_the_universal_turing_machine.pdf.

Davis, M. D. (2000, July-August). Overheard in the park. *American Scientist 88*, 366–367.

Davis, M. D. (2003, May-June). Paradoxes in paradise. *American Scientist 91*, 268–269. http://www.americanscientist.org/bookshelf/pub/paradoxes-in-paradise.

Davis, M. D. (2004). The myth of hypercomputation. In C. Teuscher (Ed.), *Alan Turing: The Life and Legacy of a Great Thinker*, pp. 195–212. Berlin: Springer. http://www1.maths.leeds.ac.uk/~pmt6sbc/docs/davis.myth.pdf.

Davis, M. D. (2006a). The Church-Turing thesis: Consensus and opposition. In A. Beckmann, U. Berger, B. Löwe, and J. Tucker (Eds.), *Logical Approaches to Computational Barriers: Second Conference on Computability in Europe, CiE 2006, Swansea, UK, June 30–July 5*, pp. 125–132. Berlin: Springer-Verlag Lecture Notes in Computer Science 3988.

Davis, M. D. (2006b, November). What is Turing reducibility? *Notices of the AMS 53*(10), 1218–1219. http://www.ams.org/notices/200610/whatis-davis.pdf.

Davis, M. D. (2006c). Why there is no such discipline as hypercomputation. *Applied Mathematics and Computation 178*, 4–7.

Davis, M. D. (2008, November-December). Touring Turing. *American Scientist 96*(6), 520.

Davis, M. D. (2012). *The Universal Computer: The Road from Leibniz to Turing; Turing Centenary Edition*. Boca Raton, FL: CRC Press/Taylor & Francis Group. also published as *Engines of Logic: Mathematicians and the Origin of the Computer* (New York: W.W. Norton, 2000).

Davis, M. D. and E. J. Weyuker (1983). *Computability, Complexity and Langauges*. New York: Academic Press.

Davis, P. J. and R. Hersh (1998). *The Mathematical Experience*. Boston: Houghton Mifflin.

Davis, R., P. Samuelson, M. Kapor, and J. Reichman (1996, March). A new view of intellectual property and software. *Communications of the ACM 39*(3), 21–30. http://people.csail.mit.edu/davis/cacm96.ps; a summary version of Samuelson et al. 1994.

Davis, R. M. (1977, 18 March). Evolution of computers and computing. *Science 195*, 1096–1102.

Dawkins, R. (2016). *The Selfish Gene: 40th Anniversary Edition*. Oxford: Oxford University Press.

Dawson, J. W. (2001, March). Review of an earlier edition of Davis (2012). *Bulletin of Symbolic Logic 7*(1), 65–66. http://www.math.ucla.edu/~asl/bsl/0701/0701-003.ps.

Daylight, E. G. (2013). Towards a historical notion of "Turing—the father of computer science". http://www.dijkstrascry.com/sites/default/files/papers/Daylightpaper91.pdf.

Daylight, E. G. (2014, October). A Turing tale. *Communications of the ACM 57*(10), 36–38. http://cacm.acm.org/magazines/2014/10/178787-a-turing-tale/fulltext.

Daylight, E. G. (2016). *Turing Tales*. Geel, Belgium: Lonely Scholar.

de Leeuw, K., E. Moore, C. Shannon, and N. Shapiro (1956). Computability by probabilistic machines. In C. Shannon and J. McCarthy (Eds.), *Automata Studies*, pp. 183–212. Princeton, NJ: Princeton University Press. Reviewed and summarized in Fischer 1970.

De Millo, R. A., R. J. Lipton, and A. J. Perlis (1979, May). Social processes and proofs of theorems and programs. *Communications of the ACM 22*(5), 271–280.

De Mol, L. and G. Primiero (2015). When logic meets engineering: Introduction to logical issues in the history and philosophy of computer science. *History and Philosophy of Logic 36*(3), 195–204. http://www.tandfonline.com/doi/pdf/10.1080/01445340.2015.1084183.

de Waal, F. (2016, 8 April). What I learned from tickling apes. *New York Times Sunday Review*. https://www.nytimes.com/2016/04/10/opinion/sunday/what-i-learned-from-tickling-apes.html.

Decker, A., A. Phelps, and C. Egert (2017, March-April). Disappearing happy little sheep: Changing the culture of computing education by infusing. *Educational Technology 57*(2), 50–54.

Dedekind, R. (1890). Letter to keferstein. In J. van Heijenoort (Ed.), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pp. 98–103. Cambridge, MA: Harvard University Press.

Defoe, D. (1719). *Robinson Crusoe*. New York: W.W. Norton, 1994.

DeJohn, J. and E. Dietrich (2003, October-December). Editorial: Subvert the dominant paradigm! a review of *computationalism: new directions*, edited by Matthias Scheutz. *Journal of Experimental and Theoretical Artificial Intelligence 15*(4), 375–382.

Delvaux, M. (2016, 31 May). Draft report with recommendations to the commission on civil law rules on robotics. http://www.europarl.europa.eu/sides/getDoc.do?pubRef=-//EP//NONSGML%2BCOMPARL%2BPE-582.443%2B01%2BDOC%2BPDF%2BV0//EN; European Parliament Committee on Legal Affairs.

Dembart, L. (1977, 8 May). Experts argue whether computers could reason, and if they should. *New York Times*, 1, 34.

Dennett, D. C. (1971). Intentional systems. *Journal of Philosophy 68*, 87–106. Reprinted in Daniel C. Dennett, *Brainstorms* (Montgomery, VT: Bradford Books): 3–22.

Dennett, D. C. (1975). Why the law of effect will not go away. *Journal for the Theory of Social Behaviour 5*(2), 169–188. Reprinted in Daniel C. Dennett, *Brainstorms* (Montgomery, VT: Bradford Books): 71-89, https://dl.tufts.edu/downloads/j9602b715?filename=2r36v862p.pdf.

Dennett, D. C. (1978, July). Why you can't make a computer feel pain. *Synthese 38*(3), 415–456. https://dl.tufts.edu/downloads/9w032f88p?filename=m039kh27m.pdf. Reprinted in Daniel C. Dennett, *Brainstorms* (Montgomery, VT: Bradford Books): 190–229.

Dennett, D. C. (1981). Three kinds of intentional psychology. In R. Healey (Ed.), *Reduction, Time and Reality*, pp. 37–61. Cambridge, UK: Cambridge University Press. https://dl.tufts.edu/file_assets/tufts:ddennett-1981.00001.

Dennett, D. C. (1982, June). Notes on prosthetic imagination. *Boston Review 7*(3), 3–7.

Dennett, D. C. (1987). *The Intentional Stance*. Cambridge, MA: MIT Press.

Dennett, D. C. (1995). *Darwin's Dangerous Idea*. New York: Simon & Schuster.

Dennett, D. C. (2009a, 16 June). Darwin's 'strange inversion of reasoning'. *Proceedings of the National Academy of Science 106, suppl. 1*, 10061–10065. http://www.pnas.org/cgi/doi/10.1073/pnas.0904433106. See also Dennett 2013b.

Dennett, D. C. (2009b). Intentional systems theory. In B. McLaughlin, A. Beckermann, and S. Walter (Eds.), *The Oxford Handbook of Philosophy of Mind*, pp. 339–350. Oxford: Oxford University Press. https://ase.tufts.edu/cogstud/dennett/papers/intentionalsystems.pdf.

Dennett, D. C. (2012, 2 March). Sakes and dints: And other definitions that philosophers really need not seek. *Times Literary Supplement*, 12–14. http://ase.tufts.edu/cogstud/papers/TLS2012.pdf.

Dennett, D. C. (2013a). *Intuition Pumps and Other Tools for Thinking*. New York: W.W. Norton.

Dennett, D. C. (2013b). Turing's 'strange inversion of reasoning'. In S. B. Cooper and J. van Leeuwen (Eds.), *Alan Turing: His Work and Impact*, pp. 569–573. Amsterdam: Elsevier. See also Dennett 2009a.

Dennett, D. C. (2017). *From Bacteria to Bach and Back: The Evolution of Mind*. New York: W.W. Norton.

Denning, P. J. (1985, January-February). What is computer science? *American Scientist 73*, 16–19.

Denning, P. J. (1995, March). Can there be a science of information? *ACM Computing Surveys 27*(1), 23–25. https://pdfs.semanticscholar.org/6a53/d9d9e78685093de13017008ca54327a121b0.pdf.

Denning, P. J. (2000). Computer science: The discipline. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, Fourth Edition*. New York: Grove's Dictionaries. Page references are to 1999 preprint at http://cs.gmu.edu/cne/pjd/PUBS/ENC/cs99.pdf.

Denning, P. J. (2003, November). Great principles of computing. *Communications of the ACM 46*(11), 15–20.

Denning, P. J. (2005, April). Is computer science science? *Communications of the ACM 48*(4), 27–31.

Denning, P. J. (2007, July). Computing is a natural science. *Communications of the ACM 50*(7), 13–18.

Denning, P. J. (2009). Beyond computational thinking. *Communications of the ACM 52*(6), 28–30.

Denning, P. J. (2010, November). What is computation? opening statement. *Ubiquity 2010*(November). Article 1, http://ubiquity.acm.org/article.cfm?id=1880067.

Denning, P. J. (2013a, December). Design thinking. *Communications of the ACM 56*(12), 29–31. http://denninginstitute.com/pjd/PUBS/CACMcols/cacmDec13.pdf.

Denning, P. J. (2013b, May). The science in computer science. *Communications of the ACM 56*(5), 35–38.

Denning, P. J. (2017, June). Remaining trouble spots with computational thinking. *Communications of the ACM 60*(6), 33–39. https://m.cacm.acm.org/magazines/2017/6/217742-remaining-trouble-spots-with-computational-thinking/fulltext; see also Glass and Paulson 2017.

Denning, P. J. and T. Bell (2012, November-December). The information paradox. *American Scientist 100*, 470–477. http://denninginstitute.com/pjd/PUBS/AmSci-2012-info.pdf.

Denning, P. J., D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young (1989, January). Computing as a discipline. *Communications of the ACM 32*(1), 9–23.

Denning, P. J. and P. A. Freeman (2009, December). Computing's paradigm. *Communications of the ACM 52*(12), 28–30.

Denning, P. J. and C. H. Martell (2015). *Great Principles of Computing*. Cambridge, MA: MIT Press.

Denning, P. J. and R. D. Riehle (2009, March). Is software engineering engineering? *Communications of the ACM 52*(3), 24–26. http://denninginstitute.com/pjd/PUBS/CACMcols/cacmMar09.pdf.

Denning, P. J. and P. S. Rosenbloom (2009, September). Computing: The fourth great domain of science. *Communications of the ACM 52*(9), 27–29.

Denning, P. J., M. Tedre, and P. Yongpradit (2017, March). Misconceptions about computer science. *Communications of the ACM 60*(3), 31–33. http://denninginstitute.com/pjd/PUBS/CACMcols/cacmMar17.pdf.

Denning, P. J. and P. Wegner (2010, October). What is computation? *Ubiquity 2010*(October). http://ubiquity.acm.org/article.cfm?id=1870596.

Dershowitz, N. and Y. Gurevich (2008, September). A natural axiomatization of computability and proof of Church's thesis. *Bulletin of Symbolic Logic 14*(3), 299–350. http://research.microsoft.com/pubs/70459/tr-2007-85.pdf.

Descartes, R. (1637). Discourse on method. In E. S. Haldane and G. Ross (Eds.), *The Philosophical Works of Descartes*, pp. 79–130. Cambridge, UK: Cambridge University Press, 1970.

Deutsch, D. (1985). Quantum theory, the Church-Turing principle and the universal computer. *Proceedings of the Royal Society of London A 400*, 97–117. Page references are to preprint at http://www.cs.berkeley.edu/~christos/classics/Deutsch_quantum_theory.pdf.

Deutsch, M. (2009, September). Experimental philosophy and the theory of reference. *Mind & Language 24*(4), 445–466.

Devitt, M. and N. Porot (2018). The reference of proper names: Testing usage and intuitions. *Cognitive Science*. https://doi.org/10.1111/cogs.12609.

Devlin, K. (1992, November). Computers and mathematics (column). *Notices of the American Mathematical Society 39*(9), 1065–1066.

Dewar, R. and O. Astrachan (2009, July). CS education in the U.S.: Heading in the wrong direction?:. *Communications of the ACM 52*(7), 41–45.

Dewdney, A. (1989). *The Turing Omnibus: 61 Excursions in Computer Science*. Rockville, MD: Computer Science Press. esp. Chs. 1 ("Algorithms: Cooking Up Programs"), 28 ("Turing Machines: The Simplest Computers"), and 48 ("Universal Turing Machines: Computers as Programs").

Dewey, J. (1910). *How We Think: A Restatement of Reflective Thinking to the Educative Process, revised ed.* Boston: D.C. Heath. https://archive.org/details/howwethink000838mbp and http://rci.rutgers.edu/~tripmcc/philosophy/dewey-hwt-pt1-selections.pdf.

Diakopoulos, N. (2016, February). Accountability in algorithmic decision making. *Communications of the ACM 59*(2), 56–62.

Dick, P. K. (1968). *Do Androids Dream of Electric Sheep?* New York: Doubleday. Reprinted by Random House/Del Rey/Ballantine, 1996.

Dietrich, E. (2001, October). Homo sapiens 2.0: Why we should build the better robots of our nature. *Journal of Experimental and Theoretical Artificial Intelligence 13*(4), 323–328.

Dietrich, E. (2007). After the humans are gone. *Journal of Experimental and Theoretical Artificial Intelligence 19*(1), 55–67. http://bingweb.binghamton.edu/$dietrich/Selected_Publications.html; shorter version appears in *Philosophy Now* 61 (May/June): 16-19, https://philosophynow.org/issues/61/After_The_Humans_Are_Gone.

Dijkstra, E. W. (1968, March). Go to statement considered harmful. *Communications of the ACM 11*(3), 147–148. https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD361.html.

Dijkstra, E. W. (1972). Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.), *Structured Programming*, pp. 1–82. London: Academic Press. https://pdfs.semanticscholar.org/013b/f90f472e49c05263b90d9e36f8d2705e7fc7.pdf. See also Dijkstra, Edsger W. (2001), "What Led to 'Notes on Structured Programming' " (EWD1308), https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1308.html.

Dijkstra, E. W. (1974, June-July). Programming as a discipline of mathematical nature. *American Mathematical Monthly 81*(6), 608–612.

Dijkstra, E. W. (1975a). EWD 512: Comments at a symposium. In *Selected Writings on Computing: A Personal Perspective*, pp. 161–164. New York: Springer-Verlag. http://www.cs.utexas.edu/~EWD/ewd05xx/EWD512.PDF.

Dijkstra, E. W. (1975b, August). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*(8), 453–457. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD472.html.

Dijkstra, E. W. (1976). EWD 611: On the fact that the Atlantic Ocean has two sides. In *Selected Writings on Computing: A Personal Perspective*, pp. 268–276. New York: Springer-Verlag. http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD611.html.

Dijkstra, E. W. (1983). Fruits of misunderstanding (EWD-854). http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD854.html. Reprinted in *Datamation* (15 February 1985): 86–87.

Dijkstra, E. W. (1986). Mathematicians and computing scientists: The cultural gap. *Mathematical Intelligencer 8*(1), 48–52. Reprinted in *Abacus: The Magazine for the Computer Professional* 4(4) (1987): 26–31.

Dipert, R. R. (1993). *Artifacts, Art Works, and Agency*. Philadelphia: Temple University Press.

Downes, S. (1990). Herbert Simon's computational models of scientific discovery. *PSA: Proceedings of the [1990] Biennial Meeting of the Philosophy of Science Association 1*, 97–108.

Doxiadis, A., C. H. Papadimitriou, A. Papadatos, and A. Di Donna (2009). *Logicomix: An Epic Search for Truth*. New York: Bloomsbury USA.

Dresner, E. (2003). Effective memory and Turing's model of mind. *Journal of Experimental & Theoretical Artificial Intelligence 15*(1), 113–123.

Dresner, E. (2010). Measurement-theoretic representation and computation-theoretic realization. *Journal of Philosophy 107*(6), 272–292.

Dresner, E. (2012). Turing, Matthews and Millikan: Effective memory, dispositionalism and pushmepullyou states. *International Journal of Philosophical Studies 20*(4), 461–472.

Dretske, F. (1981). *Knowledge and the Flow of Information*. Oxford: Blackwell.

Dretske, F. (1985, September). Machines and the mental. *Proceedings and Addresses of the American Philosophical Association 59*(1), 23–33.

Dreyfus, H. L. (2001). *On the Internet, 2nd Edition*. London: Routledge. http://cryptome.org/2013/01/aaron-swartz/On-the-Internet.pdf.

Dreyfus, S. E. and H. L. Dreyfus (1980). A five-state model of the mental activities involved in directed skill acquisition. Technical Report ORC-80-2, Operations Research Center, University of California, Berkeley. http://www.dtic.mil/dtic/tr/fulltext/u2/a084551.pdf.

Dumas, A. p. (1844). *The Count of Monte Cristo*. London: Penguin Books, 2003.

Duncan, W. D. (2017). Ontological distinctions between hardware and software. *Applied Ontology 12*(17), 5–32. Earlier version (2009) at http://tinyurl.com/yajq4hnw.

Dunn, J. M. (2008). Information in computer science. In P. Adriaans and J. van Benthem (Eds.), *Philosophy of Information*, pp. 587–614. North-Holland. Short pre-print at http://www.illc.uva.nl/HPI/Draft_Information_in_Computer_Science.pdf.

Dunn, J. M. (2013). A guide to the Floridi Keys. *Metascience 22*, 93–98. Essay review of Floridi (2011).

Dunning, B. (16 May 2007). The importance of teaching critical thinking. http://skeptoid.com/episodes/4045.

Dunning, B. (8 May 2018). Are you living in a simulation? https://skeptoid.com/episodes/4622.

Dyson, F. (2004, 13 May). The world on a string. *New York Review of Books*, 16–19.

Dyson, F. (2011a, 10 November). The case for far-out possibilities. *New York Review of Books*, 26–27.

Dyson, F. (2011b, 10 March). How we know. *New York Review of Books*, 8, 10, 12.

Dyson, G. (2012a, 23 February). Turing centenary: The dawn of computing. *Nature 482*(7386), 459–460. doi:10.1038/482459a.

Dyson, G. (2012b). *Turing's Cathedral: The Origins of the Digital Universe*. New York: Pantheon.

Easton, T. A. (2006, May). Beyond the algorithmization of the sciences. *Communications of the ACM 49*(5), 31–33.

Eco, U. (1982). On the impossibility of drawing a map of the empire on a scale of 1 to 1. In *Hot to Travel with a Salmon and Other Essays*, pp. 95–106. New York: Harcourt Brace.

Edelman, S. (2008a). *Computing the Mind*. New York: Oxford University Press.

Edelman, S. (2008b, September). On the nature of minds; or: Truth and consequences. *Journal of Experimental & Theoretical Artificial Intelligence 20*(3), 181–196. http://kybele.psych.cornell.edu/~edelman/Edelman-JETAI.pdf.

Eden, A. H. (2005, 21 September). Software ontology as a cognitive artefact. Talk given to the SUNY Buffalo Center for Cognitive Science; slides at https://www.cse.buffalo.edu//~rapaport/eden2005-SWOntCogArt.pdf.

Eden, A. H. (2007, July). Three paradigms of computer science. *Minds and Machines 17*(2), 135–167. http://www.ic.unicamp.br/~wainer/cursos/2s2006/epistemico/filosofia-cs.pdf.

Eden, A. H., J. H. Moor, J. H. Søraker, and E. Steinhart (Eds.) (2012). *Singularity Hypotheses: A Scientific and Philosophical Assessment*. Berlin: Springer. http://singularityhypothesis.blogspot.co.uk/.

Eden, A. H. and R. Turner (2007). Problems in the ontology of computer programs. *Applied Ontology 2*(1), 13–36. http://www.eden-study.org/articles/2007/problems-ontology-programs_ao.pdf.

Edmonds, D. and N. Warburton (2010). *Philosophy Bites*. Oxford: Oxford University Press.

Edwards, D. A. (2013, April). Platonism is the law of the land. *Notices of the AMS 60*(4), 475–478. https://www.ams.org/notices/201304/rnoti-p475.pdf.

Egan, F. (1991, April). Must psychology be individualistic? *Philosophical Review 100*(2), 179–203.

Egan, F. (1995, April). Computation and content. *Philosophical Review 104*(2), 181–203.

Egan, F. (2010, September). Computational models: A modest role for content. *Studies in History and Philosophy of Science 41*(3), 253–259.

Egan, F. (2012, January-March). Metaphysics and computational cognitive science: Let's not let the tail wag the dog. *Journal of Cognitive Science (South Korea) 13*(1), 39–49. http://cogsci.snu.ac.kr/jcs/issue/vol13/no1/02_Frances+Egan.pdf.

Egan, F. (2014, August). How to think about mental content. *Philosophical Studies 170*(1), 115–135. Preprint at https://www.academia.edu/4160744/How_to_think_about_Mental_Content; video at https://vimeo.com/60800468.

Eilon, S. (1969, December). What is a decision? *Management Science 16*(4, Applicaiton Series), B–172–B–189.

Einstein, A. (1921, 27 January). Geometry and experience. Address to the Prussian Academy of Sciences, Berlin; English version at http://www-history.mcs.st-andrews.ac.uk/Extras/Einstein_geometry.html.

Einstein, A. (1940, 24 May). Considerations concerning the fundaments of theoretical physics. *Science 91*(2369), 487–492.

Ekdahl, B. (1999, Part B). Interactive computing does not supersede Church's thesis. *The Association of Management and the International Association of Management, 17th Annual International Conference, San Diego, CA, August 6–8, Proceedings Computer Science 17*(2), 261–265.

Ellerton, P. (2016, 14 September). What exactly is the scientific method and why do so many people get it wrong? *The Conversation*. https://theconversation.com/what-exactly-is-the-scientific-method-and-why-do-so-many-people-get-it-wrong-65117.

Elser, V. (2012, September-October). In a class by itself. *American Scientist 100*, 418–420. https://www.americanscientist.org/article/in-a-class-by-itself; see also https://www.americanscientist.org/author/veit_elser.

Ensmenger, N. (2003, September-October). Bits of history: Review of A.R. Burks's *who invented the computer? the legal battle that changed computing history*. *American Scientist 91*, 467–468. http://www.americanscientist.org/bookshelf/pub/bits-of-history.

Ensmenger, N. (2011, April). Building castles in the air. *Communications of the ACM 54*(4), 28–30. http://dx.doi.org/10.1145/1924421.1924432.

Evans, J. S. and K. E. Stanovich (2013). Dual-process theories of higher cognition: Advancing the debate. *Perspectives on Psychological Science 8*(3), 223–241. http://www.keithstanovich.com/Site/Research_on_Reasoning_files/Evans_Stanovich_PoPS13.pdf.

Everett, M. (2012, September/October). Answer to 'what's the most important question, and why?'. *Philosophy Now 92*, 38–41.

Eysenck, M. W. (1990). Artificial intelligence. In M. Eysenck (Ed.), *The Blackwell Dictionary of Cognitive Psychology*, pp. 22. Oxford: Basil Blackwell.

Farkas, D. K. (1999, February). The logical and rhetorical construction of procedural discourse. *Technical Communication 46*(1), 42–54. http://www.hcde.washington.edu/sites/default/files/people/docs/proceduraldiscourse.pdf.

Feferman, S. (1992). Turing's 'oracle': From absolute to relative computability—and back. In J. Echeverria, A. Ibarra, and T. Mormann (Eds.), *The Space of Mathematics: Philosophical, Epistemological, and Historical Exporations*, pp. 314–348. Berlin: Walter de Gruyter.

Feferman, S. (2006a, June). Are there absolutely unsolvable problems? gödel's dichotomy. *Philosophia Mathematica 14*(2), 134–152. https://pdfs.semanticscholar.org/246c/95f81bdf8118d0c0354efd19643fe74da0af.pdf.

Feferman, S. (2006b). Turing's [PhD] thesis. *Notices of the AMS 53*(10), 1200–1206.

Feferman, S. (2011). Gödel's incompleteness theorems, free will, and mathematical thought. In R. Swinburne (Ed.), *Free Will and Modern Science*. Oxford University Press/British Academy. https://philarchive.org/archive/FEFGIT.

Feigenbaum, E. A. (2003, January). Some challenges and grand challenges for computational intelligence. *Journal of the ACM 50*(1), 32–40.

Fekete, T. and S. Edelman (2011, September). Towards a computational theory of experience. *Consciousness and Cognition 20*(3), 807–827. http://kybele.psych.cornell.edu/~edelman/Fekete-Edelman-ConCog11-in-press.pdf.

Fellows, M. R. and I. Parberry (1993, January). SIGACT trying to get children excited about CS. *Computing Research News*, 7. https://larc.unt.edu/ian/pubs/crn1993.pdf.

Fetzer, J. H. (1988, September). Program verification: The very idea. *Communications of the ACM 31*(9), 1048–1063.

Fetzer, J. H. (1991). Philosophical aspets of program verification. *Minds and Machines 1*, 197–216. Reprinted in Colburn et al. 1993, pp. 403–427.

Fetzer, J. H. (1993). Program verification. In A. Kent and J. G. Williams (Eds.), *Encyclopedia of Computer Science and Technology, Vol. 28, Supp. 13*, pp. 237–254. New York: Marcel Dekker. Reprinted in Allen Kent & James G. Williams (eds.), *Encyclopedia of Microcomputers*, Vol. 14: "Productivity and Software Maintenance: A Managerial Perspective to Relative Addressing" (New York: Marcel Dekker): 47–64.

Fetzer, J. H. (1996, Summer). Computer reliability and public policy: Limits of knowledge of computer-based systems. *Social Philosophy and Policy 13*(2), 229–266.

Fetzer, J. H. (1998). Computer sytems: The uncertainty of their reliability. *Bridges 5*(3/4), 197–217.

Fetzer, J. H. (1999). The role of models in computer science. *The Monist 82*(1), 20–36.

Feyerabend, P. (1975). *Against Method: Outline of an Anarchistic Theory of Knowledge*. London: Verso.

Figdor, C. (2017). On the proper domain of psychological predicates. *Synthese 194*, 4289–4310.

Findler, N. V. (1993). Heuristic. In A. Ralston and E. D. Reilly (Eds.), *Encyclopedia of Computer Science, 3rd Edition*, pp. 611–612. New York: Van Nostrand Reinhold.

Fine, A. (1986, April). Unnatural attitudes: Realist and instrumentalist attachments to science. *Mind 95*(378), 149–179.

Fischer, P. C. (1970, September). Review of de Leeuw et al. 1956. *Journal of Symbolic Logic 35*(3), 481–482.

Fisher, L. M. (1989, 15 December). Xerox sues Apple Computer over Macintosh copyright. *New York Times*. http://www.nytimes.com/1989/12/15/business/company-news-xerox-sues-apple-computer-over-macintosh-copyright.html.

Fiske, E. B. (1989, 29 March). Between the 'two cultures': Finding a place in the curriculum for the study of technology. *New York Times*, B8. http://www.nytimes.com/1989/03/29/us/education-lessons.html.

Fitch, W. T. (2005). Computation and cognition: Four distinctions and their implications. In A. Cutler (Ed.), *Twenty-First Century Psycholinguistics: Four Cornerstones*, pp. 381–400. Mahway, NJ: Lawrence Erlbaum Associates. http://homepage.univie.ac.at/tecumseh.fitch/wp-content/uploads/2010/08/Fitch2005Computation.pdf.

Fitch, W. T., M. D. Hauser, and N. Chomsky (2005). The evolution of the language faculty: Clarifications and implications. *Cognition 97*, 179–210. http://dash.harvard.edu/bitstream/handle/1/3117935/Hauser_EvolutionLanguageFaculty.pdf.

Fitzsimmons, E. G. (2013, 24 December). Alan Turing, Enigma code-breaker and computer pioneer, wins royal pardon. *New York Times*. http://www.nytimes.com/2013/12/24/world/europe/alan-turing-enigma-code-breaker-and-computer-pioneer-wins-royal-pardon.html.

Flanagan, O. (2012, 13 January). Buddhism without the hocus-pocus. *The Chronicle [of Higher Education] Review 58*(19), B4–B5.

Fletcher, J. (1972). Indicators of humanhood: A tentative profile of man. *Hastings Center Report 2*(5), 1–4.

Fletcher, L. R. (2007). Slow reading: The affirmation of authorial intent. http://www.freelance-academy.org/slowread.htm.

Floridi, L. (2002, January). What is the philosophy of information? *Metaphilosophy 33*(1–2), 123–145.

Floridi, L. (2003). *The Philosophy of Information*. Springer. Two special issues of *Minds and Machines*, 13(4) and 14(1).

Floridi, L. (2004a). *The Blackwell Guide to the Philosophy of Computing and Information*. Malden, MA: Blackwell.

Floridi, L. (2004b, July). Open problems in the philosophy of information. *Metaphilosophy 35*(4), 554–582.

Floridi, L. (2010). *Information: A Very Short Introduction*. Oxford: Oxford University Press.

Floridi, L. (2011). *The Philosophy of Information*. Oxford: Oxford University Press. See also Dunn (2013).

Florman, S. C. (1994). *The Existential Pleasures of Engineering, Second Edition*. New York: St. Martin's Press.

Fodor, J. and E. Lepore (1992). *Holism: A Shopper's Guide*. Cambridge, MA: Basil Blackwell.

Fodor, J. A. (1968). *Psychological Explanation: An Introduction to the Philosophy of Psychology*. New York: Random House.

Fodor, J. A. (1974, October). Special sciences (or: The disunity of science as a working hypothesis). *Synthese 28*(2), 97–115.

Fodor, J. A. (1975). *The Language of Thought*. New York: Thomas Y. Crowell Co.

Fodor, J. A. (1978). Tom Swift and his procedural grandmother. *Cognition 6*, 229–247. http://www.nyu.edu/gsas/dept/philo/courses/mindsandmachines/Papers/tomswift.pdf.

Fodor, J. A. (1980, March). Methodological solipsism considered as a research strategy in cognitive psychology. *Behavioral and Brain Sciences 3*(1), 63–109.

Fodor, J. A. (1981, January). The mind-body problem. *Scientific American 244*(1), 114–123.

Foley, J. (2002, September). Computing $>$ computer science. *Computing Research News 14*(4), 6. revised version at http://archive.cra.org/reports/computing/index.html; response by Robert L. Glass at http://archive.cra.org/reports/computing/glass.html.

Folger, T. (2016, February). The quantum hack. *Scientific American 314*(2), 48–55. http://www.cs.virginia.edu/~robins/The_Quantum_Hack.pdf.

Folina, J. (1998). Church's thesis: Prelude to a proof. *Philosophia Mathematica 6*, 302–323.

Ford, H. (1928, April). My philosophy of industry. *The Forum 79*(4). Interview conducted by Fay Leone Faurote; see also http://quoteinvestigator.com/2016/04/05/so-few/.

Forester, T. and P. Morrison (1994). *Computer Ethics: Cautionary Tales and Ethical Dilemmas in Computing; Second Edition*. Cambridge, MA: MIT Press.

Forster, E. (1909). The machine stops. http://archive.ncsa.illinois.edu/prajlich/forster.html.

Forster, E. (1910). *Howards End*. Mineola, NY: Dover Publications (2002). https://www.gutenberg.org/files/2946/2946-h/2946-h.htm.

Forsythe, G. E. (1967a, January). A university's educational program in computer science. *Communications of the ACM 10*(1), 3–8.

Forsythe, G. E. (1967b, May). What to do till the computer scientist comes. *American Mathematical Monthly 75*(5), 454–462.

Forsythe, G. E. (1968). Computer science and education. *Information Processing 68: Proceedings of IFIP Congress 1968*, 1025–1039.

Fortnow, L. (2006, 14 July). Principles of problem solving: A TCS response. http://blog.computationalcomplexity.org/2006/07/principles-of-problem-solving-tcs.html.

Fortnow, L. (2009, September). The status of the P versus NP problem. *Communications of the ACM 52*(9), 78–86. http://cacm.acm.org/magazines/2009/9/38904-the-status-of-the-p-versus-np-problem/fulltext.

Fortnow, L. (2010, December). What is computation? *Ubiquity 2010*. Article 5, http://ubiquity.acm.org/article.cfm?id=1921573.

Fortnow, L. (2013). *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton, NJ: Princeton University Press.

Fortnow, L. (2018a, 26 January). From art to science. http://blog.computationalcomplexity.org/2018/01/from-art-to-science.html.

Fortnow, L. (2018b, 29 March). A reduced Turing award. https://blog.computationalcomplexity.org/2018/03/a-reduced-turing-award.html.

Fox, M. (2013, 30 April). Janos Starker, master of the cello, dies at 88. *New York Times*, B16. http://www.nytimes.com/2013/04/30/arts/music/janos-starker-master-cellist-dies-at-88.html.

Frailey, D. J. (2010, November). What is computation? computation is process. *Ubiquity 2010*(November). Article 5; http://ubiquity.acm.org/article.cfm?id=1891341.

Frakt, A. (2015, 9 December). Your new medical team: Algorithms and physicians. *New York Times*, A27. http://www.nytimes.com/2015/12/08/upshot/your-new-medical-team-algorithms-and-physicians.html.

Frances, B. (2017, January). Extensive philosophical agreement and progress. *Metaphilosophy 48*(1–2), 47–57.

Franzén, T. (2005). *Gödel's Theorem: An Incomplete Guide to Its Use and Abuse*. Wellesley, MA: A K Peters.

Frazer, J. G. (1911–1915). *The Golden Bough: A Study in Magic and Religion, 3rd ed.* London: Macmillan.

Freeman, P. A. (1995, March). Effective computer science. *ACM Computing Surveys 27*(1), 27–29.

Freeth, T., e. a. (2006, 30 November). Decoding the ancient greek astronomical calculator known as the antikythera mechanism. *Nature 444*, 587–591.

Freeth, T. (2009, December). Decoding an ancient computer. *Scientific American 301*(6), 76–83.

Frege, G. (1892). On sense and reference. In P. Geach and M. Black (Eds.), *Translations from the Philosophical Writings of Gottlob Frege*, pp. 56–78. Oxford: Basil Blackwell, 1970. M. Black, trans.

French, R. M. (2000). The Turing test: The first fifty years. *Trends in Cognitive Sciences 4*(3), 115–121. http://leadserv.u-bourgogne.fr/rfrench/french/TICS_turing.pdf.

Frenkel, E. (2013). *Love and Math: The Heart of Hidden Reality*. New York: Basic Books.

Frenkel, K. (1993, January). An interview with Robin Milner. *Communications of the ACM 36*(1), 90–97. http://delivery.acm.org/10.1145/160000/151241/a1991-frenkel.pdf.

Friedman, B. and J. Kahn, Peter H. (1997). People are responsible, computers are not. In M. D. Ermann, M. B. Williams, and M. S. Shauf (Eds.), *Computers, Ethics, and Society, Second Edition*, pp. 303–314. New York: Oxford University Press. Excerpt from their "Human Agency and Responsible Computing: Implications for Computer System Design", *Journal of Systems and Software* (1992): 7–14.

Frigg, R., S. Hartman, and I. Cyrille (2009). Special issue on models and simulations. *Synthese 169*(3), 425–626. http://link.springer.com/journal/11229/169/3.

Frigg, R. and S. Hartmann (2012). Models in science. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Fall 2012 Edition)*. Metaphysics Research Lab, Stanford University. http://plato.stanford.edu/archives/fall2012/entries/models-science/.

Fry, H. (2019, 1 July). Looks like rain: How weather forecasting got good. *The New Yorker*, 61–65. https://www.newyorker.com/magazine/2019/07/01/why-weather-forecasting-keeps-getting-better.

Gal-Ezer, J. and D. Harel (1998, September). What (else) shoud CS educators know? *Communications of the ACM 41*(9), 77–84.

Galbi, E. W. (1971, April). Software and patents: A status report. *Communications of the ACM 14*(4), 274–280. http://simson.net/ref/2007/cs3610/ref/p274-galbi.pdf.

Gandy, R. (1980). Church's thesis and principles for mechanisms. In J. Barwise, H. Keisler, and K. Kunen (Eds.), *The Kleene Symposium*, pp. 123–148. North-Holland.

Gandy, R. (1988). The confluence of ideas in 1936. In R. Herken (Ed.), *The Universal Turing Machine: A Half-Century Survey, Second Edition*, pp. 51–102. Vienna: Springer-Verlag. https://fi.ort.edu.uy/innovaportal/file/20124/1/41-herken_ed._95_-_the_universal_turing_machine.pdf.

Gazzaniga, M. S. (2010, July). Neuroscience and the correct level of explanation for understanding mind: An extraterrestrial roams through some neuroscience laboratories and concludes earthlings are not grasping how best to understand the mind-brain interface. *Trends in Cognitive Sciences 14*(7), 291–292.

Gelenbe, E. (2011, February). Natural computation. *Ubiquity 2011*(February). Article 1, http://ubiquity.acm.org/article.cfm?id=1940722.

Gemignani, M. (1981, March). What is a computer program? *American Mathematical Monthly 88*(3), 185–188.

George, A. (1983, January/February). Philosophy and the birth of computer science. *Robotics Age*, 26–31.

George Washington University Department of Computer Science (2003). Computer science careers. http://www.seas.gwu.edu/~simhaweb/misc/cscareers.html.

Gettier, E. L. (1963). Is justified true belief knowledge? *Analysis 23*, 121–123. http://www.ditext.com/gettier/gettier.html.

Gigerenzer, G. and D. G. Goldstein (1996). Mind as computer: Birth of a metaphor. *Creativity Research Journal 9*(2–3), 131–144. http://web.ebscohost.com/ehost/pdfviewer/pdfviewer?sid=9fd0749e-892b-4f80-8663-7a539038aa82%40sessionmgr198&vid=1&hid=114.

Gillis, J. (2017, 20 August). Should you trust climate science? maybe the eclipse is a clue. *New York Times*, A14. https://nyti.ms/2v9dLji.

Ginsberg, M. L. (Ed.) (1987). *Readings in Nonmonotonic Reasoning*. Los Altos, CA: Morgan Kaufmann.

Gladwell, M. (2011, 16 May). Creation myth. *New Yorker*, 44, 46, 48–50, 52A–53A. slightly different version at http://www.newyorker.com/reporting/2011/05/16/110516fa_fact_gladwell.

Glanzberg, M. (2016). Truth. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2016 ed.). Stanford University. http://plato.stanford.edu/archives/fall2016/entries/truth/.

Glass, R. L. (2002, August). The proof of correctness wars. *Communications of the ACM 45*(8), 19–21.

Glass, R. L. and L. C. Paulson (2017, September). Computational thinking is not necessarily computational. *Communications of the ACM 60*(9), 8. https://m.cacm.acm.org/magazines/2017/9/220430-computational-thinking-is-not-necessarily-computational/fulltext.

Gleick, J. (2008, 18 December). 'if Shakespeare had been able to Google…'. *New York Review of Books 55*(20), 77–79; highlighted paragraph on p. 78, cols. 2–3.

Gleick, J. (2011). *The Information: A History, a Theory, a Flood*. New York: Pantheon.

Gleick, J. (2014, 24 August). A unified theory. *New York Times Book Review*, 17. http://www.nytimes.com/2014/08/24/books/review/geek-sublime-by-vikram-chandra.html.

Gödel, K. (1964). Postcriptum. In M. Davis (Ed.), *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, pp. 71–73. New York: Raven Press, 1965.

Gödel, K. (perhaps about 1938). Undecidable Diophantine propositions. In S. Feferman et al. (Eds.), *Collected Works, Vol. III*, pp. 164–175. Oxford: Oxford University Press. http://tinyurl.com/ybw84oa6.

Gold, E. M. (1965). Limiting recursion. *Journal of Symbolic Logic 30*(1), 28–48.

Gold, E. M. (1967). Language identification in the limit. *Information and Control 10*, 447–474. https://www.sciencedirect.com/science/article/pii/S0019995867911655.

Goldberger, N. R. (1979). Developmental assumptions underlying models of general education. Paper presented at the Conference on General Education, William Patterson College.

Goldfain, A. (2006). Embodied enumeration: Appealing to activities for mathematical explanation. In M. Beetz, K. Rajan, M. Thielscher, and R. B. Rusu (Eds.), *Cognitive Robotics: Papers from the AAAI Workshop (CogRob2006), Technical Report WS-06-03*, pp. 69–76. Menlo Park, CA: AAAI Press.

Goldfain, A. (2008). A computational theory of early mathematical cognition. PhD dissertation (Buffalo: SUNY Buffalo Department of Computer Science & Engineering), http://www.cse.buffalo.edu/sneps/Bibliography/GoldfainDissFinal.pdf.

Goldin, D. and P. Wegner (2004). The origins of the Turing thesis myth. https://web.archive.org/web/20120212032143/http://www.engr.uconn.edu/~dqg/papers/myth.pdf.

Goldin, D. and P. Wegner (2008, Spring). The interactive nature of computing: Refuting the strong church-turing thesis. *Minds and Machines 18*(1), 17–38.

Goldin, D. Q., S. A. Smolka, P. C. Attie, and E. L. Sonderegger (2004, November). Turing machines, transition systems, and interaction. *Information and Computation 194*(2), 101–128. http://www.cse.uconn.edu/~dqg/papers/its.pdf.

Goldstein, R. N. (2014). *Plato at the Googleplex: Why Philosophy Won't Go Away*. New York: Pantheon Books.

Goldstine, H. H. (1972). *The Computer from Pascal to von Neumann*. Princeton, NJ: Princeton University Press.

Goodman, N. D. (1984, July). The knowing mathematician. *Synthese 60*(1), 21–38.

Goodman, N. D. (1987, October). Intensions, Church's thesis, and the formalization of mathematics. *Notre Dame Journal of Formal Logic 28*(4), 473–489. http://projecteuclid.org/download/pdf_1/euclid.ndjfl/1093637644.

Google, Inc. (2012, 12 April). Google's 4/12/12 copyright liability trial brief, case no. 3:10-CV 03561-WHA, document 897. http://www.wired.com/wp-content/uploads/blogs/wiredenterprise/wp-content/uploads/2012/04/20120412-Googles-Copyright-Stance.pdf.

Gopnik, A. (1996, December). The scientist as child. *Philosophy of Science 63*(4), 485–514. http://courses.media.mit.edu/2002fall/mas962/MAS962/gopnik.pdf.

Gopnik, A. (2009a). *The Philosophical Baby: What Children's Minds Tell Us about Truth, Love, and the Meaning of Life*. New York: Farrar, Straus and Giroux.

Gopnik, A. (2009b, 23 November). What's the recipe? *New Yorker*, 106–112. http://www.newyorker.com/arts/critics/atlarge/2009/11/23/091123crat_atlarge_gopnik.

Gopnik, A. (2013, 11 & 18 February). Moon man: What Galileo saw. *New Yorker*, 103–109. http://www.newyorker.com/magazine/2013/02/11/moon-man.

Gopnik, A. (2015a). The evolution catechism. *The New Yorker (online)*. http://www.newyorker.com/news/daily-comment/evolution-catechism.

Gopnik, A. (2015b, 30 November). Spooked. *New Yorker*, 84–86. http://www.newyorker.com/magazine/2015/11/30/spooked-books-adam-gopnik.

Gordin, M. D. (2012, 21 September). Separating the pseudo from science. *The Chronicle [of Higher Education] Review 59*(4), B10–B12. http://chronicle.com/article/Separating-the-Pseudo-From/134412/.

Gordon, D. M. (2016, February). Collective wisdom of ants. *Scientific American 314*(2), 44–47. https://web.stanford.edu/~dmgordon/articles/other/Gordon%20Scientific%20American.pdf.

Gottlieb, A. (2016). *The Dream of Enlightenment: The Rise of Modern Philosophy*. New YOrk: Liveright (W.W. Norton).

Grabiner, J. V. (1988, October). The centrality of mathematics in the history of western thought. *Mathematics Magazine 61*(4), 220–230. http://www.maa.org/sites/default/files/images/images/upload_library/22/Allendoerfer/1989/0025570x.di021156.02p00042.pdf.

Grady, D. (2019, 20 May). A.I. took a test to detect lung cancer. It got an A. *New York Times*. https://www.nytimes.com/2019/05/20/health/cancer-artificial-intelligence-ct-scans.html.

Green, C. D. (2001). Scientific models, connectionist networks, and cognitive science. *Theory and Psychology 11*, 97–117. http://www.yorku.ca/christo/papers/models-TP2.htm.

Green, C. D. (2005). Was Babbage's analytical engine intended to be a mechanical model of the mind? *History of Psychology 8*(1), 35–45. http://www.yorku.ca/christo/papers/babbage-HoP.pdf.

Green, P. (2014, 17 July). Magic in the mundane. *New York Times*, D1, D4. http://www.nytimes.com/2014/07/17/garden/putting-magic-in-the-mundane.html.

Greengard, S. (2009, December). Making automation work. *Communications of the ACM 52*(12), 18–19.

Grey, D. S. (2016). Language in use: Research on color words. http://www.putlearningfirst.com/language/research/colour_words.html.

Grier, D. A. (2005). *When Computers Were Human*. Princeton, NJ: Princeton University Press. Reviewed in Skinner (2006).

Gries, D. (1981). *The Science of Programming, 3rd printing*. New York: Springer-Verlag, 1985.

Grobart, S. (2011, 27 March). Spoiled by the all-in-one gadget. *New York Times*, WK3. http://www.nytimes.com/2011/03/27/weekinreview/27grobart.html.

Grossmann, R. (1974). *Meinong*. London: Routledge and Kegan Paul.

Grover, L. K. (1999, July/August). Quantum computing. *The Sciences*, 24–30. http://cryptome.org/qc-grover.htm.

Grover, S. and R. Pea (2013). Computational thinking in K–12: A review of the state of the field. *Educsational Researcher 42*(1), 38–43.

Gruber, J. (2007, April). Apple's computer, incorporated. *Macworld*, 112. http://www.macworld.com/article/1056519/aprilspotlight.html.

Grünbaum, A. (1984). *The Foundations of Psychoanalysis*. Berkeley: University of California Press.

Guernsey, L. (2009, 23 January). Computers track the elusive metaphor. *Chronicle of Higher Education*, A11.

Gurevich, Y. (1999, February). The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science 67*, 93–124. http://research.microsoft.com/en-us/um/people/gurevich/Opera/136.pdf.

Gurevich, Y. (2011). What is an algorithm? In *SOFSEM 2012: Theory and Practice of Computer Science*, pp. 31–42. Springer Lecture Notes in Computer Science 7147. http://research.microsoft.com/pubs/155608/209-3.pdf.

Gurevich, Y. (2012, February). Foundational analyses of computation. Technical Report MSR-TR-2012-14, Microsoft Research, Redmond, WA. http://research.microsoft.com/pubs/158617/210.pdf.

Guzdial, M. (2008, August). Paving the way for computational thinking. *Communications of the ACM 51*(8), 25–27. https://www.researchgate.net/publication/234812396_Education_Paving_the_way_for_computational_thinking.

Guzdial, M. (2011, 22 March). A definition of computational thinking from Jeannette Wing. *Computing Education Blog*, https://computinged.wordpress.com/2011/03/22/a-definition-of-computational-thinking-from-jeanette-wing/.

Guzdial, M. and A. Kay (2010, 24 May). The core of computer science: Alan Kay's 'triple whammy'. *Computing Education Blog*. http://computinged.wordpress.com/2010/05/24/the-core-of-computer-science-alan-kays-triple-whammy/.

Haberman, C. (2009, 27 November). The story of a landing. *New York Times Book Review*. http://www.nytimes.com/2009/11/29/books/review/Haberman-t.html.

Habib, S. (2000). Emulation. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, 4th Edition*, pp. 647–648. London: Nature Publishing Group.

Hafner, K. (2002, 19 September). Happy birthday :-) to you: A smiley face turns 20. *New York Times*, G4. http://www.nytimes.com/2002/09/19/technology/typographic-milestones-happy-birthday-to-you-a-smiley-face-turns-20.html.

Hafner, K. (2012, 4 December). Could a computer outthink this doctor? *New York Times*, D1, D6. http://www.nytimes.com/2012/12/04/health/quest-to-eliminate-diagnostic-lapses.html.

Haigh, T. (2013). 'Stored program concept' considered harmful: History and historiography. In P. Bonizzoni, V. Brattka, and B. Löwe (Eds.), *CiE 2013*, pp. 241–251. Berlin: Springer-Verlag Lecture Notes in Computer Science 7921.

Haigh, T. (2014, January). Actually, Turing did not invent the computer. *Communications of the ACM 57*(1), 36–41.

Haigh, T. and M. Priestley (2016, January). Where code comes from: Architectures of automatic control from Babbage to Algol. *Communications of the ACM 59*(1), 39–44. http://www.tomandmaria.com/Tom/Writing/WhereCodeComesFromCACM.pdf.

Hailperin, M., B. Kaiser, and K. Knight (1999). *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. Pacific Grove, CA: Brooks/Cole. https://gustavus.edu/mcs/max/concrete-abstractions.html.

Halmos, P. R. (1973, April). The legend of John von Neumann. *American Mathematical Monthly*, 382–394. http://poncelet.math.nthu.edu.tw/disk5/js/biography/v-n.pdf and http://stepanov.lk.net/mnemo/legende.html.

Halpern, J. Y., R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu (2001, June). On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic 7*(2), 213–236. http://www.lsi.upc.edu/~roberto/EffectivenessOfLogic.pdf.

Halpern, S. (2015, 2 April). How robots & algorithms are taking over. *New York Review of Books 62*(6), 24, 26, 28. http://www.nybooks.com/articles/archives/2015/apr/02/how-robots-algorithms-are-taking-over/.

Halpern, S. (2016, 24 November). Our driverless future. *New York Review of Books 63*(18), 18–20. http://www.nybooks.com/articles/2016/11/24/driverless-intelligent-cars-road-ahead/.

Hamming, R. (1968, January). One man's view of computer science. *Journal of the Association for Computing Machinery 16*(1), 3–12.

Hamming, R. (1980a, February). The unreasonable effectiveness of mathematics. *American Mathematical Monthly 87*(2). http://www.dartmouth.edu/~matc/MathDrama/reading/Hamming.html.

Hamming, R. (1980b). We would know what they thought when they did it. In N. Metropolis, J. Howlett, and G.-C. Rota (Eds.), *A History of Computing in the Twentieth Century: A Collection of Essays*, pp. 3–9. New York: Academic Press.

Hamming, R. (1998, August-September). Mathematics on a distant planet. *American Mathematical Monthly 105*(7), 640–650.

Hammond, T. A. (6 May 2003). What is computer science? Myths vs. truths. http://web.archive.org/web/20030506091438/http://www.columbia.edu/~tah10/cs1001/whatcs.html.

Hansson, S. O. (2015). Science and pseudo-science. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2015 ed.). Stanford University. http://plato.stanford.edu/archives/spr2015/entries/pseudo-science/.

Harel, D. (1980, July). On folk theorems. *Communications of the ACM 23*(7), 379–389. http://www.univasf.edu.br/~marcus.ramos/pc-2008-2/p379-harel.pdf.

Harman, G. (1965, January). The inference to the best explanation. *The Philosophical Review 74*(1), 88–95.

Harnish, R. M. (2002). Coda: Computation for cognitive science, or what IS a computer, anyway? In *Minds, Brains, Computers: An Historical Introduction to the Foundations of Cognitive Science*, pp. 394–412. Malden, MA: Blackwell. Portions may be online at https://www.amazon.com/Minds-Brains-Computers-Introduction-Foundations/dp/0631212604/.

Hartmanis, J. (1993). Some observations about the nature of computer science. In R. Shyamasundar (Ed.), *Foundations of Software Technology and Theoretical Computer Science*, Volume 761 of *Lecture Notes in Computer Science*, pp. 1–12. Springer Berlin/Heidelberg. https://www.researchgate.net/publication/221583809_Some_Observations_About_the_Nature_of_Computer_Science.

Hartmanis, J. (1995a, March). On computational complexity and the nature of computer science. *ACM Computing Surveys 27*(1), 7–16. Reprinted, with added commentaries, from *Communications of the ACM* 37(10) (October 1994): 37–43.

Hartmanis, J. (1995b, March). Response to the essays 'On computational complexity and the nature of computer science'. *ACM Computing Surveys 27*(1), 59–61.

Hartmanis, J. and H. Lin (1992). What is computer science and engineering? In J. Hartmanis and H. Lin (Eds.), *Computing the Future: A Broader Agenda for Computer Science and Engineering*, pp. 163–216. Washington, DC: National Academy Press. Ch. 6.

Hartmanis, J. and R. Stearns (1965, May). On the computational complexity of algorithms. *Transactions of the American Mathematical Society 117*, 285–306. https://fi.ort.edu.uy/innovaportal/file/20124/1/60-hartmanis_stearns_complexity_of_algorithms.pdf.

Hartmanis, J. and R. Stearns (1967, May). Sets of numbers defined by finite automata. *American Mathematical Monthly*, 539–542.

Hartree, D. (1949). *Calculating Instruments and Machines*. Urbana, IL: University of Illinois Press. https://archive.org/details/calculatinginstr00doug.

Haugeland, J. (1981a, Spring). Analog and analog. *Philosophical Topics 12*(1), 213–225.

Haugeland, J. (1981b). Semantic engines: An introduction to mind design. In J. Haugeland (Ed.), *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pp. 1–34. Cambridge, MA: MIT Press.

Hauser, L. (2001). Chinese room argument. *Internet Encyclopedia of Philosophy*. https://www.iep.utm.edu/chineser/.

Hauser, M. D., N. Chomsky, and W. T. Fitch (2002, 22 November). The faculty of language: What is it, who has it, and how did it evolve. *Science 298*, 1569–1579. http://www.chomsky.info/articles/20021122.pdf.

Hauser, S. (2017, January-February). Computing and connecting. *Rochester Review 79*(3), 16–17. https://www.rochester.edu/pr/Review/V79N3/0306_hopper.html.

Hayes, B. (1994, September-October). The World Wide Web. *American Scientist 82*, 416–420. http://bit-player.org/wp-content/extras/bph-publications/AmSci-1994-09-Hayes-www.pdf.

Hayes, B. (1995, July-August). The square root of NOT. *American Scientist 83*, 304–308. http://bit-player.org/wp-content/extras/bph-publications/AmSci-1995-07-Hayes-quantum.pdf.

Hayes, B. (2000, May-June). The nerds have won. *American Scientist 88*, 200–204. http://bit-player.org/wp-content/extras/bph-publications/AmSci-2000-05-Hayes-Nerds.pdf.

Hayes, B. (2002, July-August). The search for rigor. *American Scientist 90*, 382–384.

Hayes, B. (2004, March-April). Small-town story. *American Scientist*, 115–119. http://bit-player.org/wp-content/extras/bph-publications/AmSci-2004-03-Hayes-smalltowns.pdf.

Hayes, B. (2007a, May-June). Calculating the weather. *American Scientist 95*(3). https://web.archive.org/web/20150228152542/http://www.americanscientist.org/bookshelf/pub/calculating-the-weather.

Hayes, B. (2007b, March-April). Trains of thought. *American Scientist 95*(2). http://bit-player.org/wp-content/extras/bph-publications/AmSci-2007-03-Hayes-trains.pdf.

Hayes, B. (2014a, September-October). Pencil, paper, and pi. *American Scientist 102*(1), 342–345. https://www.americanscientist.org/article/pencil-paper-and-pi.

Hayes, B. (2014b, January-February). Programming your quantum computer. *American Scientist 102*(1), 22–25. https://www.americanscientist.org/article/programming-your-quantum-computer.

Hayes, B. (2015, March-April). The 100-billion-body problem. *American Scientist 103*(2), 90–93. https://www.americanscientist.org/article/the-100-billion-body-problem.

Hayes, P. J. (1983, 18–20 May). Proceedings of the fifth annual conference of the cognitive science society. https://cognitivesciencesociety.org/wp-content/uploads/2019/01/cogsci_5.pdf.

Hayes, P. J. (1997, July). What is a computer? *Monist 80*(3), 389–404. http://philo.at/mii/mii/node30.html.

Hearst, M. and H. Hirsh (2000, January/February). AI's greatest trends and controversies. *IEEE Intelligent Systems 15*(1), 8–17. http://www.cs.cornell.edu/courses/cs472/2002fa/handouts/challenges-ai.pdf.

Hedger, L. (1998, April). Analog computation: Everything old is new again. *Indiana University Research & Creative Activity 21*(2). http://www.indiana.edu/~rcapub/v21n2/p24.html.

Heingartner, D. (2006, 18 July). Maybe we should leave that up to the computer. *New York Times*. http://www.nytimes.com/2006/07/18/technology/18model.html.

Heller, N. (2019, 29 July). Driven. *The New Yorker*, 24–29. Online as "Was the Automotive Era a Terrible Mistake?", https://www.newyorker.com/magazine/2019/07/29/was-the-automotive-era-a-terrible-mistake.

Hempel, C. G. (1942, 15 January). The function of general laws in history. *Journal of Philosophy 39*(2), 35–48.

Hempel, C. G. (1962). Deductive-nomological vs. statistical explanation. In H. Feigl and G. Maxwell (Eds.), *Minnesota Studies in the Philosophy of Science, Vol. 3: Scientific Explanationa, Space, and Time*, pp. 98–169. Minneapolis: University of Minnesota Press. http://mcps.umn.edu/philosophy/completeVol3.html.

Hendler, J., N. Shadbolt, W. Hall, T. Berners-Lee, and D. Weitzner (2008, July). Web science: An interdisciplinary approach to understanding the Web. *Communications of the ACM 51*(7), 60–69. http://cacm.acm.org/magazines/2008/7/5366-web-science/fulltext.

Heng, K. (2014, May-June). The nature of scientific proof in the age of simulations. *American Scientist 102*(3), 174–177. https://www.americanscientist.org/article/the-nature-of-scientific-proof-in-the-age-of-simulations.

Henkin, L. (1962, 16 November). Are logic and mathematics identical? *Science 138*(3542), 788–794.

Henzinger, T. (1996, December). Some myths about formal verification. *ACM Computing Surveys 28*(4es). Article No. 119.

Herman, G. (1983). Algorithms, theory of. In A. S. Ralston and E. D. Riley (Eds.), *Encyclopedia of Computer Science, 3rd edition*, pp. 37–39. New York: Van Nostrand Reinhold.

Hidalgo, C. (2015, August). Planet hard drive. *Scientific American 313*(2), 72–75.

Higginbotham, A. (2014, 9 November). The disillusionist [the unbelievable skepticism of the Amazing Randi]. *New York Times Magazine*, 48–53, 60–61. http://www.nytimes.com/2014/11/09/magazine/the-unbelievable-skepticism-of-the-amazing-randi.html.

Hilbert, D. (1899). *Foundations of Geometry*. La Salle, IL: Open Court. 2nd edition trans. by Leo Unger; 10th edition revised and enlarged by Paul Bernays; different edition online at https://math.berkeley.edu/~wodzicki/160/Hilbert.pdf.

Hilbert, D. (1900, July). Mathematical problems: Lecture delivered before the International Congress of Mathematicians at Paris in 1900. *Bulletin of the American Mathematical Society 8*(10), 437–479. Trans. by Mary Winston Newson; first published in *Göttinger Nachrichten* (1900): 253–297.

Hilbert, D. and W. Ackermann (1928). *Principles of Mathematical Logic*. New York (1950): Chelsea. Robert E. Luce (ed.); Lewis M. Hammon, George G. Leckie, & F. Steinhardt (trans.); based on *Grundzüge der Theoretischen Logik*, 2nd edition, 1938.

Hilbert, M. and P. López (2011, 1 April). The world's technological capacity to store, communicate, and compute information. *Science 332*, 60–65.

Hill, D. J. (2016a, 11 May). AI teaching assistant helped students online—and no one knew the difference. *SingularityHUB*. https://singularityhub.com/2016/05/11/ai-teaching-assistant-helped-students-online-and-no-one-knew-the-difference/.

Hill, R. (25 September 2008). Empire, regime, and perspective change in our creative activities. Conference on ReVisioning the (W)hole II: Curious Intersections, http://web.archive.org/web/20080925064006/http://www.newhumanities.org/events/revisioning.html.

Hill, R. K. (2013, June). What an algorithm is, and is not. *Communications of the ACM 56*(6), 8–9. http://mags.acm.org/communications/june_2013/?pg=11.

Hill, R. K. (2016b). What an algorithm is. *Philosophy and Technology 29*, 35–59.

Hill, R. K. (2017a, 26 February). Fact versus frivolity in Facebook. *BLOG@CACM*, http://cacm.acm.org/blogs/blog-cacm/214075-fact-versus-frivolity-in-facebook/fulltext.

Hill, R. K. (2017b, March). What makes a program elegant? *Communications of the ACM 60*(3), 13. http://cacm.acm.org/blogs/blog-cacm/208547-what-makes-a-program-elegant/fulltext.

Hill, R. K. (2018, 21 May). Articulation of decision responsibility. *BLOG@CACM*. https://cacm.acm.org/blogs/blog-cacm/227966-articulation-of-decision-responsibility/fulltext.

Hill, R. K. and W. J. Rapaport (2018, Fall). Exploring the territory: The logicist way and other paths into the philosophy of computer science. *American Philosophical Association Newsletter on Philosophy and Computers 18*(1), 34–37. https://cdn.ymaws.com/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/ComputersV18n1.pdf.

Hillis, W. D. (1998). *The Pattern on the Stone: The Simple Ideas that Make Computers Work*. New York: Basic Books.

Hills, D. (November 2007). professional website. http://www.stanford.edu/~dhills/cv.html.

Hilpinen, R. (2011). Artifact. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab. http://plato.stanford.edu/entries/artifact/.

Hinchey, M., M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria (2008, September). Software engineering and formal methods. *Communications of the ACM 51*(9), 54–59.

Hintikka, J. and A. Mutanen (1997). An alternative concept of computability. In J. Hintikka (Ed.), *Language, Truth, and Logic in Mathematics*, pp. 174–188. Dordrecht, The Netherlands: Springer.

Hirst, G. (1991). Existence assumptions in knowledge representation. *Artificial Intelligence 49*, 199–242.

Hoare, C. A. R. (1969, October). An axiomatic basis for computer programming. *Communications of the ACM 12*(10), 576–580, 583.

Hoare, C. A. R. (1986, August). Mathematics of programming. *Byte*, 115ff. Reprinted in Timothy R. Colburn, James H. Fetzer, & Terry L. Rankin (eds.), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers, 1993): 135–154.

Hoare, C. A. R. (2009, October). Retrospective: An axiomatic basis for computer programming. *Communications of the ACM 52*(10), 30–32.

Hodges, A. (2006, November). Review of Copeland (2004b). *Notices of the AMS 53*(10), 1190–1199. http://www.ams.org/notices/200610/rev-hodges.pdf.

Hodges, A. (2012a). *Alan Turing: The Enigma; Centenary Edition*. Princeton, NJ: Princeton University Press.

Hodges, A. (2012b, 13 April). Beyond Turing's machines. *Science 336*, 163–164.

Hodges, A. (2013). Church's review of computable numbers. In S. B. Cooper and J. van Leeuwen (Eds.), *Alan Turing: His Work and Impact*, pp. 117–118. Amsterdam: Elsevier.

Hodges, W. (2014). Tarski's truth definitions. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2014 ed.). Stanford University. http://plato.stanford.edu/archives/fall2014/entries/tarski-truth/.

Hoffman, D. D. (2009). The interface theory of perception: Natural selection drives true perception to swift extinction. In S. Dickinson, M. Tarr, A. Leonardis, and B. Schiele (Eds.), *Object Categorization: Computer and Human Vision Perspectives*, pp. 148–165. Cambridge, UK: Cambridge University Press. http://cogsci.uci.edu/~ddhoff/interface.pdf.

Hofstadter, D. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.

Hofstadter, D. (2007). *I Am a Strange Loop*. New York: Basic Books.

Hofstadter, D. and E. Sander (2013). *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking*. New York: Basic Books.

Hofstadter, D. R. (1981, May). Metamagical themas: A coffeehouse conversation on the Turing test to determine if a machine can think. *Scientific American*, 15–36. https://cs.brynmawr.edu/~dblank/csem/coffeehouse.html. Reprinted as "The Turing Test: A Coffeehouse Conversation", in Douglas R. Hofstadter & Daniel C. Dennett (eds.), *The Mind's I: Fantasies and Reflections on Self and Soul* (New York: Basic Books, 1981): 69–95.

Hofstadter, D. R. (1983, 13 November). Mind, body and machine. *New York Times Book Review*. https://cse.buffalo.edu/~rapaport/hofstadter1983-MindBodyMachine-nytbr.pdf.

Hogarth, M. (1992, April). Does general relativity allow an observer to view an eternity in a finite time? *Foundations of Physics Letters 5*(2), 173–181. http://research.cs.queensu.ca/~akl/cisc879/papers/SELECTED_PAPERS_FROM_VARIOUS_SOURCES/Hogarth.pdf.

Hollan, J., E. Hutchins, and D. Kirsh (2000, June). Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction 7*(2), 174–196. https://www.lri.fr/~mbl/Stanford/CS477/papers/DistributedCognition-TOCHI.pdf.

Holst, P. A. (2000). Analog computer. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, 4th Edition*, pp. 53–59. New York: Grove's Dictionaries.

Holt, C. M. (1989, April). More on the very idea (letter to the editor). *Communications of the ACM 32*(4), 508–509.

Holt, J. (2001, 5 March). The ada perplex. *The New Yorker*, 88–93.

Holt, J. (2009, 15 February). Death: Bad? *New York Times Book Review*, BR27. http://www.nytimes.com/2009/02/15/books/review/Holt-t.html.

Holt, J. (2012, 7 June). How the computers exploded. *New York Review of Books*, 32–34.

Holt, J. (2016, 10 November). Something faster than light? What is it? *New York Review of Books 63*(17), 50–52.

Homer, S. and A. L. Selman (2011). *Computability and Complexity Theory, 2nd Edition*. New York: Springer.

Hopcroft, J. E. and J. D. Ullman (1969). *Formal Languages and Their Relation to Automata*. Reading, MA: Addison-Wesley.

Hopper, G. M. (1952). The education of a computer. In *ACM '52 Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, pp. 243–249. Pittsburgh: ACM.

Hopper, G. M. (1981, July). The first bug. *Annals of the History of Computing 3*(3), 285–286. http://ieeexplore.ieee.org/stamp/stamp.jsp?reload=true&tp=&arnumber=4640691.

Horgan, J. (1990, January). Profile: Claude E. Shannon; unicyclist, juggler and father of information theory. *Scientific American*, 22, 22A–22B.

Horst, S. (2014, 14 December). The computational theory of mind: Alan Turing & the Cartesian challenge. https://web.archive.org/web/20150114211656/http://www.thecritique.com/articles/the-computational-theory-of-mind-alan-turing-the-cartesian-challenge/.

Horsten, L. (2015). Philosophy of mathematics. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/entries/philosophy-mathematics/#StrNom.

Howard, D. A. (2017). Einstein's philosophy of science. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Fall 2017 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/fall2017/entries/einstein-philscience/.

Hoyningen-Huene, P. (2015). A note on the concept of game (or rather *spiel*. In G. Betz, D. Koppelberg, Löwenstein, and A. Wehofsits (Eds.), *Weiter denken: über Philosophie, Wissenschaft und Relgion*, pp. 205–210. Berlin: de Gruyter. http://tinyurl.com/yd54qf3v.

Huber, H. G. (1966, September). Algorithm and formula. *Communications of the ACM 9*(9), 653–654.

Hudelson, R. (1980, April). Popper's critique of Marx. *Philosophical Studies 37*(3), 259–270.

Huemer, W. (2010). Franz Brentano. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab. http://plato.stanford.edu/entries/brentano/.

Hugo, V. (1862). *Les Misérables*. New York: Signet Classics, 1987. Trans. by Lee Fahnestock & Norman MacAfee.

Humphreys, P. (1990). Computer simulations. *PSA: Proceedings of the [1990] Biennial Meeting of the Philosophy of Science Association 2*, 497–506.

Humphreys, P. (2002, September). Computational models. *Philosophy of Science 69*, S1–S11. http://ist.uap.asia/~jpira/compthink/17232634.pdf.

Husbands, P., M. Wheeler, and H. Owen (2008). Introduction: The mechanical mind. In P. Husbands, M. Wheeler, and H. Owen (Eds.), *The Mechanical Mind in History*, pp. 1–17. Cambridge, MA: MIT Press.

Hutchins, E. (1995a). *Cognition in the Wild*. Cambridge, MA: MIT Press.

Hutchins, E. (1995b). How a cockpit remembers its speeds. *Cognitive Science 19*, 265–288.

Huxley, A. (1932). *Brave New World*. Huxley.net. http://www.huxley.net/bnw/.

Hyman, A. (1982). *Charles Babbage: Pioneer of the Computer*. Princeton, NJ: Princeton University Press. Reviewed in O'Hanlon (1982).

Hyman, P. (2012, July). Lost and found. *Communications of the ACM 55*(7), 21.

Irmak, N. (2012). Software is an abstract artifact. *Grazer Philosophische Studien 86*, 55–72. http://philpapers.org/archive/IRMSIA.pdf.

Israel, D. (2002, May). Reflections on Gödel's and Gandy's reflections on Turing's thesis. *Minds and Machines 12*(2), 181–201.

Nature Editors (2016, 28 January). Digital intuition. *Nature 529*, 437. http://www.nature.com/news/digital-intuition-1.19230.

Jackendoff, R. (2012). *A User's Guide to Thought and Meaning*. Oxford: Oxford University Press.

Jackendoff, R. and J. Audring (2018). Morphology and memory: Toward an integrated theory. *Topics in Cognitive Science*. https://onlinelibrary.wiley.com/doi/epdf/10.1111/tops.12334.

Jackendoff, R. and S. Pinker (2005). The nature of the language faculty and its implications for evolution of language. *Cognition 97*, 211–225. http://pinker.wjh.harvard.edu/articles/papers/2005_09_Jackendoff_Pinker.pdf.

Jackman, H. (2017). Meaning holism. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2017 ed.). Metaphysics Research Lab, Stanford University.

Jackson, A. S. (1960). *Analog Computation*. New York: McGraw-Hill.

Jackson, M. (2003). Why software writing is difficult and will remain so. *Information Processing Letters 88*, 13–25. http://users.mct.open.ac.uk/mj665/turski07.pdf.

Jacob, P. (2019). Intentionality. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy*. Mataphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2019/entries/intentionality/.

James, W. (1892). Psychology: Briefer course. In G. E. Myers (Ed.), *William James: Writings 1878–1899*, pp. 1–443. New York: Library of America. Original edition at https://archive.org/details/psychologybriefe00willuoft.

James, W. (1897). The will to believe. In G. E. Myers (Ed.), *William James: Writings 1878–1899*, pp. 457–479. New York: Library of America. http://www.gutenberg.org/files/26659/26659-h/26659-h.htm.

Jenny, M. (2018). Counterpossibles in science: The case of relative computability. *Noûs 52*(3), 530–560.

Johnson, D. G. (2001a). *Computer Ethics*. Upper Saddle River, NJ: Prentice Hall.

Johnson, D. G. and J. W. Snapper (Eds.) (1985). *Ethical Issues in the Use of Computers*. Wadsworth.

Johnson, D. G. and M. Verdicchio (2017, Winter). Reframing AI discourse. *Minds and Machines 27*(4), 575–590. https://www.researchgate.net/profile/Dennis_Mazur/project/Consent-and-Informed-Consent/attachment/5a1c9e7f4cde267c3e6f2148/AS:565449286000640@1511825023010/download/Reframing+AI+Discourse-1111.pdf?context=ProjectUpdatesLog.

Johnson, G. (2001b, 25 March). All science is computer science. *New York Times*. http://www.nytimes.com/2001/03/25/weekinreview/the-world-in-silica-fertilization-all-science-is-computer-science.html.

Johnson, G. (2001c, 27 Februry). Claude Shannon, mathematician, dies at 84. *New York Times*, B7. http://www.nytimes.com/2001/02/27/nyregion/27SHAN.html.

Johnson, G. (2002a, 14 July). To err is human. *New York times*. http://www.nytimes.com/2002/07/14/weekinreview/deus-ex-machina-to-err-is-human.html.

Johnson, K. (2004, October). Gold's theorem and cognitive science. *Philosophy of Science 71*, 571–592. http://www.lps.uci.edu/~johnsonk/Publications/Johnson.GoldsTheorem.pdf.

Johnson, M. (2002b). Review of Davis (2012). *MAA Reviews*. http://mathdl.maa.org/mathDL/19/?pa=reviews&sa=viewBook&bookId=68938.

Johnson-Laird, P. N. (1981). Mental models in cognitive science. In D. A. Norman (Ed.), *Perspectives on Cognitive Science*, Chapter 7, pp. 147–191. Norwood, NJ: Ablex.

Johnson-Laird, P. N. (1988). *The Computer and the Mind: An Introduction to Cognitive Science*. Cambridge, MA: Harvard University Press. Ch. 3 ("Computability and Mental Processes"), pp. 37–53.

Joyce, D. (2005). The Dedekind/Peano axioms. http://aleph0.clarku.edu/~djoyce/numbers/peano.pdf.

Jurafsky, D. and J. H. Martin (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, NJ: Prentice Hall.

Kahn, A. (2014, 3 November). Rover disguised as penguin chick does research better than scientists. *Los Angeles Times Science Now*. http://www.latimes.com/science/sciencenow/la-sci-sn-penguin-robot-chick-study-20141103-story.html.

Kahneman, D. (2011). *Thinking, Fast and Slow*. New York: Farrar, Strauss and Giroux.

Kaiser, D. (2012, 10 February). Paradoxical roots of 'social construction'. *Science 335*, 658–659.

Kalmár, L. (1959). An argument against the plausibility of Church's thesis. In A. Heiting (Ed.), *Constructivity in Mathematics*, pp. 72–80. Amsterdam: North-Holland.

Kanat-Alexander, M. (2008, 10 October). What is a computer? *Code Simplicity*. http://www.codesimplicity.com/post/what-is-a-computer/.

Kant, I. (1781). *Critique of Pure Reason*. New York: St. Martin's Press, 1929. Trans. by Norman Kemp Smith.

Kasparov, G. (2018, 7 December). Chess, a Drosophila of reasoning. *Science 362*(6419), 1087. http://science.sciencemag.org/content/362/6419/1087.

Katz, J. J. (1978). Effability and translation. In F. Guenthner and M. Guenthner-Reutter (Eds.), *Meaning and Translation: Philosophical and Linguistic Approaches*, pp. 191–234. London: Duckworth.

Kay, K. (2001). Machines and the mind. *The Harvard Brain*, Vol. 8 (Spring), http://www.hcs.harvard.edu/~hsmbb/BRAIN/vol8-spring2001/ai.htm.

Kay, M. (2010). Introduction. In D. Flickinger and S. Oepen (Eds.), *Collected Papers of Martin Kay: A Half Century of Computational Linguistics*, pp. 1–18. Stanford, CA: CSLI Studies in Computational Linguistics.

Kearns, J. (1997). Thinking machines: Some fundamental confusions. *Minds and Machines 7*(2), 269–287.

Keats, J. (2009, September). The mechanical loom. *Scientific American*, 88. In "The Start of Everything", http://www.readcube.com/articles/10.1038/scientificamerican0909-88.

Keith, T. (2014, 5 July). The letter that kicked off a radio career. *NPR.org*, http://www.npr.org/2014/07/05/328512614/the-letter-that-kicked-off-a-radio-career.

Kemeny, J. G. (1959). *A Philosopher Looks at Science*. Princeton, NJ: D. van Nostrand.

Kennedy, H. C. (1968, November). Giuseppe Peano at the University of Turin. *Mathematics Teacher*, 703–706. Reprinted in Kennedy, Hubert C. (2002), *Twelve Articles on Giuseppe Peano* (San Francisco: Peremptory Publications): 14–19, http://hubertkennedy.angelfire.com/TwelveArticles.pdf.

Kennedy, Jr., T. (1946, 15 February). Electronic computer flashes answers, may speed engineering. *New York Times*, 1, 16. https://timesmachine.nytimes.com/timesmachine/1946/02/15/93052340.pdf.

Kernan, M. (1990, May). The object at hand. *Smithsonian 21*, 22, 24, 26.

Kfoury, A., R. N. Moll, and M. A. Arbib (1982). *A Programming Approach to Computability*. New York: Springer-Verlag.

Khalil, H. and L. S. Levy (1978, June). The academic image of computer science. *ACM SIGCSE Bulletin 10*(2), 31–33.

Kidder, T. (1985, 29 December). Less (and more) than meets the eye'. *New York Times Book Review*, 6–7. http://www.nytimes.com/1985/12/29/books/less-and-more-than-meets-the-eye.html. Review of Stein (1985).

Kim, E. E. and B. A. Toole (1999, May). Ada and the first computer. *Scientific American*, 76–81. http://people.cs.kuleuven.be/~danny.deschreye/AdaTheFirst.pdf.

King, J. C. (2016). Structured propositions. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2016 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2016/entries/propositions-structured/.

Kirk, R. (1974, January). Sentience and behaviour. *Mind 83*(329), 43–60.

Kleene, S. C. (1935-1936a). General recursive functions of natural numbers. *Mathematische Annalen 112*, 727–742.

Kleene, S. C. (1936b). λ-definability and recursiveness. *Duke Mathematical Journal 2*, 340–353.

Kleene, S. C. (1952). *Introduction to Metamathematics*. Princeton, NJ: D. Van Nostrand.

Kleene, S. C. (1967). *Mathematical Logic*. New York: Wiley.

Kleene, S. C. (1981, January). Origins of recursive function theory. *Annals of the History of Computing 3*(1), 52–67.

Kleene, S. C. (1987, October). Reflections on Church's thesis. *Notre Dame Journal of Formal Logic 28*(4), 490–498. http://projecteuclid.org/download/pdf_1/euclid.ndjfl/1093637645.

Kleene, S. C. (1995). Turing's analysis of computability, and major applications of it. In R. Herken (Ed.), *The Universal Turing Machine: A Half-Century Survey, Second Edition*, pp. 15–49. Vienna: Springer-Verlag. https://fi.ort.edu.uy/innovaportal/file/20124/1/41-herken_ed._95_-_the_universal_turing_machine.pdf.

Klemens, B. (2006). *Math You Can't Use: Patents, Copyright, and Software*. Brookings Institution Press. Excerpted in Klemens, Ben (2006), "Private Numbers", *Chronicle [of Higher Education] Review* 52(22) (3 February): B2.

Kling, R., P. Wegner, J. R. Rice, and E. A. Weiss (1993, February). Broadening computer science. *Communications of the ACM 36*(2), 15–19.

Knight, W. (2017, 11 April). The dark secret at the heart of AI. *MIT Technology Review*. https://www.technologyreview.com/s/604087/the-dark-secret-at-the-heart-of-ai/.

Knobe, J. (2008/2009, December/January). Can a robot, an insect, or God be aware? *Scientific American Mind 19*(6), 68–71.

Knobe, J. (2015, 15 June). Do corporations have minds? *New York Times Opinionator*. http://opinionator.blogs.nytimes.com/2015/06/15/do-corporations-have-minds/.

Knuth, D. E. (1966, September). Algorithm and program: Information and data. *Communications of the ACM 9*(9), 654.

Knuth, D. E. (1972a, July). Ancient Babylonian algorithms. *Communications of the ACM 15*(7), 671–677.

Knuth, D. E. (1972b, August). George Forsythe and the development of computer science. *Communications of the ACM 15*(8), 721–727.

Knuth, D. E. (1973). *The Art of Computer Programming, Second Edition*. Reading, MA: Addison-Wesley.

Knuth, D. E. (1974a, December). Computer programming as an art. *Communications of the ACM 17*(12), 667–673.

Knuth, D. E. (1974b, April). Computer science and its relation to mathematics. *American Mathematical Monthly 81*(4), 323–343.

Knuth, D. E. (1984). Literate programming. *The Computer Journal 27*(2), 97–111. http://comjnl.oxfordjournals.org/content/27/2/97.full.pdf+html. Reprinted in Knuth 1992. Also at http://www.literateprogramming.com/knuthweb.pdf.

Knuth, D. E. (1985, March). Algorithmic thinking and mathematical thinking. *American Mathematical Monthly 92*(3), 170–181.

Knuth, D. E. (1992). *Literate Programming*. Stanford, CA: Center for the Study of Language and Information, CSLI Lecture Notes 27. https://www-cs-faculty.stanford.edu/~knuth/lp.html.

Knuth, D. E. (2001). *Things a Computer Scientist Rarely Talks About*. Stanford, CA: CSLI Publications. CSLI Lecture Notes Number 136.

Knuth, D. E. (2014, 20 May). Twenty questions for Donald Knuth. http://www.informit.com/articles/article.aspx?p=2213858&WT.mc_id=Author_Knuth_20Questions.

Koellner, P. (2018, July). On the question of whether the mind can be mechanized, I: From Gödel to Penrose. *Journal of Philosophy 115*(7), 337–360.

Koen, B. V. (1988). Toward a definition of the engineering method. *European Journal of Engineering Education 13*(3), 307–315. Reprinted from *Engineering Education* (December 1984): 150–155, http://dx.doi.org/10.1080/03043798808939429.

Koen, B. V. (2009, July). The engineering method and its implications for scientific, philosphical, and universal methods. *The Monist 92*(3), 357–386.

Koepsell, D. R. (1998). Review of Smith 1996. *Philosophical Psychology 11*(3), 389–391.

Koepsell, D. R. (2000). *The Ontology of Cyberspace: Philosophy, Law, and the Future of Intellectual Property*. Chicago: Open Court.

Kolata, G. (2004, 21 March). New studies question value of opening arteries. *New York Times*, A1, A21. http://www.nytimes.com/2004/03/21/us/new-heart-studies-question-the-value-of-opening-arteries.html.

Korf, R. E. (1992). Heuristics. In S. C. Shapiro (Ed.), *Encycolpedia of Artificial Intelligence, 2nd edition*, pp. 611–615. New York: John Wiley & Sons.

Korfhage, R. R. (1993). Algorithm. In A. Ralston and E. D. Reilly (Eds.), *Encyclopedia of Computer Science, 3rd Edition*, pp. 27–29. New York: Van Nostrand Reinhold.

Kornblith, H. (2013, November). Naturalism vs. the first-person perspective. *Proceedings & Addresses of the American Philosophical Association 87*, 122–141. http://www.apaonline.org/global_engine/download.asp?fileid=A2DBB747-4555-43B3-9514-28CBB4F6EEB6.

Korsmeyer, C. (2012, October). Touch and the experience of the genuine. *British Journal of Aesthetics 52*(4), 365–377.

Kosslyn, S. M. (2005). Mental images and the brain. *Cognitive Neuropsychology 22*(3/4), 333–347. http://neurosci.info/courses/systems/FMRI/kosslyn05.pdf.

Kramer, J. (2007, April). Is abstracton the key to computing? *Communications of the ACM 50*(4), 36–42. http://www.ics.uci.edu/~andre/informatics223s2007/kramer.pdf.

Krantz, S. G. (1984, November). Letter to the editor. *American Mathematical Monthly 91*(9), 598–600.

Krauss, L. M. (2016, 29 September). Gravity's black rainbow. *New York Review of Books 63*(14), 83–85. http://www.nybooks.com/articles/2016/09/29/gravitys-black-rainbow/.

Krebs, A. and R. M. Thomas, Jr. (1981, 7 August). Historic moth. *New York Times*. http://www.nytimes.com/1981/08/07/nyregion/notes-on-people-historic-moth.html.

Kreisel, G. (1987). Church's thesis and the ideal of informal rigour. *Notre Dame Journal of Formal Logic 28*(4), 499–519.

Kripke, S. A. (2011). Vacuous names and fictional entities. In S. A. Kripke (Ed.), *Philosophical Troubles: Collected Papers, Volume 1*, pp. 52–74. New York: Oxford University Press. https://www.phil.pku.edu.cn/documents/20120822193331_Vacuous_Names_and_Fictional_Entities.pdf.

Kripke, S. A. (2013). The Church-Turing 'thesis' as a special corollary of Gödel's completeness theorem. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 77–104. Cambridge, MA: MIT Press.

Kuang, C. (2017, 26 November). Can A.I. be taught to explain itself? *New York Times Magazine*, MM46ff. https://nyti.ms/2hR2weQ.

Kuczynski, J.-M. (2015, 31 December). Is the human brain a computer? Alan Turing on mind & computers. *The Critique*. http://www.thecritique.com/articles/is-the-human-brain-a-computer-alan-turing-on-mind-computers/.

Kugel, P. (1986a, March). Thinking may be more than computing. *Cognition 22*(2), 137–198. http://cs.bc.edu/~kugel/Publications/Cognition.pdf.

Kugel, P. (1986b, June). When is a computer not a computer? *Cognition 23*(1), 89–94. http://cs.bc.edu/~kugel/Publications/When.pdf.

Kugel, P. (2002, November). Computing machines can't be intelligent (... and Turing said so). *Minds and Machines 12*(4), 563–579. http://cs.bc.edu/~kugel/Publications/Hyper.pdf.

Kugel, P. (2004). Toward a theory of intelligence. *Theoretical Computer Science 317*, 13–30. http://cs.bc.edu/~kugel/Publications/Toward%20a%20theory%20of%20intelligence.pdf.

Kugel, P. (2005, November). It's time to think outside the computational box. *Communications of the ACM 48*(11), 33–37. http://cs.bc.edu/~kugel/Publications/BoxTextandPictures.pdf.

Kuhn, T. S. (1957). *The Copernican Revolution: Planetary Astronomy in the Development of Western Thought*. Cambridge, MA: Harvard University Press.

Kuhn, T. S. (1962). *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press.

Kumar, D. (1994). From beliefs and goals to intentions and actions: An amalgamated model of inference and acting. Unpublished PhD dissertation, Department of Computer Science, SUNY Buffalo, http://www.cse.buffalo.edu/tech-reports/94-04.ps.

Kumar, D. (1996, January). The SNePS BDI architecture. *Decision Support Systems 16*(1), 3–19. http://www.cse.buffalo.edu/sneps/Bibliography/kum96.pdf.

LaChat, M. R. (1986). Artificial Intelligence and ethics: An exercise in the moral imagination. *AI Magazine 7*(2), 70–79.

Ladyman, J. (2009, 17 February). What does it mean to say that a physical system implements a computation? *Theoretical Computer Science 410*(4–5), 376–383. https://www.sciencedirect.com/science/article/pii/S0304397508007238.

Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. Chicago: University of Chicago Press.

Lakoff, G. and M. Johnson (1980a). Conceptual metaphor in everyday language. *Journal of Philosophy 77*(8), 453–486.

Lakoff, G. and M. Johnson (1980b). *Metaphors We Live By*. Chicago: University of Chicago Press.

Lammens, J. (1990). Universal program. http://www.cse.buffalo.edu/~rapaport/lammens.lisp.

Lamport, L. (2011). Euclid writes an algorithm: A fairytale. *International Journal of Software and Informatics 5*(1-2, Part 1), 7–20. Page references to PDF version at http://research.microsoft.com/en-us/um/people/lamport/pubs/euclid.pdf.

Lamport, L. (2012, March). How to write a 21st century proof. *Journal of Fixed Point Theory and Applications 11*(1), 43–63. http://research.microsoft.com/en-us/um/people/lamport/pubs/proof.pdf.

Lamport, L. (2015, April). Who builds a house without drawing blueprints? *Communications of the ACM 58*(4), 38–41. http://cacm.acm.org/magazines/2015/4/184705-who-builds-a-house-without-drawing-blueprints/fulltext.

Landgrebe, J. and B. Smith (2019a). Making AI meaningful again. *Synthese*. https://doi.org/10.1007/s11229-019-02192-y.

Landgrebe, J. and B. Smith (2019b, 14 June). There is no general AI: Why Turing machines cannot pass the Turing test. https://arxiv.org/pdf/1906.05833.pdf.

Langacker, R. (1999). *Foundations of Cognitive Grammar*. Stanford, CA: Stanford University Press.

Langewiesche, W. (2009). *Fly by Wire: The Geese, the Glide, the Miracle on the Hudson*. New York: Farrar, Strauss & Giroux.

Langton, R. and D. Lewis (1998, 1998). Defining 'intrinsic'. *Philosophy and Phenomenological Research 58*(2), 333–345. http://web.mit.edu/langton/www/pubs/DefiningIntrinsic.pdf.

Lanier, J. (2005, July-August). Early computing's long, strange trip. *American Scientist 93*(4), 364–365. http://www.americanscientist.org/bookshelf/pub/early-computings-long-strange-trip.

Lazowska, E. (2014, 16 May). Letter to the editor. *New York Times*, A26. http://www.nytimes.com/2014/05/16/opinion/should-young-children-learn-coding.html.

Leavitt, D. (2005). *The Man Who Knew Too Much: Alan Turing and the Invention of the Computer*. W.W. Norton.

Lee, J. (1994). Konrad Zuse. *The History of Computing*. http://ei.cs.vt.edu/~history/Zuse.html.

Lehoux, D. and J. Foster (2012, 16 November). A revolution of its own. *Science 338*, 885–886.

Leiber, J. (2006). Turing's golden: How well Turing's work stands today. *Philosophical Psychology 19*(1), 13–46.

Leibniz, G. W. (1677). Towards a universal characteristic. In P. P. Wiener (Ed.), *Leibniz: Selections*, pp. 17–25. New York: Charles Scribner's Sons, 1951. http://www.rbjones.com/rbjpub/philos/classics/leibniz/meth_math.htm.

Leibniz, G. W. (1683–1685). Introduction to a secret encyclopedia. In M. Dascal (Ed.), *G.W. Leibniz: The Art of Controversies*, pp. 219–224. Dordrecht, The Netherlands: Springer (2008). http://tinyurl.com/Leibniz1683.

Leibniz, G. W. (1714). The Principles of Philosophy Known as Monadology. Trans. by Jonathan Bennett (July 2007). Accessed from *Some Texts from Early Modern Philosophy*, http://www.earlymoderntexts.com/assets/pdfs/leibniz1714b.pdf.

Leiter, B. (12 October 2005). Why is there a Nobel prize in economics? *Leiter Reports: A Philosophy Blog*, http://leiterreports.typepad.com/blog/2005/10/why_is_there_a_.html.

Leiter, B. (20 September 2009). Alex Rosenberg on Cochrane and economics. *Leiter Reports: A Philosophy Blog*, http://leiterreports.typepad.com/blog/2009/09/alex-rosenberg-on-cochrane-and-economics.html.

Leiter, B. (8 October 2004). Is economics a 'science'? *Leiter Reports: A Philosophy Blog*, note = http://leiterreports.typepad.com/blog/2004/10/is_economics_a_.html.

Leler, W. (3 December 1985). net.ai newsgroup message; contribution to discussion of "definition of AI". https://www.cse.buffalo.edu//~rapaport/definitions.of.ai.html.

Lem, S. (1971). Non serviam. In *A Perfect Vacuum*. New York: Harcourt Brace Jovanovich. 1979.

Lemonick, M. (2015, 6 March). The Pluto wars revisited. *The New Yorker* online, http://www.newyorker.com/tech/elements/nasa-dawn-ceres-pluto-dwarf-planets.

Lepore, J. (2018). *These Truths: A History of the United States*. New York: W.W. Norton.

Leroy, X. (2009, July). Formal verification of a realistic compiler. *Communications of the ACM 52*(7), 107–115. http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf.

Lessing, G. E. (1778). Anti-Goetze: Eine Duplik. In H. Göpfert (Ed.), *Werke*, pp. Vol. 8, pp. 32–33. (published online). http://harpers.org/blog/2007/11/lessings-search-for-truth/.

Levesque, H. J. (2009). Is it enough to get the behaviour right? In *IJCAI'09: Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pp. 1439–1444. San Francisco: Morgan Kaufmann. https://www.ijcai.org/Proceedings/09/Papers/241.pdf.

Levesque, H. J. (2017). *Common Sense, the Turing Test, and the Quest for Real AI*. Cambridge, MA: MIT Press.

Levin, J. (2017). Functionalism. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2017 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2017/entries/functionalism/.

Levy, S. (1984, November). A spreadsheet way of knowledge. *Harper's Magazine*. Reprinted in *Wired* (24 October 2014), https://www.wired.com/2014/10/a-spreadsheet-way-of-knowledge/.

Levy, S. (2013, November). 101 objects that made america: The brief history of the ENIAC computer. *Smithsonian*, 62–64. http://www.smithsonianmag.com/history-archaeology/The-Brief-History-of-the-ENIAC-Computer-228879421.html.

Lewis, D. (1970, December). General semantics. *Synthese 22*(1/2), 18–67.

Lewis, D. (1983, September). Extrinsic properties. *Philosophical Studies 44*(2), 197–200.

Lewis, W. (1953, October). Electronic computers and telephone switching. *Proceedings of the Institute of Radio Engineers 41*(10), 1242–1244.

Lewis-Kraus, G. (2016, 14 December). The great A.I. awakening. *New York Times Magazine*. http://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html.

Lewontin, R. (2014, 8 May). The new synthetic biology: Who gains? *New York Review of Books 61*(8), 22–23. http://www.nybooks.com/articles/archives/2014/may/08/new-synthetic-biology-who-gains/.

Liao, M.-H. (1998). Chinese to English machine translation using SNePS as an interlingua. Unpublished PhD dissertation, Department of Linguistics, SUNY Buffalo, http://www.cse.buffalo.edu/sneps/Bibliography/tr97-16.pdf.

Libbey, P. and K. A. Appiah (2019, 11 April). Socrates questions, a contemporary philosopher answers. *New York Times*. https://www.nytimes.com/2019/04/11/theater/socrates-democracy-public-theater.html.

Licklider, J. and R. W. Taylor (1968, April). The computer as a communication device. *Science and Technology*. http://memex.org/licklider.pdf.

Lindell, S. (2004). Revisiting finite-visit computations. http://www.haverford.edu/cmsc/slindell/Presentations/Revisiting%20finite-visit%20computations.pdf.

Lindell, S. (2006). A physical analysis of mechanical computability. http://www.haverford.edu/cmsc/slindell/Presentations/A%20physical%20analysis%20of%20mechanical%20computability.pdf.

Lindell, S. (24 January 2001). Computer science as a liberal art: The convergence of technology and reason. Talk presented at Haverford College, http://www.haverford.edu/cmsc/slindell/Presentations/Computer%20Science%20as%20a%20Liberal%20Art.pdf.

Linker, D. (2014, 6 May). Why Neil deGrasse Tyson is a Philistine. *The Week*. http://theweek.com/article/index/261042/why-neil-degrasse-tyson-is-a-philistine.

Linker, D. (2015, 1 July). No, your brain isn't a computer. *The Week*. http://theweek.com/articles/563975/no-brain-isnt-computer.

Linnebo, Ø. (2018). Platonism in the philosophy of mathematics. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2018 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2018/entries/platonism-mathematics/.

Lipton, P. (2004). *Inference to the Best Explanation, 2nd Edition*. Routledge.

Lipton, Z. C. (2016). The mythos of model interpretability. In *2016 ICML Workshop on Human Interpretability in Machine Learning (WHI 2016)*. https://arxiv.org/abs/1606.03490.

Liskov, B. and S. Zilles (1974, April). Programming with abstract data types. *ACM SIGPLAN Notices 9*(4), 50–59. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.3043&rep=rep1&type=pdf.

Livnat, A. and C. Papadimitriou (2016, November). Sex as an algorithm: The theory of evolution under the lens of computation. *Communications of the ACM 59*(11), 84–93. http://cacm.acm.org/magazines/2016/11/209128-sex-as-an-algorithm/fulltext.

Lloyd, S. (1990, September/October). The calculus of intricacy: Can the complexity of a forest be compared with that of *finnegans wake*? *The Sciences 38*(44), 38–44.

Lloyd, S. (2000, 31 August). Ultimate physical limits to computation. *Nature 406*, 1047–1054. http://arxiv.org/pdf/quant-ph/9908043.pdf?origin=publicationDetail and http://cds.cern.ch/record/396654/files/9908043.pdf.

Lloyd, S. (2002, June). Computational capacity of the universe. *Physical Review Letters 88*(23), 237901–1 – 237901–4. http://fab.cba.mit.edu/classes/862.16/notes/computation/Lloyd-2002.pdf.

Lloyd, S. (2006). *Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos*. New York: Alfred A. Knopf.

Lloyd, S. and Y. J. Ng (2004, November). Black hole computers. *Scientific American 291*(5), 52–61.

Locke, J. (1694). *An Essay concerning Human Understanding*. Oxford: Oxford University Press, 1975. Edited by Peter H. Nidditch.

Lodder, J. (2014, 18 July). Introducing logic via Turing machines. http://www.math.nmsu.edu/hist_projects/j13.html.

Loebner, H. G. (1994). In response [to Shieber 1994a]. *Communications of the ACM 37*(6), 79–82. https://web.archive.org/web/20040206132330/http://www.loebner.net/Prizef/In-response.html.

Lohr, S. (1996, 19 February). The face of computing 50 years and 18,000 tubes ago. *New York Times*, D3. http://www.nytimes.com/1996/02/19/business/the-face-of-computing-50-years-and-18000-tubes-ago.html.

Lohr, S. (2001, 17 December). Frances E. Holberton, 84, early computer programmer. *New York Times*, F5. http://www.nytimes.com/2001/12/17/business/frances-e-holberton-84-early-computer-programmer.html.

Lohr, S. (2002, 6 August). Scientist at work: Frances Allen; would-be math teacher ended up educating a computer revolution. *New York Times*, F3. http://www.nytimes.com/2002/08/06/science/scientist-work-frances-allen-would-be-math-teacher-ended-up-educating-computer.html.

Lohr, S. (2006, 2 November). Group of university researchers to make Web science a field of study. *New York Times*, C6. http://www.nytimes.com/2006/11/02/technology/02compute.html.

Lohr, S. (2008, 1 April). Does computing add up in the classroom? *New York Times Bits (blog)*. http://bits.blogs.nytimes.com/2008/04/01/does-computing-add-up-in-the-classroom/.

Lohr, S. (2010, 3 April). Inventor whose pioneer PC helped inspire Microsoft dies. *New York Times*, A1–A3. http://www.nytimes.com/2010/04/03/business/03roberts.html.

Lohr, S. (2013, 10 March). Algorithms get a human hand in steering Web. *New York Times*. http://www.nytimes.com/2013/03/11/technology/computer-algorithms-rely-increasingly-on-human-helpers.html.

Loui, M. C. (1987, December). Computer science is an engineering discipline. *Engineering Education 78*(3), 175–178.

Loui, M. C. (1995, March). Computer science is a new engineering discipline. *ACM Computing Surveys 27*(1), 31–32.

Loui, M. C. (1996, March). Computational complexity theory. *ACM Computing Surveys 28*(1), 47–49. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.9179&rep=rep1&type=pdf.

Loui, R. (1998). Review of Smith 1996. *Artificial Intelligence 106*, 353–358.

Lu, J. J. and G. H. Fletcher (2009, March). Thinking about computational thinking. *SIGCSE Bulletin 41*(1), 260–264. https://ai2-s2-pdfs.s3.amazonaws.com/66b0/fb58a6091d3fec8c63046acd0eecff587c9f.pdf.

Lu, T. K. and O. Purcell (2016, April). Machine life. *Scientific American 314*(4), 58–63.

Lycan, W. G. (1990). The continuity of levels of nature. In W. G. Lycan (Ed.), *Mind and Cognition: A Reader*, pp. 77–96. Cambridge, MA: Basil Blackwell. Excerpted from Chs. 4–5 of William G. Lycan, *Consciousness* (Cambridge, MA: MIT Press, 1987).

Macari, M. (2012, 13 April). Oracle thinks you can copyright a programming language, Google disagrees. *The Verge*. http://www.theverge.com/2012/4/13/2944440/google-oracle-lawsuit-programming-language-copyright.

MacFarlane, A. (2013). Ada Lovelace (1815–1852). *Philosophy Now 96*. http://philosophynow.org/issues/96/Ada_Lovelace_1815-1852.

Machamer, P., L. Darden, and C. F. Craver (2000, March). Thinking about mechanisms. *Philosophy of Science 67*(1), 1–25. https://mechanism.ucsd.edu/teaching/w10/machamer.darden.craver.pdf.

Machery, E. (2012). Why i stopped worrying about the definition of life ... and why you should as well. *Synthese 185*, 145–164. http://www.pitt.edu/~machery/papers/Definition%20of%20life%20Synthese%20machery.pdf.

Machlup, F. and U. Mansfield (Eds.) (1983). *The Study of Information: Interdisciplinary Messages*. New York: John Wiley & Sons.

MacKenzie, D. (1992, November). Computers, formal proofs, and the law courts. *Notices of the American Mathematical Society 39*(9), 1066–1069. Also see introduction by Keith Devlin (1992).

MacKenzie, D. (2001). *Mechanizing Proof: Computing, Risk, and Trust*. Cambridge, MA: MIT Press.

Madigan, T. (2014, January/February). A mind is a wonderful thing to meet. *Philosophy Now 100*, 46–47. http://philosophynow.org/issues/100/A_Mind_is_a_Wonderful_Thing_to_Meet.

Mahoney, M. S. (2011). *Histories of Computing*. Cambridge, MA: Harvard University Press. Edited by Thomas Haigh.

Maida, A. S. and S. C. Shapiro (1982). Intensional concepts in propositional semantic networks. *Cognitive Science 6*, 291–330. https://www.cse.buffalo.edu/~shapiro/Papers/maisha82.pdf. Reprinted in Ronald J. Brachman & Hector J. Levesque (eds.), *Readings in Knowledge Representation* (Los Altos, CA: Morgan Kaufmann, 1985): 169–189.

Malpas, R. (2000, June). The universe of engineering: A UK perspective. Technical report, Royal Academy of Engineering, London. http://www.engc.org.uk/ecukdocuments/internet/document%20library/The%20Universe%20of%20Engineering%20Report%20(The%20Malpas%20Report).pdf.

Manchak, J. and B. W. Roberts (2016). Supertasks. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2016 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2016/entries/spacetime-supertasks/.

Manchak, J. B. (2018). Malament-Hogarth machines. *British Journal for the Philosophy of Science forthcoming*. https://doi.org/10.1093/bjps/axy023.

Mander, K. (February 2007). Demise of computer science exaggerated. BCS: The Chartered Institute for IT; Features, Press and Policy, http://www.bcs.org/content/ConWebDoc/10138.

Manovich, L. (2013, 20 December). The algorithms of our lives. *The Chronicle [of Higher Education] Review 60*(16), B10–B13. https://chronicle.com/article/The-Algorithms-of-Our-Lives-/143557/.

Manzano, M. (1997). Alonzo Church: His life, his work and some of his miracles. *History and Philosophy of Logic 18*, 211–232.

Marcus, G. (2015, 28 June). Face it, your brain is a computer. *New York Times Sunday Review*, SR12. http://www.nytimes.com/2015/06/28/opinion/sunday/face-it-your-brain-is-a-computer.html.

Marcus, G., F. Rossi, and M. Veloso (Eds.) (2016). *Beyond the Turing Test*. Special issue of *AI Magazine* 37 (1, Spring). https://aaai.org/ojs/index.php/aimagazine/issue/view/213.

Markoff, J. (2000, 18 September). A tale of the tape from the days when it was still Micro Soft. *New York Times*, C1, C4. http://www.nytimes.com/2000/09/18/business/technology-a-tale-of-the-tape-from-the-days-when-it-was-still-micro-soft.html.

Markoff, J. (2002, 10 August). Edsger Dijkstra, 72, physicist who shaped computer era'. *New York Times*, A11. http://www.nytimes.com/2002/08/10/obituaries/10DIJK.html.

Markoff, J. (2005). *What the Dormouse Said: How the Sixties Counterculture Shaped the Personal Computer Industry*. New York: Viking. Reviewed in Lanier (2005).

Markoff, J. (2011, 8 November). It started digital wheels turning. *New York Times/Science Times*, D1, D4. http://www.nytimes.com/2011/11/08/science/computer-experts-building-1830s-babbage-analytical-engine.html.

Markoff, J. (2015, 7 April). Planes without pilots. *New York Times*, D1, D4. http://www.nytimes.com/2015/04/07/science/planes-without-pilots.html.

Markov, A. (1954). Theory of algorithms. *Tr. Mat. Inst. Steklov 42*, 1–14. trans. by Edwin Hewitt, in *American Mathematical Society Translations*, Series 2, Vol. 15 (1960).

Marr, D. (1977). Artificial Intelligence: A personal view. *Artificial Intelligence 9*, 37–48. Reprinted in Partridge and Wilks 1990, pp. 97–107.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. New York: W.H. Freeman.

Marshall, D. (2016, March). The varieties of intrinsicality. *Philosophy and Phenomenological Research 92*(2), 237–263. https://www.academia.edu/7096450/The_Varieties_of_Intrinsicality.

Martin, C. (2015, 1 May). She seconds that emotion. *The Chronicle [of Higher Education] Review 61*(33), B16. http://chronicle.com/article/Choosing-Love/229569/.

Martin, D. (2008, 29 June). David Caminer, 92, dies; a pioneer in computers. *New York Times*, 28. http://www.nytimes.com/2008/06/29/technology/29caminer.html.

Martin, D. (2013, 24 November). Mavis Batey, 92, Allied code breaker in World War II. *New York Times*, 32. http://www.nytimes.com/2013/11/23/world/europe/mavis-batey-world-war-ii-code-breaker-dies-at-92.html.

Martins, J. a. P. and S. C. Shapiro (1988). A model for belief revision. *Artificial Intelligence 35*(1), 25–79. http://www.cse.buffalo.edu/~shapiro/Papers/marsha88.pdf.

Marx, K. (1845). Theses on Feuerbach. https://www.marxists.org/archive/marx/works/1845/theses/theses.htm.

Matthews, R. J. and E. Dresner (2017). Measurement and computational skepticism. *Nouûs 51*(4), 832–854.

Mauchly, B., J. Bernstein, M. Dowson, D. K. Adams, and J. Holt (2012, 27 September). Who gets credit for the computer?: An exchange. *New York Review of Books*, 96, 98.

McAllister, N. (2012, 19 April). Oracle vs. Google: Who owns the Java APIs? *InfoWorld*. http://www.infoworld.com/article/2617268/java/oracle-vs\verb2--2google\verb2--2who-owns-the-java-apis-.html.

McBride, N. (22 January 2007). The death of computing. BCS: The Chartered Institute for IT; Features, Press and Policy, urlhttp://www.bcs.org/content/ConWebDoc/9662.

McCarthy, J. (1 November 1988). Contribution to newsgroup discussion of "AI as CS and the scientific epistemology of the common sense world". Article 1818 of comp.ai.digest, http://www.cse.buffalo.edu/~rapaport/mccarthy.txt.

McCarthy, J. (12 November 2007). What is AI? http://www-formal.stanford.edu/jmc/whatisai.html.

McCarthy, J. (1959). Programs with common sense. In D. Blake and A. Uttley (Eds.), *Proceedings of the ["Teddington"] Symposium on Mechanization of Thought Processes*. London: HM Stationary Office. Original version at http://aitopics.org/sites/default/files/classic/TeddingtonConference/Teddington-1.3-McCarthy.pdf; McCarthy archive version at http://www-formal.stanford.edu/jmc/mcc59/mcc59.html.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg (Eds.), *Computer Programming and Formal Systems*. North-Holland. Page references to PDF version at http://www-formal.stanford.edu/jmc/basis.html.

McCarthy, J. (1974). Review of 'Artificial Intelligence: A General Survey'. *Artificial Intelligence 5*, 317–322.

McCarthy, J. and P. J. Hayes (1969). Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 4*. Edinburgh: Edinburgh University Press. http://www-formal.stanford.edu/jmc/mcchay69.html.

McCarthy, J., M. Minsky, N. Rochester, and C. Shannon (31 August 1955). A proposal for the Dartmouth Summer Research Project on Artificial Intelligence. http://www-formal.stanford.edu/jmc/history/dartmouth.html.

McCulloch, W. S. and W. H. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 7*, 114–133. http://www.cs.cmu.edu/~epxing/Class/10715/reading/McCulloch.and.Pitts.pdf. Reprinted in Warren S. McCulloch, *Embodiments of Mind* (Cambridge, MA: MIT Press, 1965): 19–39.

McDermott, D. (1980). Artificial intelligence meets natural stupidity. In J. Haugeland (Ed.), *Mind Design: Philosophy, Psychology, Artificial Intelligence*, pp. 143–160. Cambridge, MA: MIT Press. http://www.inf.ed.ac.uk/teaching/courses/irm/mcdermott.pdf.

McDermott, D. (2001). *Mind and Mechanism*. Cambridge, MA: MIT Press.

McDermott, D. (2014, 31 December). What was Alan Turing's imitation game? https://web.archive.org/web/20150105021303/http://www.thecritique.com/articles/what-was-alan-turings-imitation-game/.

McGinn, C. (1989, July). Can we solve the mind-body problem? *Mind 98*(391), 349–366. http://mind.oxfordjournals.org/content/XCVIII/391/349.full.pdf.

McGinn, C. (1993). *Problems in Philosophy: The Limits of Inquiry*. Oxford: Blackwell.

McGinn, C. (2015a). *Philosophy of Language: The Classics Explained*. Cambridge, MA: MIT Press.

McGinn, C. (2015b, January). The science of philosophy. *Metaphilosophy 46*(1), 84–103. http://onlinelibrary.wiley.com/store/10.1111/meta.12116/asset/meta12116.pdf?v=1&t=i5dvrsm5&s=210194c1272c25e8bac66889cc9db416aac0340d; video at https://www.youtube.com/watch?v=TEkTbq5EE_M; preprint at https://docs.google.com/file/d/0BzokFqaWjk4JZ0NPMHFiUnNnZWs/edit.

McGinn, C. (4 March 2012b). Philosophy by another name. *The Stone/The New York Times Opinionator*, http://opinionator.blogs.nytimes.com/2012/03/04/philosophy-by-another-name/.

McGinn, C. (9 March 2012a). Name calling: Philosophy as ontical science. *The Stone/The New York Times Opinionator*, http://opinionator.blogs.nytimes.com/2012/03/09/name-calling-philosophy-as-ontical-science/.

McMillan, R. (2013, 7 July). The end of digital tyranny: Why the future of computing is analog. *Wired*. http://www.wired.com/wiredenterprise/2013/07/analogfuture/.

McSherry, C. (2014, 9 May). Dangerous decision in Oracle v. Google: Federal circuit reverses sensible court ruling on APIs. *Electronic Frontier Foundation*, https://www.eff.org/deeplinks/2014/05/dangerous-ruling-oracle-v-google-federal-circuit-reverses-sensible-lower-court.

Mearian, L. (2013, 12 February). AI found better than doctors at diagnosing, treating patients. *Computerworld*. http://www.computerworld.com/article/2494918/healthcare-it/ai-found-better-than-doctors-at-diagnosing--treating-patients.html.

Meigs, J. B. (2012, 10 December). Inside the future: How popmech predicted the next 110 years. *Popular Mechanics*. https://www.popularmechanics.com/technology/a8562/inside-the-future-how-popmech-predicted-the-next-110-years-14831802/.

Melville, H. (1851). *Moby-Dick (Norton Critical Edition, second edition, 2002)*. New York: W.W. Norton. Hershel Parker & Harrison Hayford (eds.).

Menabrea, L. F. and A. A. Lovelace (1843, September 1843). English translation of *notions sur la machine analytique de m. charles babbage*. *[Richard Taylor's] Scientific Memoirs 3*, 666ff. Translation by Lovelace; translation at https://psychclassics.yorku.ca/Lovelace/menabrea.htm, Lovelace's notes at https://psychclassics.yorku.ca/Lovelace/lovelace.htm.

Menand, L. (2014, 20 October). Crooner in rights spat: Are copyright laws too strict? *New Yorker*, 84–89. http://www.newyorker.com/magazine/2014/10/20/crooner-rights-spat.

Mendelson, E. (1990, May). Second thoughts about Church's thesis and mathematical proofs. *Journal of Philosophy 87*(5), 225–233.

Mertens, S. (2004, March-April). The revolution will be digitized. *American Scientist 92*, 195–196.

Mervis, C. B. and E. Rosch (1981). Categorization of natural objects. *Annual Review of Psychology 32*, 89–115.

Metz, C. (2017, 6 November). Building A.I. that can build A.I. *New York Times*, B1. https://nyti.ms/2j1KU0d.

Metz, C. (2018, 7 March). Google researchers are learning how machines learn. *New York Times*, B3. https://www.nytimes.com/2018/03/06/technology/google-artificial-intelligence.html.

Metz, C. (2019a, 16 August). A.I. is learning from humans. many humans. *New York Times*. https://www.nytimes.com/2019/08/16/technology/ai-humans.html.

Metz, C. (2019b, 27 March). Turing award won by 3 pioneers in artificial intelligence. *New York Times*. https://www.nytimes.com/2019/03/27/technology/turing-award-ai.html.

Micali, S. (2015, January). What it means to receive the Turing award. *Communications of the ACM 58*(1), 52–53. http://cacm.acm.org/magazines/2015/1/181611-what-it-means-to-receive-the-turing-award/fulltext.

Michaelson, G. (2012, April). A visit to the Turing machine: A short story. *CS4FN: Computer Science for Fun 14*. http://www.cs4fn.org/turing/avisittotheturingmachine.php.

Michie, D. (1961). Trial and error. *Science Survey 2*, 129–145. Reprinted in Donald Michie, *On Machine Intelligence* (New York: John Wiley, 1974): 5–19; see also Donald Michie, "Experiments on the Mechanization of Game-Learning Part I. Characterization of the Model and Its Parameters", http://comjnl.oxfordjournals.org/content/6/3/232.full.pdf.

Michie, D. (1971). Formation and execution of plans by machine. In N. Findler and B. Meltzer (Eds.), *Artificial Intelligence and Heuristic Programming*, pp. 101–124. New York: American Elsevier.

Michie, D. (2008). Alan Turing's mind machines. In P. Husband, O. Holland, and M. Wheeler (Eds.), *The Mechanical Mind in History*, pp. Ch. 4, pp. 61–74. Cambridge, MA: MIT Press.

Mickevich, A. P. (1961, 15 June 2018). The game. http://www.hardproblem.ru/en/posts/Events/a-russian-chinese-room-story-antedating-searle-s-1980-discussion/. Written under the pseudonym "A. Dneprov".

Mili, A., J. Desharnais, and J. R. Gagné (1986, September). Formal models of stepwise refinement of programs. *ACM Computing Surveys 18*(3), 231–276.

Miłkowski, M. (2018). From computer metaphor to computational modeling: The evolution of computationalism. *Minds and Machines*. https://doi.org/10.1007/s11023-018-9468-3.

Miller, C. A. e. (2004, April). Human-computer etiquette: Managing expectations with intentional agents. *Communications of the ACM 47*(4), 30–61.

Miller, G. A., E. Galanter, and K. H. Pribram (1960). *Plans and the Structure of Behavior*. New York: Henry Holt.

Mills, H. D. (1971). Top-down programming in large systems. In R. Rustin (Ed.), *Debugging Techniques in Large Systems*, pp. 41–55. Englewood Cliffs, NJ: Prentice-Hall.

Milner, R. (1993, January). Elements of interaction: Turing Award lecture. *Communications of the ACM 36*(1), 78–89. http://delivery.acm.org/10.1145/160000/151240/a1991-milner.pdf.

Minsky, M. (1967). *Computation: Finite and Infinite Machines*. Englewood Cliffs, NJ: Prentice-Hall.

Minsky, M. (1968). Preface. In M. Minsky (Ed.), *Semantic Information Processing*, pp. v. Cambridge, MA: MIT Press.

Minsky, M. (1979). Computer science and the representation of knowledge. In L. Dertouzos and J. Moses (Eds.), *The Computer Age: A Twenty Year View*, pp. 392–421. Cambridge, MA: MIT Press.

Mish, F. C. (Ed.) (1983). *Webster's Ninth New Collegiate Dictionary*. Springfield, MA: Merriam-Webster.

Mitcham, C. (1994). *Thinking through Techology: The Path between Engineering and Philosophy*. Chicago: University of Chicago Press.

Mitchell, M. (2011, February). What is computation? Biological computation. *Ubiquity 2011*(February). Article 3, http://ubiquity.acm.org/article.cfm?id=1944826.

Mithen, S. (2016, 24 November). Our 86 billion neurons: She showed it. *New York Review of Books 63*(18), 42–44. http://www.nybooks.com/articles/2016/11/24/86-billion-neurons-herculano-houzel/.

Mizoguchi, R. and Y. Kitamura (2009). A functional ontology of artifacts. *The Monist 92*(3), 387–402.

Monroe, C. R. and D. J. Wineland (2008, August). Quantum computing with ions. *Scientific American*, 64–71. http://www.cs.virginia.edu/~robins/Quantum_Computing_with_Ions.pdf.

Montague, R. (1960, December). Towards a general theory of computability. *Synthese 12*(4), 429–438.

Montague, R. (1970). English as a formal language. In R. H. Thomason (Ed.), *Formal Philosophy: Selected Papers of Richard Montague*, pp. 192–221. New Haven, CT: Yale University Press, 1974. http://strangebeautiful.com/uwo/metaphys/montague-formal-philosophy.pdf.

Monticello, M. (2016, May). The state of the self-driving car; will self-driving cars make our road safer? *Consumer Reports 81*(5), 44–49. https://www.consumerreports.org/self-driving-cars/state-of-the-self-driving-car/ and https://www.consumerreports.org/self-driving-cars/will-self-driving-cars-make-our-roads-safer/.

Moody, T. C. (1986, January). Progress in philosophy. *American Philosophical Quarterly 23*(1), 35–46.

Moody, T. C. (1993). *Philosophy and Artificial Intelligence*. Englewood Cliffs, NJ: Prentice-Hall.

Mooers, C. N. (1975, March). Computer software and copyright. *Computing Surveys 7*(1), 45–72.

Moor, J. H. (1978, September). Three myths of computer science. *British Journal for the Philosophy of Science 29*(3), 213–222.

Moor, J. H. (1979). Are there decisions computers should never make? *Nature and System 1*, 217–229. https://www.researchgate.net/publication/242529825_Are_There_Decisions_Computers_Should_Never_Make.

Moor, J. H. (1985, October). What is computer ethics? *Metaphilosophy 16*(4), 266–275. http://web.cs.ucdavis.edu/~rogaway/classes/188/spring06/papers/moor.html.

Moor, J. H. (Ed.) (2003). *The Turing Test: The Elusive Standard of Artificial Intelligence*. Dordrecht, The Netherlands: Kluwer Academic.

Moravec, H. (1998, March). When will computer hardware match the human brain? *Journal of Evolution and Technology 1*(1). http://www.jetpress.org/volume1/moravec.htm and http://www.jetpress.org/volume1/moravec.pdf.

Morris, C. (1938). *Foundations of the Theory of Signs*. Chicago: University of Chicago Press.

Morris, F. and C. Jones (1984, April). An early program proof by alan turing. *Annals of the History of Computing 6*(2), 139–143. https://fi.ort.edu.uy/innovaportal/file/20124/1/09-turing_checking_a_large_routine_earlyproof.pdf.

Morris, G. J. and E. D. Reilly (2000). Digital computer. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, Fourth Edition*, pp. 539–545. New York: Grove's Dictionaries.

Morrisett, G. (2009, July). A compiler's story. *Communications of the ACM 52*(7), 106. https://www.deepdyve.com/lp/association-for-computing-machinery/technical-perspective-a-compiler-s-story-30zQuKanaE.

Moschovakis, Y. N. (1998). On founding the theory of algorithms. In H. Dales and G. Oliveri (Eds.), *Truth in Mathematics*, pp. 71–104. Oxford: Clarendon Press. http://www.math.ucla.edu/~ynm/papers/foundalg.ps.

Moschovakis, Y. N. (2001). What is an algorithm? In B. Engquist and W. Schmid (Eds.), *Mathematics Unlimited: 2001 and Beyond*, pp. 918–936. Berlin: Springer.

Moskin, J. (2018, 9 July). Overlooked no more: Fannie farmer, modern cookery's pioneer. *New York Times*, B5. https://www.nytimes.com/2018/06/13/obituaries/fannie-farmer-overlooked.html.

Muggleton, S. (1994). Logic and learning: Turing's legacy. In K. Furukawa, D. Michie, and S. Muggleton (Eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning*, pp. 37–56. Oxford: Clarendon Press.

Mukherjee, S. (2017, 3 April). The algorithm will see you now. *New Yorker 93*(7), 46–53. http://www.newyorker.com/magazine/2017/04/03/ai-versus-md.

Mukherjee, S. (2018, 7 January). This cat sensed death. what if computers could, too? *New York Times Magazine*, MM14. https://nyti.ms/2DXqzyd.

Munroe, R. (2015). *Thing Explainer: Complicated Stuff in Simple Words*. New York: Houghton Mifflin Harcourt.

Murphy, G. L. (2019, May). On fodor's first law of the nonexistence of cognitive science. *Cognitive Science 43*(5). https://doi.org/10.1111/cogs.12735.

Mycielski, J. (1983, February). The meaning of the conjecture $p \neq np$ for mathematical logic. *American Mathematical Monthly 90*, 129–130.

Myers, W. (1986, November). Can software for the strategic defense initiative ever be error-free? *IEEE Computer*, 61–67.

Myhill, J. (1972, August-September). What is a real number? *American Mathematical Monthly 79*(7), 748–754.

Nagel, E., J. R. Newman, and D. R. Hofstadter (2001). *Gödel's Proof, Revised Edition*. New York: New York University Press.

Nagel, T. (1987). *What Does It All Mean? A Very Short Introduction to Philosophy*. New York: Oxford University Press.

Nagel, T. (2016, 29 September). How they wrestled with the new (review of Gottlieb 2016). *New York Review of Books 63*(14), 77–79. http://www.nybooks.com/articles/2016/09/29/hobbes-spinoza-locke-leibniz-hume-wrestled-new/.

Nahmias, E., S. G. Morris, T. Nadelhoffer, and J. Turner (2006, July). Is incompatibilism intuitive? *Philosophy and Phenomenological Research 73*(1), 28–53.

Natarajan, P. (2014, 23 October). What scientists really do. *New York Review of Books 61*(16), 64–66. http://www.nybooks.com/articles/archives/2014/oct/23/what-scientists-really-do/.

Natarajan, P. (2017, 25 May). Calculating women. *New York Review of Books 64*(9), 38–39. http://www.nybooks.com/articles/2017/05/25/hidden-figures-calculating-women/.

Naur, P. (2007, January). Computing versus human thinking. *Communications of the ACM 50*(1), 85–94. http://www.computingscience.nl/docs/vakken/exp/Articles/NaurThinking.pdf.

Neumann, P. G. (1993, June). Modeling and simulation. *Communications of the ACM 36*(6), 124.

Neumann, P. G. (1996, July). Using formal methods to reduce risks. *Communications of the ACM 39*(7), 114.

Nevejans, N. (2016). European civil law rules in robotics. Technical Report PE 571.379, Directorate-General for Internal Policies, Policy Department C: Citizens' Rights and Constitutional Affairs, Brussels. http://www.europarl.europa.eu/RegData/etudes/STUD/2016/571379/IPOL_STU(2016)571379_EN.pdf.

New Scientist (2016, 2 January). Feedback. *New Scientist 229*(3054), 56. https://www.newscientist.com/article/mg22930541-200-feedback-ibm-feels-the-heat-over-hairdryer-hacking-campaign/.

Newcombe, C., T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff (2015, April). How Amazon web services uses formal methods. *Communications of the ACM 58*(4), 66–73. http://delivery.acm.org/10.1145/2700000/2699417/p66-newcombe.pdf and http://m.cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/fulltext.

Newell, A. (1980). Physical symbol systems. *Cognitive Science 4*, 135–183. http://repository.cmu.edu/cgi/viewcontent.cgi?article=3504&context=compsci.

Newell, A. (1985-1986). Response: The models are broken, the models are broken. *University of Pittsburgh Law Review 47*, 1023–1031.

Newell, A., A. J. Perlis, and H. A. Simon (1967, 22 September). Computer science. *Science 157*(3795), 1373–1374.

Newell, A., J. Shaw, and H. A. Simon (1958). Elements of a theory of human problem solving. *Psychological Review 65*(3), 151–166.

Newell, A. and H. A. Simon (1976, March). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM 19*(3), 113–126.

New York Times (2006, 17 September). Editorial: Bush restrained. pp. WK13. http://www.nytimes.com/2006/09/17/opinion/17sun1.html.

New York Times (2009, 8 November). Editorial: Quick, patent it! *New York Times*. http://www.nytimes.com/2009/11/08/opinion/08sun3.html.

Nicas, J., N. Kitroeff, D. Gelles, and J. Glanz (2019, 1 June). Boeing built deadly assumptions into 737 Max, blind to a late design change. *New York Times*. https://www.nytimes.com/2019/06/01/business/boeing-737-max-crash.html.

Nicas, J. and Z. Wichter (2019, 14 March). A worry for some pilots: Their hands-on flying skills are lacking. *New York Times*. https://www.nytimes.com/2019/03/14/business/automated-planes.html.

Nichols, S. (2011, 18 March). Experimental philosophy and the problem of free will. *Science 331*(6023), 1401–1403.

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill.

Nilsson, N. J. (1983, Winter). Artificial Intelligence prepares for 2001. *AI Magazine 4*(4), 7–14.

Northcott, R. and G. Piccinini (2018). Conceived this way: Innateness defended. *Philosophers' Imprint 18*(18), 1–16. https://quod.lib.umich.edu/p/phimp/3521354.0018.018/1.

O'Connor, J. and E. Robertson (1999). Abu Ja'far Muhammad ibn Musa Al-Khwarizmi. *The MacTutor History of Mathematics Archive*, http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Al-Khwarizmi.html.

O'Connor, J. and E. Robertson (October 2005). The function concept. *The MacTutor History of Mathematics Archive*, http://www-history.mcs.st-andrews.ac.uk/HistTopics/Functions.html.

O'Hanlon, R. (1982, 14 November). A calculating man. *New York Times Book Review*, BR26–BR27. Review of Hyman (1982).

OHeigeartaigh, S. (2013, 9 August). Would you hand over a moral decision to a machine? Why not? Moral outsourcing and Artificial Intelligence. *Practical Ethics*. http://blog.practicalethics.ox.ac.uk/2013/08/would-you-hand-over-a-moral-decision-to-a-machine-why-not-moral-outsourcing-and-artificial-intelligence/.

Ohlsson, S., R. H. Sloan, G. Turàn, and A. Urasky (2015, 11 September). Measuring an Artificial Intelligence system's performance on a verbal IQ test for young children. http://arxiv.org/abs/1509.03390, http://arxiv.org/pdf/1509.03390v1.pdf.

Oldehoeft, R., J. Montague, M. E. Thatcher, and D. E. Pingry (2007, December). Patented algorithms are bad, copyrighted software is good. *Communications of the ACM 50*(12), 9–10. (Letter to the Editor).

Olszewski, A., J. Woleński, and R. Janusz (Eds.) (2006). *Church's Thesis after 70 Years*. Frankfurt: Ontos Verlag.

O'Neill, O. (2013, March/April). Interpreting the world, changing the world. *Philosophy Now Issue 95*, 8–9. http://philosophynow.org/issues/95/Interpreting_The_World_Changing_The_World.

O'Neill, S. (2015, 11 June). The human race is a computer. *New Scientist 227*(3029), 26–27. Interview with César Hidalgo.

Oommen, B. J. and L. G. Rueda (2005, May). A formal analysis of why heuristic functions work. *Artificial Intelligence 164*(1-2), 1–22.

Oppy, G. and D. Dowe (2019). The turing test. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2019 ed.). Metaphysics Research Lab, Stanford University.

Oracle America, I. (2012, 12 April). Oracle's April 12, 2012 brief regarding copyright issues, case no. 3:10-CV 10-03561-WHA, document 900. http://assets.sbnation.com/assets/1057275/Oracle_s_Brief.pdf.

O'Regan, G. (2008). *A Brief History of Computing*. Springer.

Orr, H. A. (2013, 7 February). Awaiting a new darwin. *New York Review of Books 60*(2), 26–28.

O'Sullivan, M. (2017, 12 May). The untold story of QF72: What happens when 'psycho' automation leaves pilots powerless? *Sydney Morning Herald*. https://www.smh.com.au/lifestyle/the-untold-story-of-qf72-what-happens-when-psycho-automation-leaves-pilots-powerless-20170511-gw26ae.html.

O'Toole, G. (2016, 5 January). In a woman the flesh must be like marble; in a statue the marble must be like flesh. *Quote Investigator*, urlhttp://quoteinvestigator.com/2017/01/05/marble/.

Pandya, H. (2013, 23 April). Shakuntala Devi, 'human computer' who bested the machines, dies at 83. *New York Times*. http://www.nytimes.com/2013/04/24/world/asia/shakuntala-devi-human-computer-dies-in-india-at-83.html.

Papadimitriou, C. H. (2001, March-April). The sheer logic of IT. *American Scientist 89*, 168–171. http://www.americanscientist.org/bookshelf/pub/the-sheer-logic-of-it.

Papakonstantinou, Y. (2015, June). Created computed universe. *Communications of the ACM 58*(6), 36–38.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.

Papert, S. (1996). An exploration in the space of mathematics education. *International Journal of Computers for Mathematical Learning 1*(1), 95–123. http://www.papert.org/articles/AnExplorationintheSpaceofMathematicsEducations.html.

Papineau, D. (2017, 1 June). Is philosophy simply harder than science? *Times Literary Supplement Online*. http://www.the-tls.co.uk/articles/public/philosophy-simply-harder-science/.

Pappano, L. (2017, 9 April). Thinking in code/learning to think like a computer. *New York Times Education Life*, ED18. https://www.nytimes.com/2017/04/04/education/edlife/teaching-students-computer-code.html.

Park, E. (1996, February). The object at hand. *Smithsonian 26*(11), 20–23.

Parlante, N. (2005, June). What is computer science? *Inroads—The SIGSCE Bulletin 37*(2), 24–25.

Parnas, D. L. (1985). Software aspects of strategic defense systems. *American Scientist 73*(5), 432–440. Reprinted in *Communications of the ACM* 28(12) (December 1985): 1326–1335.

Parnas, D. L. (1998). Software engineering programmes are not computer science programs. *Annals of Software Engineering 6*, 19–37. Page references to pre-print at http://www.cas.mcmaster.ca/serg/papers/crl361.pdf; a paper with the same title (but American spelling as 'program') appeared in *IEEE Software* 16(6) (1990): 19–30.

Parnas, D. L. (2017, October). The real risks of artificial intelligence. *Communications of the ACM 60*(10), 27–31. http://www.csl.sri.com/users/neumann/cacm242.pdf.

Parsons, K. M. (2015, 8 April 2015). What is the public value of philosophy? *Huffington Post*. http://www.huffingtonpost.com/keith-m-parsons/what-is-the-public-value-of-philosophy_b_7018022.html.

Parsons, T. W. (1989, July). More on verification (letter to the editor). *Communications of the ACM 32*(7), 790–791.

Partridge, D. (1990). What's in an AI program? In D. Partridge and Y. Wilks (Eds.), *The Foundations of Artificial Intelligence: A Sourcebook*, pp. 112–118. Cambridge, UK: Cambridge University Press.

Partridge, D. and Y. Wilks (Eds.) (1990). *The Foundations of Artificial Intelligence: A Sourcebook*. Cambridge, UK: Cambridge University Press.

Paton, E. and V. Friedman (2018, 29 May). 'diamonds are forever,' and made by machine. *New York Times*. https://www.nytimes.com/2018/05/29/business/de-beers-synthetic-diamonds.html.

Pattis, R. E., J. Roberts, and M. Stehlik (1995). *Karel the Robot: A Gentle Introduction to the Art of Programming, Second Edition*. New York: John Wiley & Sons.

Paulson, L., A. Cohen, and M. Gordon (1989, March). The very idea (letter to the editor). *Communications of the ACM 32*(3), 375.

Pavese, C. (2015, November). Practical senses. *Philosophers' Imprint 15*(29), 1–25. http://hdl.handle.net/2027/spo.3521354.0015.029.

Pawley, A. L. (2009, October). Universalized narratives: Patterns in how faculty members define 'engineering'. *Journal of Engineering Education 98*(4), 309–319.

Peacocke, C. (1995). Content, computation and externalism. *Philosophical Issues 6*, 227–264.

Peacocke, C. (1999, June). Computation as involving content: A response to Egan. *Mind & Language 14*(2), 195–202.

Peano, G. (1889). The principles of arithmetic, presented by a new method. In J. van Heijenoort (Ed.), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pp. 83–97. Cambridge, MA: Harvard University Press.

Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge, UK: Cambridge University Press.

Pelletier, F. J. (1999). A history of natural deduction and elementary logic textbooks. *History and Philosophy of Logic 20*, 1–31. http://www.sfu.ca/~jeffpell/papers/NDHistory.pdf.

Penrose, R. (1989). *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics*. Oxford: Oxford University Press.

Perlis, A. (1962). The computer in the university. In M. Greenberger (Ed.), *Management and the Computer of the Future*, pp. 181–217. Cambridge, MA: MIT Press.

Perovic, K. (2017). Bradley's regress. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2017 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/win2017/entries/bradley-regress/.

Perruchet, P. and A. Vinter (2002, June). The self-organizing consciousness. *Behavioral and Brain Sciences 25*(3), 297–388. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.7056&rep=rep1&type=pdf.

Perry, Jr., W. G. (1970). *Forms of Intellectual and Ethical Development in the College Years: A Scheme*. New York: Holt, Rinehart and Winston.

Perry, Jr., W. G. (1981). Cognitive and ethical growth: The making of meaning. In A. Chickering and Associates (Eds.), *The Modern American College*, pp. 76–116. San Francisco: Jossey-Bass.

Peschl, M. and M. Scheutz (2001a). Some thoughts on computation and simulation in cognitive science. In *Proceedings of the 6th Congress of the Austrian Philosophical Society*, pp. 534–540. publisher unknown. Citations to online verison at http://hrilab.tufts.edu/publications/scheutzpeschl00linz.pdf.

Peschl, M. F. and M. Scheutz (2001b). Explicating the epistemological role of simulation in the development of theories of cognition. In *Proceedings of the 7th Colloquium on Cognitive Science (ICCS-01)*, pp. 274–280. publisher unknown. http://www.academia.edu/719102/Explicating_the_epistemological_role_of_simulation_in_the_development_of_theories_of_cognition.

Petersen, S. (2007, March). The ethics of robot servitude. *Journal of Experimental and Theoretical Artificial Intelligence 19*(1), 43–54. http://www.stevepetersen.net/petersen-ethics-robot-servitude.pdf.

Petersen, S. (2011). Designing people to serve. In P. Lin, K. Abney, and G. A. Bekey (Eds.), *Robot Ethics: The Ethical and Social Implications of Robotics*, pp. 283–298. Cambridge, MA: MIT Press. http://www.stevepetersen.net/petersen-designing-people.pdf.

Petroski, H. (2003, May-June). Early education. *American Scientist 91*, 206–209. http://childrensengineering.org/resources/Petroski.pdf.

Petroski, H. (2005, July-August). Technology and the humanities. *American Scientist 93*, 304–307.

Petroski, H. (2008a, September-October). Scientists as inventors. *American Scientist 96*(5), 368–371.

Petroski, H. (2008b, May-June). Twists, tags and ties. *American Scientist 96*(3), 188–192.

Petroski, H. (2010, January-February). Occasional design. *American Scientist 98*(1), 16–19.

Petzold, C. (2008). *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Indianapolis: Wiley.

"Philonous" (1919, 3 December). A slacker's apology. *The New Republic*, 19–20. Pseudonym of Morris Raphael Cohen.

Picard, R. (1997). *Affective Computing*. Cambridge, MA: MIT Press.

Piccinini, G. (2000). Turing's rules for the imitation game. *Minds and Machines 10*(4), 573–582.

Piccinini, G. (2003). Alan Turing and the mathematical objection. *Minds and Machines 13*, 23–48. http://www.umsl.edu/~piccininig/Alan_Turing_and_Mathematical_Objection.pdf.

Piccinini, G. (2004a). The first computational theory of mind and brain: A close look at McCulloch and Pitts's 'Logical calculus of ideas immanent in nervous activity'. *Synthese 141*, 175–215. For a reply, see Aizawa 2010.

Piccinini, G. (2004b, September). Functionalism, computationalism, and mental contents. *Canadian Journal of Philosophy 34*(3), 375–410. http://www.umsl.edu/~piccininig/Functionalism_Computationalism_and_Mental_Contents.pdf.

Piccinini, G. (2005). Symbols, strings, and spikes: The empirical refutation of computationalism. Abstract at https://philpapers.org/rec/PICSSA; unpublished paper at https://web.archive.org/web/20080216023546/http://www.umsl.edu/~piccininig/Symbols%20Strings%20and%20Spikes%2019.htm; superseded by Piccinini and Bahar 2013.

Piccinini, G. (2006a). Computation without representation. *Philosophical Studies 137*(2), 204–241. http://www.umsl.edu/~piccininig/Computation_without_Representation.pdf.

Piccinini, G. (2006b, December). Computational explanation in neuroscience. *Synthese 153*(3), 343–353. http://www.umsl.edu/~piccininig/Computational%20Explanation%20in%20Neuroscience.pdf.

Piccinini, G. (2007a). Computational explanation and mechanistic explanation of mind. In M. Marraffa, M. D. Caro, and F. Ferretti (Eds.), *Cartographies of the Mind: Philosophy and Psychology in Intersection*, pp. 23–36. Dordrecht, The Netherlands: Springer.

Piccinini, G. (2007b, March). Computational modelling vs. computational explanation: Is everything a Turing machine, and does it matter to the philosophy of mind? *Australasian Journal of Philosophy 85*(1), 93–115. http://www.umsl.edu/~piccininig/Is_Everything_a_TM.pdf.

Piccinini, G. (2007c). Computationalism, the Church-Turing thesis, and the Church-Turing fallacy. *Synthese 154*, 97–120.

Piccinini, G. (2007d, October). Computing mechanisms. *Philosophy of Science 74*(4), 501–526.

Piccinini, G. (2008). Computers. *Pacific Philosophical Quarterly 89*, 32–73. http://www.umsl.edu/~piccininig/Computers.pdf.

Piccinini, G. (2009). Computationalism in the philosophy of mind. *Philosophy Compass 4*(3), 515–532. 10.1111/j.1747-9991.2009.00215.x.

Piccinini, G. (2010a, September). The mind as neural software? Understanding functionalism, computationalism, and computational functionalism. *Philosophy and Phenomenological Research 81*(2), 269–311.

Piccinini, G. (2010b, December). The resilience of computionalism. *Philosophy of Science 77*(5), 852–861.

Piccinini, G. (2011). The physical Church-Turing thesis: Modest or bold? *British Journal for the Philosophy of Science 62*, 733–769. http://www.umsl.edu/~piccininig/CT_Modest_or_Bold.pdf.

Piccinini, G. (2012). Computationalism. In E. Margolis, R. Samuels, and S. P. Stich (Eds.), *The Oxford Handbook of Philosophy of Cognitive Science*. Oxford University Press. http://www.umsl.edu/~piccininig/Computationalism.pdf.

Piccinini, G. (2015). *Physical Computation: A Mechanistic Account*. Oxford: Oxford University Press.

Piccinini, G. (2017). Computation in physical systems. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2017 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2017/entries/computation-physicalsystems/.

Piccinini, G. (2018, Spring). Computation and representation in cognitive neuroscience. *Minds and Machines 28*(1), 1–6. https://link.springer.com/content/pdf/10.1007%2Fs11023-018-9461-x.pdf.

Piccinini, G. and S. Bahar (2013). Neural computation and the computational theory of cognition. *Cognitive Science 34*, 453–488. http://www.umsl.edu/~piccininig/Neural_Computation_and_the_Computational_Theory_of_Cognition.pdf.

Piccinini, G. and C. Craver (2011). Integrating psychology and neuroscience: Functional analyses as mechanism sketches. *Synthese 183*(3), 283–311. http://philosophy.artsci.wustl.edu/files/philosophy/imce/integrating_psychology_and_neuroscience_functional_analyses_as_mechanism_sketches_0.pdf.

Piccinini, G. and A. Scarantino (2011). Information processing, computation, and cognition. *Journal of Biological Physics 37*, 1–38.

Pigliucci, M. (2014, 12 May). Neil deGrasse Tyson and the value of philosophy. *Scientia Salon*. http://scientiasalon.wordpress.com/2014/05/12/neil-degrasse-tyson-and-the-value-of-philosophy/.

Pigliucci, M. and M. Boudry (2013a, 10 October). The dangers of pseudoscience. *New York Times*. http://opinionator.blogs.nytimes.com/2013/10/10/the-dangers-of-pseudoscience/.

Pigliucci, M. and M. Boudry (Eds.) (2013b). *Philosophy of Pseudoscience: Reconsidering the Demarcation Problem*. Chicago: University of Chicago Press.

Pincock, C. (2011). Philosophy of mathematics. In J. Saatsi and S. French (Eds.), *Companion to the Philosophy of Science*, pp. 314–333. Continuum. http://pincock-yilmazer.com/chris/pincock%20philosophy%20of%20mathematics%20final%20draft.pdf.

Pinker, S. and R. Jackendoff (2005). The faculty of language: What's special about it? *Cognition 95*, 201–236. http://pinker.wjh.harvard.edu/articles/papers/2005_03_Pinker_Jackendoff.pdf.

Plaice, J. (1995, March). Computer science is an experimental science. *ACM Computing Surveys 27*(1), 33. https://www.cse.unsw.edu.au/~plaice/archive/JAP/P-ACMcs95-experimentalCS.pdf.

Plato (1961a). Phaedrus. In E. Hamilton and H. Cairns (Eds.), *The Collected Dialogues of Plato, including the Letters*. Princeton, NJ: Princeton University Press.

Plato (1961b). Republic. In E. Hamilton and H. Cairns (Eds.), *The Collected Dialogues of Plato, including the Letters*, pp. 575–844. Princeton, NJ: Princeton University Press.

Pleasant, J. C. (1989, March). The very idea (letter to the editor). *Communications of the ACM 32*(3), 374–375.

Polger, T. W. (2011). Are sensations still brain processes? *Philosophical Psychology 24*(1), 1–21.

Pollan, M. (2013, 23 & 30 December). The intelligent plant. *The New Yorker*, 92–105. http://www.newyorker.com/reporting/2013/12/23/131223fa_fact_pollan.

Pollock, J. L. (2008, March). What am i? virtual machines and the mind/body problem. *Philosophy and Phenomenological Research 76*(2), 237–309.

"PolR" (11 November 2009). An explanation of computation theory for lawyers. http://www.groklaw.net/article.php?story=20091111151305785; also at http://www.groklaw.net/staticpages/index.php?page=20091110152507492 and http://www.groklaw.net/pdf2/ComputationalTheoryforLawyers.pdf.

Polya, G. (1957). *How to Solve It: A New Aspect of Mathematical Method, 2nd edition*. Garden City, NY: Doubleday Anchor.

Popek, G. J. and R. P. Goldberg (1974, July). Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*(7), 412–421. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.4815&rep=rep1&type=pdf.

Popova, M. (2012). What is philosophy? an omnibus of definitions from prominent philosophers. http://www.brainpickings.org/index.php/2012/04/09/what-is-philosophy/.

Popper, K. (1959). *The Logic of Scientific Discovery*. New York: Harper & Row.

Popper, K. (1978). Three worlds. http://tannerlectures.utah.edu/_documents/a-to-z/p/popper80.pdf.

Post, E. L. (1941). Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation. In M. Davis (Ed.), *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, pp. 375–441. Boston: Birkhaäuser.

Post, E. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics 65*, 197–215.

Post, E. L. (1944). Recursively enumerable sets of positive integers. *Bulletin of the American Mathematical Society 5*, 284–316. https://projecteuclid.org/download/pdf_1/euclid.bams/1183505800.

Pour-El, M. B. (1974, November). Abstract computability and its relation to the general purpose analog computer (some connections between logic, differential equations and analog computers). *Transactions of the American Mathematical Society 199*, 1–28.

Powell, L. (2014, 8 May). An open letter to Neil deGrasse Tyson. *The Horseless Telegraph*. http://horselesstelegraph.wordpress.com/2014/05/08/an-open-letter-to-neil-degrasse-tyson/.

Powers, R. (1995). *Galatea 2.2: A Novel*. New York: Farrar Straus & Giroux. Excerpts at http://www.amazon.com/gp/reader/0312423136/ref=sib_dp_pt/104-4726624-5955928#reader-link and https://play.google.com/store/books/details/Richard_Powers_Galatea_2_2?id=9xCw4QPsy88C.

Prasse, M. and P. Rittgen (1998). Why Church's thesis still holds: Some notes on Peter Wegner's tracts on interaction and computability. *The Computer Journal 41*(6), 357–362. http://research.cs.queensu.ca/~akl/cisc879/papers/SELECTED_PAPERS_FROM_VARIOUS_SOURCES/Prasse.pdf.

Prescott, T. (2015, 21 March). Me in the machine. *New Scientist 225*(3013), 36–39. http://eprints.whiterose.ac.uk/95397/1/NewSci_preprint.pdf.

Press, L. (1993, September). Before the Altair: The history of personal computing. *Communications of the ACM 36*(9), 27–33. http://bpastudio.csudh.edu/fac/lpress/articles/hist.htm.

Preston, B. (2013). *A Philosophy of Material Culture: Action, Function, and Mind*. New York: Routledge.

Preston, J. (2012). Paul Feyerabend. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University. http://plato.stanford.edu/archives/win2012/entries/feyerabend/.

Preston, J. and M. Bishop (Eds.) (2002). *Views into the Chinese Room: New Essays on Searle and Artificial Intelligence*. Oxford: Clarendon Press.

Primiero, G. (2016). Information in the philosophy of computer science. In L. Floridi (Ed.), *The Routledge Handbook of Philosophy of Information*, pp. 90–106. London: Routledge. https://www.academia.edu/26800126/Information_in_the_Philosophy_of_Computer_Science.

Proudfoot, D. and B. J. Copeland (2012). Artificial Intelligence. In E. Margolis, R. Samuels, and S. P. Stich (Eds.), *The Oxford Handbook of Philosophy of Cognitive Science*. Oxford University Press.

Putnam, H. (1960). Minds and machines. In S. Hook (Ed.), *Dimensions of Mind: A Symposium*, pp. 148–179. New York: New York University Press.

Putnam, H. (1965, March). Trial and error predicates and the solution to a problem of Mostowski. *Journal of Symbolic Logic 30*(1), 49–57. http://www.ninagierasimczuk.com/flt2013/wp-content/uploads/2013/01/Putnam_1965.pdf.

Putnam, H. (1975). The meaning of 'meaning'. In K. Gunderson (Ed.), *Minnesota Studies in the Philosophy of Science, Vol. 7: Language, Mind, and Knowledge*, pp. 131–193. Minneapolis: University of Minnesota Press. Reprinted in Hilary Putnam, *Mind, Language and Reality* (Cambridge, UK: Cambridge University Press): 215–271; http://mcps.umn.edu/assets/pdf/7.3_Putnam.pdf.

Putnam, H. (1988). *Representation and Reality*. Cambridge, MA: MIT Press.

Putnam, H. (2015, 18 February). Rational reconstruction. *Sardonic Comment* blog, http://putnamphil.blogspot.com/2015/02/rational-reconstuction-in-1976-when.html.

Pylyshyn, Z. (2003, March). Return of the mental image: Are there really pictures in the brain? *Trends in Cognitive Sciences 7*(3), 113–118. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.294.3609&rep=rep1&type=pdf.

Pylyshyn, Z. W. (1973). What the mind's eye tells the mind's brain. *Psychological Bulletin 80*(1), 1–24. https://www.researchgate.net/publication/232553762_What_the_Mind%27s_Eye_Tells_the_Mind%27s_Brain_A_Critique_of_Mental_Imagery.

Pylyshyn, Z. W. (1984). *Computation and Cognition: Towards a Foundation for Cognitive Science*. Cambridge, MA: MIT Press. Ch. 3 ("The Relevance of Computation"), pp. 48–86, esp. §"The Role of Computer Implementation" (pp. 74–78).

Pylyshyn, Z. W. (1992). Computers and the symbolization of knowledge. In R. Morelli, W. M. Brown, D. Anselmi, K. Haberlandt, and D. Lloyd (Eds.), *Minds, Brains & Computers: Perspectives in Cognitive Science and Artificial Intelligence*, pp. 82–94. Norwood, NJ: Ablex. Page references are to 1987 preprint at http://ruccs.rutgers.edu/images/personal-zenon-pylyshyn/docs/suffolk.pdf.

qFiasco, F. (2018). Book review [of kasparov & greengard, *deep thinking*]. *Artificial Intelligence 260*, 36–41.

Qian, L. and E. Winfree (2011, 3 June). Scaling up digital circuit computation with DNA strand displacement cascades. *Science 332*, 1196–1201. Reviewed in Reif 2011.

Quillian, M. R. (1994, September). A content-independent explanation of science's effectivenss. *Philosohy of Science 61*(3), 429–448.

Quine, W. v. O. (1948). On what there is. *Review of Metaphysics 2*(5), 21–38. Reprinted in W.V.O. Quine, *From a Logical Point of View: 9 Logico-Philosophical Essays, Second Edition, revised* (Cambridge, MA: Harvard University Press, 1980): 1–19; http://math.boisestate.edu/~holmes/Phil209/Quine%20-%20On%20What%20There%20Is.pdf.

Quine, W. v. O. (1951). Two dogmas of empiricism. *Philosophical Review 60*, 20–43. Reprinted in W.V.O. Quine, *From a Logical Point of View: 9 Logico-Philosophical Essays, Second Edition, revised* (Cambridge, MA: Harvard University Press, 1980): 20–46; http://www.theologie.uzh.ch/dam/jcr:ffffffff-fbd6-1538-0000-000070cf64bc/Quine51.pdf.

Quine, W. v. O. (1976). Whither physical objects? In R. Cohen, P. Feyerabend, and M. Wartofsky (Eds.), *Essays in Memory of Imre Lakatos*, pp. 497–504. Dordrecht, Holland: D. Reidel.

Quine, W. v. O. (1987). *Quiddities: An Intermittently Philosophical Dictionary*. Cambridge, MA: Harvard University Press.

Quine, W. v. O. (1988, 4 May 2015). An unpublished letter from quine to hookway. In Hilary Putnam's *Sardonic Comment* blog, http://putnamphil.blogspot.com/2015/05/an-unpublished-letter-from-quine-to.html.

Radó, T. (1962, May). On non-computable functions. *The Bell System Technical Journal*, 877–884. http://alcatel-lucent.com/bstj/vol41-1962/articles/bstj41-3-877.pdf.

Rajagopalan, B. (2011?). Halting problem is solvable. (Humor), http://www.netfunny.com/rhf/jokes/89q4/halting.760.html.

Ralston, A. (1999). Let's abolish pencil-and-paper arithmetic. *Journal of Computers in Mathematics and Science Teaching 18*(2), 173–194. https://web.archive.org/web/20170719125406/http://tonyralston.com/papers/abolpub.htm.

Ramakrishna, P. (2019, 18 June). 'there's just no doubt that it will change the world': David Chalmers on V.R. and A.I. *New York Times/The Stone*. https://www.nytimes.com/2019/06/18/opinion/david-chalmers-virtual-reality.html.

Ramsey, F. P. (1929). Theories. In D. Mellor (Ed.), *Foundations: Essays in Philosophy, Logic, Mathematics and Economics; revised edition*, pp. 101–125. Atlantic Highlands, NJ: Humanities Press, 1978.

Randell, B. (1994). The origins of computer programming. *IEEE Annals of the History of Computing 16*(4), 6–14. https://www.researchgate.net/publication/3330487_The_origins_of_computer_programming/.

Rapaport, W. J. (1978, May). Meinongian theories and a Russellian paradox. *Noûs 12*(2), 153–180. https://www.cse.buffalo.edu//~rapaport/Papers/meinong-russell.pdf; errata, *No(u)s* 13 (1979): 125, https://www.cse.buffalo.edu//~rapaport/Papers/meinong-russell-errata.pdf.

Rapaport, W. J. (1982). Unsolvable problems and philosophical progress. *American Philosophical Quarterly 19*, 289–298. http://www.cse.buffalo.edu/~rapaport/Papers/apq.pdf.

Rapaport, W. J. (1984a, May/June). Can philosophy solve its own problems? *The [SUNY] News 13*, F2–F3. http://www.cse.buffalo.edu/~rapaport/Papers/rapaport84b.canphilsolve.pdf.

Rapaport, W. J. (1984b, Spring/Summer). Critical thinking and cognitive development. *American Philosophical Association Newsletter on Pre-College Instruction in Philosophy 1*, 4–5. Reprinted in *Proceedings and Addresses of the American Philosophical Association* 57(5) (May 1984): 610–615; http://www.cse.buffalo.edu/~rapaport/Papers/rapaport84-perryAPA.pdf.

Rapaport, W. J. (1985-1986b). Non-existent objects and epistemological ontology. *Grazer Philosophische Studien 25/26*, 61–95. http://www.cse.buffalo.edu/~rapaport/Papers/rapaport8586.pdf.

Rapaport, W. J. (1986a). Logical foundations for belief representation. *Cognitive Science 10*, 371–422. http://csjarchive.cogsci.rpi.edu/1986v10/i04/p0371p0422/MAIN.PDF.

Rapaport, W. J. (1986c, Summer). Philosophy, Artificial Intelligence, and the Chinese-room argument. *Abacus: The Magazine for the Computer Professional 3*, 6–17. Correspondence, *Abacus* 4 (Winter 1987): 6–7; 4 (Spring): 5–7; http://www.cse.buffalo.edu/~rapaport/Papers/abacus.pdf.

Rapaport, W. J. (1986d, June). Philosophy of Artificial Intelligence: A course outline. *Teaching Philosophy 9*(2), 103–120. http://www.cse.buffalo.edu/~rapaport/Papers/teachphil1986.pdf.

Rapaport, W. J. (1986e, September). Review of Deborah Johnson and John Snapper (eds.), *ethical issues in the use of computers*. *Teaching Philosophy 9*(3), 275–278. https://cse.buffalo.edu/~rapaport/Papers/johnson-snapper-review.pdf.

Rapaport, W. J. (1986f). Searle's experiments with thought. *Philosophy of Science 53*, 271–279. http://www.cse.buffalo.edu/~rapaport/Papers/philsci.pdf.

Rapaport, W. J. (1988a). Syntactic semantics: Foundations of computational natural-language understanding. In J. H. Fetzer (Ed.), *Aspects of Artificial Intelligence*, pp. 81–131. Dordrecht, The Netherlands: Kluwer Academic Publishers. http://www.cse.buffalo.edu/~rapaport/Papers/synsem.pdf; reprinted with numerous errors in Eric Dietrich (ed.) (1994), *Thinking Machines and Virtual Persons: Essays on the Intentionality of Machines* (San Diego: Academic Press): 225–273.

Rapaport, W. J. (1988b, December). To think or not to think: Review of john searle, *minds, brains and science*. *Noûs 22*(4), 585–609. http://www.cse.buffalo.edu/~rapaport/Papers/2Tor-2T.pdf.

Rapaport, W. J. (1990). Computer processes and virtual persons: Comments on Cole's 'Artificial Intelligence and personal identity'. Technical Report 90-13, SUNY Buffalo Department of Computer Science, Buffalo. http://www.cse.buffalo.edu/~rapaport/Papers/cole.tr.17my90.pdf.

Rapaport, W. J. (1992). Logic, propositional. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence, 2nd edition*, pp. 891–897. New York: John Wiley. http://www.cse.buffalo.edu/~rapaport/Papers/logic, propositional.pdf.

Rapaport, W. J. (1995). Understanding understanding: Syntactic semantics and computational cognition. In J. E. Tomberlin (Ed.), *AI, Connectionism, and Philosophical Psychology (Philosophical Perspectives, Vol. 9)*, pp. 49–88. Atascadero, CA: Ridgeview. http://www.cse.buffalo.edu/~rapaport/Papers/rapaport95-uu.pdf. Reprinted in Toribio, Josefa, & Clark, Andy (eds.) (1998), *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory (Artificial Intelligence and Cognitive Science: Conceptual Issues*, Vol. 4) (New York: Garland).

Rapaport, W. J. (1996). Understanding understanding: Semantics, computation, and cognition. Technical Report Technical Report 96-26, SUNY Buffalo Department of Computer Science, Buffalo. http://www.cse.buffalo.edu/~rapaport/Papers/book.pdf.

Rapaport, W. J. (1998). How minds can be computational systems. *Journal of Experimental and Theoretical Artificial Intelligence 10*, 403–419. http://www.cse.buffalo.edu/~rapaport/Papers/jetai-sspp98.pdf.

Rapaport, W. J. (1999). Implementation is semantic interpretation. *The Monist 82*, 109–130. http://www.cse.buffalo.edu/~rapaport/Papers/monist.pdf.

Rapaport, W. J. (2000a). Cognitive science. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, 4th edition*, pp. 227–233. New York: Grove's Dictionaries. http://www.cse.buffalo.edu/~rapaport/Papers/cogsci.pdf.

Rapaport, W. J. (2000b, October). How to pass a Turing test: Syntactic semantics, natural-language understanding, and first-person cognition. *Journal of Logic, Language, and Information 9*(4), 467–490. http://www.cse.buffalo.edu/~rapaport/Papers/TURING.pdf. Reprinted in Moor 2003, 161–184.

Rapaport, W. J. (2002). Holism, conceptual-role semantics, and syntactic semantics. *Minds and Machines 12*(1), 3–59. http://www.cse.buffalo.edu/~rapaport/Papers/crs.pdf.

Rapaport, W. J. (2003). What did you mean by that? Misunderstanding, negotiation, and syntactic semantics. *Minds and Machines 13*(3), 397–427. http://www.cse.buffalo.edu/~rapaport/Papers/negotiation-mandm.pdf.

Rapaport, W. J. (2005a). Castañeda, Hector-Neri. In J. R. Shook (Ed.), *The Dictionary of Modern American Philosophers, 1860–1960*, pp. 452–412. Bristol, UK: Thoemmes Press. http://www.cse.buffalo.edu/~rapaport/Papers/hncdict.tr.pdf.

Rapaport, W. J. (2005b, December). Implemention is semantic interpretation: Further thoughts. *Journal of Experimental and Theoretical Artificial Intelligence 17*(4), 385–417. https://www.cse.buffalo.edu//~rapaport/Papers/jetai05.pdf.

Rapaport, W. J. (2005c, December). Philosophy of computer science: An introductory course. *Teaching Philosophy 28*(4), 319–341. http://www.cse.buffalo.edu/~rapaport/philcs.html.

Rapaport, W. J. (2005d). Review of stuart m. shieber (ed.), *the turing test: verbal behavior as the hallmark of intelligence*. *Computational Linguistics 31*(3), 407–412. https://www.aclweb.org/anthology/J05-3006.

Rapaport, W. J. (2006a). How Helen Keller used syntactic semantics to escape from a Chinese room. *Minds and Machines 16*, 381–436. http://www.cse.buffalo.edu/~rapaport/Papers/helenkeller.pdf. See reply to comments, in Rapaport 2011b.

Rapaport, W. J. (2006b, March). Review of preston & bishop, *views into the chinese room*. *Australasian Journal of Philosophy 84*(1), 129–133. https://cse.buffalo.edu/~rapaport/Papers/crareview-ajp_129to133.pdf.

Rapaport, W. J. (2006c). The Turing test. In K. Brown (Ed.), *Encyclopedia of Language and Linguistics, 2nd Edition*, pp. Vol. 13, pp. 151–159. Oxford: Elsevier. http://www.cse.buffalo.edu/~rapaport/Papers/rapaport06-turingELL2.pdf.

Rapaport, W. J. (2007, Spring). Searle on brains as computers. *American Philosophical Association Newsletter on Philosophy and Computers 6*(2), 4–9. http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/v06n2Computers.pdf.

Rapaport, W. J. (2011a, December). A triage theory of grading: The good, the bad, and the middling. *Teaching Philosophy 34*(4), 347–372. http://www.cse.buffalo.edu/~rapaport/Papers/triage.pdf.

Rapaport, W. J. (2011b, Spring). Yes, she was! Reply to Ford's 'Helen Keller was never in a Chinese room'. *Minds and Machines 21*(1), 3–17. http://www.cse.buffalo.edu/~rapaport/Papers/Papers.by.Others/rapaport11-YesSheWas-MM.pdf.

Rapaport, W. J. (2012a). Intensionality vs. intentionality. http://www.cse.buffalo.edu/~rapaport/intensional.html.

Rapaport, W. J. (2012b, January-June). Semiotic systems, computers, and the mind: How cognition could be computing. *International Journal of Signs and Semiotic Systems 2*(1), 32–71. http://www.cse.buffalo.edu/~rapaport/Papers/Semiotic_Systems,_Computers,_and_the_Mind.pdf. Revised version published as Rapaport 2018b.

Rapaport, W. J. (2013). How to write. http://www.cse.buffalo.edu/~rapaport/howtowrite.html.

Rapaport, W. J. (2017a). On the relation of computing to the world. In T. M. Powers (Ed.), *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics*, pp. 29–64. Cham, Switzerland: Springer. Paper based on 2015 IACAP Covey Award talk; preprint at http://www.cse.buffalo.edu/~rapaport/Papers/rapaport4IACAP.pdf.

Rapaport, W. J. (2017b, Fall). Semantics as syntax. *American Philosophical Association Newsletter on Philosophy and Computers 17*(1), 2–11. http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/ComputersV17n1.pdf.

Rapaport, W. J. (2017c, Spring). What is computer science? *American Philosophical Association Newsletter on Philosophy and Computers 16*(2), 2–22. Paper based on 2017 APA Barwise Award talk; http://c.ymcdn.com/sites/www.apaonline.org/resource/collection/EADE8D52-8D02-4136-9A2A-729368501E43/ComputersV16n2.pdf.

Rapaport, W. J. (2018a). How to study: A brief guide. http://www.cse.buffalo.edu/~rapaport/howtostudy.html.

Rapaport, W. J. (2018b). Syntactic semantics and the proper treatment of computationalism. In M. Danesi (Ed.), *Empirical Research on Semiotics and Visual Rhetoric*, pp. 128–176. Hershey, PA: IGI Global. References on pp. 273–307; http://www.cse.buffalo.edu/~rapaport/Papers/SynSemProperTrtmtCompnlism.pdf. Revised version of Rapaport 2012b.

Rapaport, W. J. (2018c, Spring). What is a computer? a survey. *Minds and Machines 28*(3), 385–426. https://cse.buffalo.edu/~rapaport/Papers/rapaport2018-whatisacompr-MM.pdf.

Rapaport, W. J. and M. W. Kibby (2010). Contextual vocabulary acquisition: From algorithm to curriculum. http://www.cse.buffalo.edu/~rapaport/CVA/reading4CgSJnl.pdf.

Rapaport, W. J. and S. C. Shapiro (1984). Quasi-indexical reference in propositional semantic networks. In *Proceedings of the 10th International Conference on Computational Linguistics (COLING-84, Stanford University)*, pp. 65–70. Morristown, NJ: Association for Computational Linguistics. https://www.aclweb.org/anthology/P84-1016.

Rapaport, W. J., S. C. Shapiro, and J. M. Wiebe (1997). Quasi-indexicals and knowledge reports. *Cognitive Science 21*, 63–107. http://csjarchive.cogsci.rpi.edu/1997v21/i01/p0063p0107/MAIN.PDF.

Rees, R., C. Crawford, A. Bowyer, L. Schubert, and H. Loebner (1994). Forum: Notes on the turing test. *Communications of the ACM 37*(9), 13–15.

Reese, H. (18 February 2014a). The joy of teaching computer science in the age of facebook. *The Atlantic*, http://www.theatlantic.com/education/archive/2014/02/the-joy-of-teaching-computer-science-in-the-age-of-facebook/283879/.

Reese, H. (27 February 2014b). Why study philosophy? 'to challenge your own point of view'. *The Atlantic*, http://www.theatlantic.com/education/archive/2014/02/why-study-philosophy-to-challenge-your-own-point-of-view/283954/.

Reif, J. H. (2011, 3 June). Scaling up DNA computation. *Science 332*, 1156–1157. Review of Qian and Winfree 2011.

Rendell, P. (2000). This is a Turing machine implemented in Conway's Game of Life. http://rendell-attic.org/gol/tm.htm.

Rendell, P. (2001). A Turing machine in Conway's Game of Life. https://www.ics.uci.edu/~welling/teaching/271fall09/Turing-Machine-Life.pdf.

Rendell, P. (2010). This is a universal Turing machine (utm) implemented in Conway's Game of Life. http://rendell-attic.org/gol/utm/.

Rescher, N. (1985). *The Strife of Systems: An Essay on the Grounds and Implications of Philosophical Diversity*. Pittsburgh: University of Pittsburgh Press.

Rescorla, M. (2007). Church's thesis and the conceptual analysis of computability. *Notre Dame Journal of Formal Logic 48*(2), 253–280. http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/church2.pdf.

Rescorla, M. (2012a, December). Are computational transitions sensitive to semantics? *Australian Journal of Philosophy 90*(4), 703–721. http://www.philosophy.ucsb.edu/docs/faculty/papers/formal.pdf.

Rescorla, M. (2012b, January-March). How to integrate representation into computational modeling, and why we should. *Journal of Cognitive Science (South Korea) 13*(1), 1–38. http://cogsci.snu.ac.kr/jcs/issue/vol13/no1/01+Michael+Rescorla.pdf.

Rescorla, M. (2013, December). Against structuralist theories of computational implementation. *British Journal for the Philosophy of Science 64*(4), 681–707. http://philosophy.ucsb.edu/docs/faculty/papers/against.pdf.

Rescorla, M. (2014a, January). The causal relevance of content to computation. *Philosophy and Phenomenological Research 88*(1), 173–208. http://www.philosophy.ucsb.edu/people/profiles/faculty/cvs/papers/causalfinal.pdf.

Rescorla, M. (2014b). A theory of computational implementation. *Synthese 191*, 1277–1307. http://philosophy.ucsb.edu/docs/faculty/papers/implementationfinal.pdf.

Rescorla, M. (2015a). The computational theory of mind. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2015 ed.). http://plato.stanford.edu/archives/win2015/entries/computational-mind/.

Rescorla, M. (2015b). The representational foundations of computation. *Philosophia Mathematica 23*(3), 338–366. http://www.philosophy.ucsb.edu/docs/faculty/michael-rescorla/representational-foundations.pdf.

Rey, G. (2012, 2nd Quarter). The Turing thesis vs. the Turing test. *The Philosopher's Magazine 57*, 84–89.

Richards, R. J. (2009, September-October). The descent of man. *American Scientist 97*(5), 415–417. https://www.americanscientist.org/article/the-descent-of-man.

Richards, W. (1988). *Natural Computation*. Cambridge, MA: MIT Press.

Roberts, E. S. (2006). *The Art and Science of Java*. Stanford, CA: Stanford University. http://people.reed.edu/~jerry/121/materials/artsciencejava.pdf.

Roberts, P. and J. Knobe (2016). Interview on experimental philosophy with Joshua Knobe. *Exchanges: The Warwick Research Journal 4*(1), 14–28. http://exchanges.warwick.ac.uk/index.php/exchanges/article/view/128.

Roberts, S. (2018, 17 December). The yoda of silicon valley. *New York Times*. https://www.nytimes.com/2018/12/17/science/donald-knuth-computers-algorithms-programming.html.

Robertson, D. S. (2003). *Phase Change: The Computer Revolution in Science and Mathematics*. Oxford University Press.

Robertson, J. I. (1979, June-July). How to do arithmetic. *American Mathematical Monthly 86*, 431–439.

Robinson, J. A. (1994). Logic, computers, Turing, and von Neumann. In K. Furukawa, D. Michie, and S. Muggleton (Eds.), *Machine Intelligence 13: Machine Intelligence and Inductive Learning*, pp. 1–35. Oxford: Clarendon Press. http://staffweb.worc.ac.uk/DrC/Courses%202013-14/COMP3202/Reading%20Materials/robinson94.pdf.

Rogers, Jr., H. (1959). The present theory of turing machine computability. *Journal of the Society for Industrial and Applied Mathematics 7*(1), 114–130.

Rogers, Jr., H. (1967). *Theory of Recursive Functions and Effective Computability*. New York: McGraw-Hill.

Rohlf, M. (2010). Immanuel Kant. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Fall 2010 Edition)*. Metaphysics Research Lab, Stanford University. http://plato.stanford.edu/archives/fall2010/entries/kant/.

Romanycia, M. H. and F. J. Pelletier (1985). What is a heuristic? *Computational Intelligence 1*(2), 47–58. http://www.sfu.ca/~jeffpell/papers/RomanyciaPelletierHeuristics85.pdf.

Rosch, E. (1978). Principles of categorization. In E. Rosch and B. B. Lloyd (Eds.), *Cognition and Categorization*, pp. 27–48. Hillsdale, NJ: Lawrence Erlbaum Associates.

Rosch, E. and C. B. Mervis (1975). Family resemblances: Studies in the internal structure of categories. *Cognitive Psychology 7*, 573–605.

Rosenberg, A. (1994). If economics isn't science, what is it? In D. M. Hausman (Ed.), *The Philosophy of Economics: An Anthology, Second Edition*, pp. 376–394. New York: Cambridge University Press. http://philoscience.unibe.ch/documents/kursarchiv/SS07/Rosenberg1994.pdf.

Rosenbloom, P. S. (2010, December). What is computation? computing and computation. *Ubiquity 2010*(December). Article 1, http://ubiquity.acm.org/article.cfm?id=1897729.

Rosenblueth, A. and N. Wiener (1945). The role of models in science. *Philosophy of Science 12*, 316–321.

Ross, D. (1974, 5 September). Church's thesis: What its difficulties are and are not. *Journal of Philosophy 71*(15), 515–525.

Rosser, J. B. (1939, June). An informal exposition of proofs of Gödel's theorems and Church's theorem. *Journal of Symbolic Logic 4*(2), 53–60.

Rosser, J. B. (1978). *Logic for Mathematicians: Second Edition*. Mineola, NY: Dover Publications. First edition (1953) at https://archive.org/details/logicformathemat00ross.

Rothman, J. (2016, 9 June). What are the odds we are living in a computer simulation? *New Yorker (online)*. https://www.newyorker.com/books/joshua-rothman/what-are-the-odds-we-are-living-in-a-computer-simulation.

Royce, J. (1900). *The World and the Individual*. London: Macmillan. https://archive.org/details/worldindividual00royciala.

Rubinoff, M. (1953, October). Analogue vs. digital computers—a comparison. *Proceedings of the IRE 41*(10), 1254–1262.

Rudin, W. (1964). *Principles of Mathematical Analysis, Second Edition*. New York: McGraw-Hill.

Ruff, C. (2016, 5 February). Computer science, meet humanities: In new majors, opposites attract. *Chronicle of Higher Education 62*(21), A19. http://chronicle.com/article/Computer-Science-Meet/235075.

Rupp, N. A. (9 August 2003). Computer science is philosophy. https://web.archive.org/web/20080906232002/https://weblogs.java.net/blog/n_alex/archive/2003/08/computer_scienc.html.

Russell, B. (1912). *The Problems of Philosophy*. London: Oxford University Press (1959).

Russell, B. (1917). *Mysticism and Logic and Other Essays*. London: George Allen & Unwin. http://archive.org/details/mysticism00russuoft.

Russell, B. (1927). *The Analysis of Matter*. Nottingham, UK: Spokesman (2007). https://archive.org/details/in.ernet.dli.2015.221533.

Russell, B. (1936). The limits of empiricism. *Proceedings of the Aristotelian Society, New Series 36*, 131–150.

Russell, S. (1995). Rationality and intelligence. *Artificial Intelligence 94*, 57–77. http://www.cs.berkeley.edu/~russell/papers/aij-cnt.pdf.

Russell, S. J. and P. Norvig (2003). *Artificial Intelligence: A Modern Approach; Second Edition*. Upper Saddle River, NJ: Pearson Education.

Russo, J. (1986). Saturn's rings: What GM's Saturn project is really about. *Cornell University Labor Research Review 1*(9). http://digitalcommons.ilr.cornell.edu/cgi/viewcontent.cgi?article=1084&context=lrr.

Rutenberg, J. (2019, 27 January). The tabloid myths of Jennifer Aniston and Donald Trump. *New York Times*. https://www.nytimes.com/2019/01/27/business/media/jennifer-aniston-donald-trump-tabloid-media-pregnancy.html.

Ryan, B. (1991, February). Dynabook revisited with Alan Kay. *Byte 16*(2), 203–204, 206–208.

Ryle, G. (1945). Knowing how and knowing that. *Proceedings of the Aristotelian Society, New Series 46*, 1–16. http://www.informationphilosopher.com/solutions/philosophers/ryle/Ryle_KnowHow.pdf.

Sackur, Jérôme, S. and S. Dehaene (2009). The cognitive architecture for chaining of two mental operations. *Cognition 111*, 187–211. http://www.unicog.org/publications/SackurDehaene_ChainingOfArithmeticOperations_Cognition2009.pdf.

Sale, T. (nd). The Colossus rebuild project. http://www.codesandciphers.org.uk/lorenz/rebuild.htm.

Salter, J. (2010, 14 January). The art of the ditch. *New York Review of Books 57*(1). http://www.nybooks.com/articles/archives/2010/jan/14/the-art-of-the-ditch/.

Samet, J. and D. Zaitchik (2017). Innateness and contemporary theories of cognition. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2017 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2017/entries/innateness-cognition/.

Samuel, A. L. (1953, October). Computing bit by bit, or digital computers made easy. *Proceedings of the IRE 41*(10), 1223–1230.

Samuelson, P. (1988, September). Is copyright law steering the right course? *IEEE Software*, 78–86.

Samuelson, P. (1989, May). Why the look and feel of software user interfaces should not be protected by copyright law. *Communications of the ACM 32*(5), 563–572. http://www.foo.be/andria/docs/p563-samuelson.pdf.

Samuelson, P. (1990, August). Should program algorithms be patented? *Communications of the ACM 33*(8), 23–27. https://www.law.berkeley.edu/php-programs/faculty/facultyPubsPDF.php?facID=346&pubID=94.

Samuelson, P. (1991, October). Digital media and the law. *Communications of the ACM 34*(10), 23–28.

Samuelson, P. (2003, October). Unsolicited communications as trespass? *Communications of the ACM 46*(10), 15–20. http://people.ischool.berkeley.edu/~pam/papers/acm_vol46_p15.pdf.

Samuelson, P. (2007a, October). Does copyright law need to be reformed? *Communications of the ACM 50*(10), 19–23.

Samuelson, P. (2007b, June). Software patents and the metaphysics of section 271(f). *Communications of the ACM 50*(6), 15–19. https://www.law.berkeley.edu/php-programs/faculty/facultyPubsPDF.php?facID=346&pubID=187.

Samuelson, P. (2008, July). Revisiting patentable subject matter. *Communications of the ACM 51*(7), 20–22.

Samuelson, P. (2013, November). Is software patentable? *Communications of the ACM 56*(11), 23–25. https://www.law.berkeley.edu/php-programs/faculty/facultyPubsPDF.php?facID=346&pubID=257.

Samuelson, P. (2015, March). Copyrightability of java apis revisited. *Communications of the ACM 58*(3), 22–24. http://radar.oreilly.com/2014/11/copyrightability-of-java-apis-revisited.html.

Samuelson, P., R. Davis, M. D. Kapor, and J. Reichman (1994, December). A manifesto concerning the legal protection of computer programs. *Columbia Law Review 94*(8), 2308–2431. http://scholarship.law.duke.edu/cgi/viewcontent.cgi?article=1783&context=faculty_scholarship.

Sanford, N. (1967). *Where Colleges Fail*. San Francisco: Jossey-Bass.

Saussure, F. d. (1959). *Course in General Linguistics*. New York: Philosophical Library. Charles Bally, Albert Sechehaye, & Albert Reidlinger (eds.).

Savage, N. (2016, March). When computers stand in the schoolhouse door. *Communications of the ACM 59*(3), 19–21. http://delivery.acm.org/10.1145/2880000/2875029/p19-savage.pdf.

Sayre, K. M. (1986, March). Intentionality and information processing: An alternative model for cognitive science. *Behavioral and Brain Sciences 9*(1), 121–165.

Scarantino, A. and G. Piccinini (2010, April). Information without truth. *Metaphilosophy 41*(3), 313–330.

Schächter, V. (1999, January). How does concurrency extend the paradigm of computation? *The Monist 82*(1), 37–57.

Schagrin, M. L., W. J. Rapaport, and R. R. Dipert (1985). *Logic: A Computer Approach*. New York: McGraw-Hill.

Schank, R. C. (1983, Winter-Spring). The current state of AI: One man's opinion. *AI Magazine 4*(1), 3–8.

Scherlis, W. L. and D. S. Scott (1983). First steps towards inferential programming. In R. Mason (Ed.), *Information Processing 83*, pp. 199–212. JFIP and Elsevier North-Holland. http://repository.cmu.edu/cgi/viewcontent.cgi?article=3542&context=compsci; reprinted in Colburn et al. 1993, pp. 99–133.

Scheutz, M. (1998). Implementation: Computationalism's weak spot. *Conceptus Journal of Philosophy 31*(79), 229–239. http://hrilab.tufts.edu/publications/scheutz98conceptus.pdf.

Scheutz, M. (1999). When physical systems realize functions. *Minds and Machines 9*, 161–196. http://hrilab.tufts.edu/publications/scheutz99mm.pdf.

Scheutz, M. (2001). Computational versus causal complexity. *Minds and Machines 11*, 543–566. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.8293&rep=rep1&type=pdf.

Scheutz, M. (2012, January-March). What it is *not* to implement a computation: A critical analysis of Chalmers' notion of implemention. *Journal of Cognitive Science (South Korea) 13*(1), 75–106. http://cogsci.snu.ac.kr/jcs/issue/vol13/no1/04_Matthias_Scheutz.pdf.

Schmidhuber, J. (2002). Zuse's thesis: The universe is a computer. http://www.idsia.ch/~juergen/digitalphysics.html.

Schneider, D. (2007, May-June). Neatness counts: Messy computer code is normally frowned on—but not always. *Amerian Scientist*, 213–214. https://web.archive.org/web/20160628161101/http://www.americanscientist.org/issues/pub/neatness-counts.

Schulman, A. N. (2009, Winter). Why minds are not like computers. *The New Atlantis*, 46–68. http://www.thenewatlantis.com/publications/why-minds-are-not-like-computers.

Schweizer, P. (2017). Cognitive computation *sans* representation. In T. Powers (Ed.), *Philosophy and Computing: Essays in Epistemology, Philosophy of Mind, Logic, and Ethics,*. Springer. http://www.research.ed.ac.uk/portal/files/29364320/Schweizer_IACAP15_1.pdf.

Schwitzgebel, E. (2015). Belief. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2015 ed.). Stanford University. https://plato.stanford.edu/archives/sum2015/entries/belief/.

Schwitzgebel, E. (9 January 2012). For all x, there's philosophy of x. http://schwitzsplinters.blogspot.com/2012/01/for-all-x-theres-philosophy-of-x.html.

Scott, J. and A. Bundy (2015, December). Creating a new generation of computational thinkers. *Communications of the ACM 58*(12), 37–40.

Seabrook, J. (2007, 14 May). Fragmentary knowledge. *The New Yorker*, 94–102.

Searle, J. R. (1969). *Speech Acts: An Essay in the Philosophy of Language*. Cambridge, UK: Cambridge University Press.

Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and Brain Sciences 3*, 417–457.

Searle, J. R. (1982, 29 April). The myth of the computer. *New York Review of Books*, 3–6. See correspondence, same journal, 24 June 1982, pp. 56–57.

Searle, J. R. (1983). *Intentionality: An Essay in the Philosophy of Mind*. Cambridge, UK: Cambridge University Press.

Searle, J. R. (1984). *Minds, Brains and Science*. Cambridge, MA: Harvard University Press.

Searle, J. R. (1990, November). Is the brain a digital computer? *Proceedings and Addresses of the American Philosophical Association 64*(3), 21–37. Reprinted in slightly revised form as Searle 1992, Ch. 9.

Searle, J. R. (1992). *The Rediscovery of the Mind*. Cambridge, MA: MIT Press.

Searle, J. R. (1995). *The Construction of Social Reality*. New York: Free Press.

Sellars, W. (1963). Philosophy and the scientific image of man. In *Science, Perception and Reality*, pp. 1–40. London: Routledge & Kegan Paul. http://www.ditext.com/sellars/psim.html.

Shagrir, O. (1999). What is computer science about? *The Monist 82*(1), 131–149.

Shagrir, O. (2001, April). Content, computation and externalism. *Mind 110*(438), 369–400. http://moon.cc.huji.ac.il/oron-shagrir/papers/Content_Computation_and_Externalism.pdf.

Shagrir, O. (2002, May). Effective computation by humans and machines. *Minds and Machines 12*(2), 221–240.

Shagrir, O. (2006). Gödel on Turing on computability. In A. Olszewski, J. Wołenski, and R. Janusz (Eds.), *Church's Thesis after 70 Years*, pp. 393–419. Ontos-Verlag. Page references to online version at http://edelstein.huji.ac.il/staff/shagrir/papers/Goedel_on_Turing_on_Computability.pdf.

Shagrir, O. (2012a, April-June). Can a brain possess two minds? *Journal of Cognitive Science (South Korea) 13*(2), 145–165. http://cogsci.snu.ac.kr/jcs/issue/vol13/no2/02Oron+Shagrir.pdf.

Shagrir, O. (2012b, Summer). Computation, implementation, cognition. *Minds and Machines 22*(2), 137–148.

Shagrir, O. (2012c). Structural representations and the brain. *British Journal for the Philosophy of Science 63*, 519–545. Preprint at https://oronshagrir.huji.ac.il/sites/default/files/oronshagrir/files/structural_representations_and_the_brain.pdf.

Shagrir, O. (2017, July). Review of Piccinini 2015. *Philosophy of Science 84*(3), 604–612. Preprint at https://oronshagrir.huji.ac.il/sites/default/files/oronshagrir/files/review_of_physical_computation_a_mechanistic_account.pdf.

Shagrir, O. (2018a, Spring). The brain as an input-output model of the world. *Minds & Machines 28*(1), 53–75. http://oronshagrir.huji.ac.il/sites/default/files/oronshagrir/files/the_brain_as_an_input-output_model_of_the_world.pdf.

Shagrir, O. (2018b). In defense of the semantic view of computation. *Synthese*. https://doi.org/10.1007/s11229-018-01921-z.

Shagrir, O. and W. Bechtel (2015). Marr's computational-level theories and delineating phenomena. In D. Kaplan (Ed.), *Integrating Psychology and Neuroscience: Prospects and Problems*. Oxford: Oxford University Press. http://philsci-archive.pitt.edu/11224/1/shagrir_and_bechtel.Marr's_Computational_Level_and_Delineating_Phenomena.pdf.

Shannon, C. E. (1937). A symbolic analysis of relay and switching circuits. Technical report, MIT Department of Electrical Engineering, Cambridge, MA. MS thesis, 1940; http://hdl.handle.net/1721.1/11173.

Shannon, C. E. (1948, July and October). A mathematical theory of communication. *The Bell System Technical Journal 27*, 379–423, 623–656. http://worrydream.com/refs/Shannon%20-%20A%20Mathematical%20Theory%20of%20Communication.pdf.

Shannon, C. E. (1950, March). Programming a computer for playing chess. *Philosophical Magazine, Ser. 7 41*(314). http://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf.

Shannon, C. E. (1953, October). Computers and automata. *Proceedings of the Institute of Radio Engineers 41*(10), 1234–1241. http://tinyurl.com/yalw2rx3.

Shapiro, E. and Y. Benenson (2006, May). Bringing DNA computers to life. *Scientific American 294*(5), 44–51. http://www.wisdom.weizmann.ac.il/~udi/papers/ShapiroBenensonMay06.pdf.

Shapiro, F. R. (1985, 24 March). Debugging etymologies. *New York Times*. letter to the editor, http://www.nytimes.com/1985/03/24/books/l-debugging-etymologies-114833.html.

Shapiro, F. R. (2000, April-June). Origin of the term *software*: Evidence from the JSTOR electronic journal archive. *IEEE Annals of the History of Computing 22*(2), 69–71.

Shapiro, S. (1983). Remarks on the development of computability. *History and Philosophy of Logic 4*(1), 203–220.

Shapiro, S. (1993). Understanding Church's thesis, again. *Acta Analytica 11*, 59–77.

Shapiro, S. (2009, March). We hold these truths to be self-evident: But what do we mean by that? *Review of Symbolic Logic 2*(1), 175–207. http://www.pgrim.org/philosophersannual/29articles/shapirowehold.pdf.

Shapiro, S. (2013). The open texture of computability. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 153–181. Cambridge, MA: MIT Press.

Shapiro, S. C. (1977). Representing numbers in semantic networks: Prolegomena. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pp. 284. Los Altos, CA: Morgan Kaufmann. http://www.cse.buffalo.edu/~shapiro/Papers/sha77b.

Shapiro, S. C. (1989). The Cassie projects: An approach to natural language competence. In J. Martins and E. Morgado (Eds.), *EPIA 89: 4th Portugese Conference on Artificial Intelligence Proceedings*, pp. 362–380. Berlin: Springer-Verlag Lecture Notes in Artificial Intelligence 390. http://www.cse.buffalo.edu/sneps/epia89.pdf.

Shapiro, S. C. (1992a). Artificial Intelligence. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence, 2nd edition*, pp. 54–57. New York: John Wiley & Sons. https://www.cse.buffalo.edu/~shapiro/Papers/ai.pdf. Revised version in Anthony Ralston, Edwin D. Reilly & David Hemmendinger (eds.), *Encyclopedia of Computer Science, 4th Edition* (New York: Grove's Dictionaries, 1993): 89–93, https://www.cse.buffalo.edu/~shapiro/Papers/ai-eofcs.pdf.

Shapiro, S. C. (1992b). *Common Lisp: An Interactive Approach*. New York: W.H. Freeman. https://www.cse.buffalo.edu/~shapiro/Commonlisp/.

Shapiro, S. C. (1995, November). Computationalism. *Minds and Machines 5*(4), 517–524. http://www.cse.buffalo.edu/~shapiro/Papers/computationalism.pdf.

Shapiro, S. C. (1998). Embodied Cassie. In *Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium*, pp. 136–143. Menlo Park, CA: AAAI Press. Technical Report FS-98-02, http://www.cse.buffalo.edu/~shapiro/Papers/embodiedcassie.pdf.

Shapiro, S. C. (2001). Computer science: The study of procedures. Technical report, Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY. http://www.cse.buffalo.edu/~shapiro/Papers/whatiscs.pdf; see also http://www.cse.buffalo.edu/~shapiro/Courses/CSE115/notes2.html.

Shapiro, S. C. and S. C. Kwasny (1975, August). Interactive consulting via natural language. *Communications of the ACM 18*(8), 459–462. http://www.cse.buffalo.edu/~shapiro/Papers/shakwa75.pdf.

Shapiro, S. C. and W. J. Rapaport (1987). SNePS considered as a fully intensional propositional semantic network. In N. Cercone and G. McCalla (Eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge*, pp. 262–315. New York: Springer-Verlag. https://www.cse.buffalo.edu/~rapaport/676/F01/shapiro.rapaport.87.pdf.

Shapiro, S. C. and W. J. Rapaport (1991). Models and minds: Knowledge representation for natural-language competence. In R. Cummins and J. Pollock (Eds.), *Philosophy and AI: Essays at the Interface*, pp. 215–259. Cambridge, MA: MIT Press. http://www.cse.buffalo.edu/~rapaport/Papers/mandm.tr.pdf.

Shapiro, S. C. and W. J. Rapaport (1995). An introduction to a computational reader of narratives. In J. F. Duchan, G. A. Bruder, and L. E. Hewitt (Eds.), *Deixis in Narrative: A Cognitive Science Perspective*, pp. 79–105. Hillsdale, NJ: Lawrence Erlbaum Associates. http://www.cse.buffalo.edu/~rapaport/Papers/shapiro.rapaport.95.pdf.

Shapiro, S. C., S. Srihari, and B. Jayaraman (December 1992). email discussion. http://www.cse.buffalo.edu/~rapaport/scs.txt.

Shapiro, S. C. and M. Wand (1976, November). The relevance of relevance. Technical Report 46, Indiana University Computer Science Department, Bloomington, IN. http://www.cs.indiana.edu/pub/techreports/TR46.pdf.

Shaw, A., V. Li, and K. R. Olson (2012, November-December). Children apply principles of physical ownership to ideas. *Cognitive Science 36*(8), 1383–1403. https://depts.washington.edu/uwkids/Shaw.Li.Olson.2012.pdf.

Shelley, M. W. (1818). *Frankenstein; or, the Modern Prometheus*. http://www.literature.org/authors/shelley-mary/frankenstein/.

Shepherdson, J. and H. Sturgis (1963, April). Computability of recursive functions. *Journal of the ACM 10*(2), 217–255.

Sheraton, M. (1981, 2 May). The elusive art of writing precise recipes. *New York Times*. http://www.nytimes.com/1981/05/02/style/de-gustibus-the-elusive-art-of-writing-precise-recipes.html.

Shieber, S. M. (1994a). Lessons from a restricted turing test. *Communications of the ACM 37*(6), 70–78.

Shieber, S. M. (1994b). On loebner's lessons. *Communications of the ACM 37*(6), 83–84.

Shieber, S. M. (Ed.) (2004). *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*. Cambridge, MA: MIT Press.

Shieber, S. M. (2007, December). The Turig Test as interactive proof. *Noûs 41*(4), 686–713. https://dash.harvard.edu/bitstream/handle/1/2027203/turing-interactive-proof.pdf.

Shieh, D. and S. Turkle (2009, 27 March). The trouble with computer simulations: Linked in with Sherry Turkle. *Chronicle of Higher Education*, A14. http://chronicle.com/article/The-Trouble-With-Computer/5386.

Shladover, S. E. (2016, June). What "self-driving" cars will really look like. *Scientific American*. http://endofdriving.org/wp-content/uploads/2016/10/What_Self-Driving_Cars_Will_Really_Look_Like-ShladoverScientificAmericanJune2016.pdf.

Shneiderman, B. (2007, June). Web science: A provocative invitation to computer science. *Communications of the ACM 50*(6), 25–27. https://www.academia.edu/2900348/Viewpoint-Web_Science_A_Provocative_Invitation_to_Computer_Science?auto=download.

Shoenfield, J. R. (1967). *Mathematical Logic*. Reading, MA: Addison-Wesley. https://www.karlin.mff.cuni.cz/~krajicek/shoenfield.pdf.

Shoham, Y. (2016, January). Why knowledge representation matters. *Communications of the ACM 59*(1), 47–49. http://cacm.acm.org/magazines/2016/1/195730-why-knowledge-representation-matters/fulltext.

Shustek, L. (2009, March). An interview with C.A.R. Hoare. *Communications of the ACM 52*(3), 38–41.

Sieg, W. (1994). Mechanical procedures and mathematical experience. In A. George (Ed.), *Mathematics and Mind*, pp. 71–117. New York: Oxford University Press. http://repository.cmu.edu/cgi/viewcontent.cgi?article=1248&context=philosophy.

Sieg, W. (1997, June). Step by recursive step: Church's analysis of effective calculability. *Bulletin of Symbolic Logic 3*(2), 154–180.

Sieg, W. (2000, 28 February). Calculations by man and machine: Conceptual analysis. Technical Report CMU-PHIL-104, Carnegie-Mellon University Department of Philosophy, Pittsburgh, PA. http://repository.cmu.edu/philosophy/178.

Sieg, W. (2006, June). Gödel on computability. *Philosophia Mathematica 14*, 189–207. Page references to preprint at http://repository.cmu.edu/cgi/viewcontent.cgi?article=1112&context=philosophy.

Sieg, W. (2007, June). On mind & Turing's machines. *Natural Computing 6*(2), 187–205. https://www.cmu.edu/dietrich/philosophy/docs/seig/OnmindTuringsMachines.pdf.

Sieg, W. (2008). Church without dogma: Axioms for computability. In S. Cooper, B. Löwe, and A. Sorbi (Eds.), *New Computational Paradigms: Changing Conceptions of What Is Computable*, pp. 139–152. Springer. https://www.cmu.edu/dietrich/philosophy/docs/tech-reports/175_Sieg.pdf.

Sieg, W. and J. Byrnes (1999, January). An abstract model for parallel computation: Gandy's thesis. *The Monist 82*(1), 150–164.

Siegelmann, H. T. (1995, 28 April). Computation beyond the Turing limit. *Science 268*(5210), 545–548. https://binds.cs.umass.edu/papers/1995_Siegelmann_Science.pdf.

Silver, D. (2016, September-October). Mathematical induction and the nature of british miracles. *American Scientist 104*(5), 296ff.

Silver, D. et al. (2016, 28 January). Mastering the game of Go with deep neural networks and tree search. *Nature 539*, 484–489.

Silver, D. et al. (2018, 7 December). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science 362*(6419), 1140–1144.

Simon, H. A. (1947). *Administrative Behavior*. New York‘: Macmillan.

Simon, H. A. (1959, June). Theories of decision-making in economics and behavioral science. *American Economic Review 49*(3), 253–283.

Simon, H. A. (1962, 12 December). The architecture of complexity. *Proceedings of the American Philosophical Society 106*(6), 467–482. https://www.cc.gatech.edu/classes/AY2013/cs7601_spring/papers/Simon-Complexity.pdf; reprinted as Simon 1996a, Ch. 8.

Simon, H. A. (1966). Thinking by computers. In R. G. Colodny (Ed.), *Mind and Cosmos: Essays in Contemporary Science and Philosophy*, pp. 3–21. Pittsburgh: University of Pittsburgh Press. http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=39897.

Simon, H. A. (1977, 18 March). What computers mean for man and society. *Science 195*(4283), 1186–1191.

Simon, H. A. (1978). Rational decision-making in business organizations. In A. Lindbeck (Ed.), *Nobel Lectures, Economics 1969–1980*, pp. 343–371. Singapore: World Scientific, 1992. http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/1978/simon-lecture.pdf. Also, *American Economic Review* 69(4) (1979): 493–513.

Simon, H. A. (1996a). Computational theories of cognition. In W. O'Donohue and R. F. Kitchener (Eds.), *The Philosophy of Psychology*, pp. 160–172. London: SAGE Publications.

Simon, H. A. (1996b). *The Sciences of the Artificial, Third Edition*. Cambridge, MA: MIT Press.

Simon, H. A. and A. Newell (1956). Models: Their uses and limitations. In L. D. White (Ed.), *The State of the Social Sciences*, pp. 66–83. Chicago: University of Chicago Press. http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=356856.

Simon, H. A. and A. Newell (1958, January-February). Heuristic problem solving: The next advance in operations research. *Operations Research 6*(1), 1–10.

Simon, H. A. and A. Newell (1962). Simulation of human thinking. In M. Greenberger (Ed.), *Computers and the World of the Future*, pp. 94–114. Cambridge, MA: MIT Press. http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=33601.

Simon, T. W. (1990). Artificial methodology meets philosophy. In D. Partridge and Y. Wilks (Eds.), *The Foundations of Artificial Intelligence: A Sourcebook*, pp. 155–164. Cambridge, UK: Cambridge University Press.

Simonite, T. (2009, 29 August). Soap bubbles to take the drag out of future cars. *New Scientist*. http://www.newscientist.com/article/dn17706-soap-bubbles-to-take-the-drag-out-of-future-cars.html.

Simonite, T. (2017, 18 October). AI experts want to end 'black box' algorithms in government. *Wired*. https://www.wired.com/story/ai-experts-want-to-end-black-box-algorithms-in-government/.

Skidelsky, R. (2014, 3 April). The programmed prospect before us. *New York Review of Books 61*(6), 35–37. http://www.skidelskyr.com/site/article/the-programmed-prospect-before-us/.

Skinner, D. (2006, Spring). The age of female computers. *The New Atlantis*, 96–103. http://www.thenewatlantis.com/publications/the-age-of-female-computers and http://www.thenewatlantis.com/docLib/TNA12-Skinner.pdf. Review of Grier 2005.

Skow, B. (2007). Are shapes intrinsic? *Philosophical Studies 133*, 111–130.

Slagle, J. R. (1971). *Artificial Intelligence: The Heuristic Programming Approach*. New York: McGraw-Hill.

Slocum, J. (1985). A survey of machine translation: Its history, current status, and future prospects. *Computational Linguistics 11*(1), 1–17. http://www.aclweb.org/anthology/J85-1001.

Sloman, A. (1971). Interactions between philosophy and Artificial Intelligence. *Artificial Intelligence 2*, 209–225. http://www.cs.bham.ac.uk/research/projects/cogaff/sloman-analogical-1971/.

Sloman, A. (1978). *The Computer Revolution in Philosophy: Philosophy, Science and Models of Mind*. Atlantic Highlands, NJ: Humanities Press. Online with afterthoughts at http://www.cs.bham.ac.uk/research/projects/cogaff/crp/crp-afterthoughts.html.

Sloman, A. (1996). Beyond Turing equivalence. In P. J. Millican and A. Clark (Eds.), *Machines and Thought: The Legacy of Alan Turing, Vol. I*, pp. 179–219. Oxford: Clarendon Press. http://www.cs.bham.ac.uk/research/projects/cogaff/Sloman.turing90.pdf.

Sloman, A. (1998, 25 January). Supervenience and implementation: Virtual and physical machines. http://www.cs.bham.ac.uk/research/projects/cogaff/Sloman.supervenience.and.implementation.pdf.

Sloman, A. (2002). The irrelevance of Turing machines to AI. In M. Scheutz (Ed.), *Computationalism: New Directions*, pp. 87–127. Cambridge, MA: MIT Press. Page references to preprint at http://www.cs.bham.ac.uk/research/projects/cogaff/sloman.turing.irrelevant.pdf.

Sloman, A. (2008). Why virtual machines really matter—for several disciplines. http://www.cs.bham.ac.uk/research/projects/cogaff/talks/information.pdf.

Sloman, A. (2019, 29 June). Jane austen's concept of information (not claude shannon's). http://www.cs.bham.ac.uk/research/projects/cogaff/misc/austen-info.html.

Sloman, A. (6 August 1989). Contribution to newsgroup discussion of "Is there a definition of AI?". Article 4702 of comp.ai, http://www.cse.buffalo.edu/~rapaport/defs.txt.

Sloman, A. (7 January 2010). Why symbol-grounding is both impossible and unnecessary, and why symbol-tethering based on theory-tethering is more powerful anyway. http://www.cs.bham.ac.uk/research/cogaff/talks/#models.

Sloman, A. and M. Croucher (1981). Why robots will have emotions. In *Proceedings of IJCAI 1981*, pp. 197–202. IJCAI. http://www.ijcai.org/Proceedings/81-1/Papers/039.pdf.

Smith, A. (1776). *Wealth of Nations*. http://www.econlib.org/library/Smith/smWN.html. For "On the Division of Labor" (Book I, Ch. I), link to http://www.econlib.org/library/Smith/smWNCover.html.

Smith, A. R. (2014a, 2 August). A business card universal Turing machine. http://alvyray.com/CreativeCommons/BizCardUniversalTuringMachine_v2.2.pdf.

Smith, A. R. (2014b, September). His just deserts: A review of four books. *Notices of the AMS 61*(8), 891–895. http://www.ams.org/notices/201408/201408-full-issue.pdf.

Smith, A. R. (2014c). Turingtoys.com. http://alvyray.com/CreativeCommons/TuringToysdotcom.htm.

Smith, B. C. (1985, January). Limits of correctness in computers. *ACM SIGCAS Computers and Society 14–15*(1–4), 18–26. Also published as *Technical Report* CSLI-85-36 (Stanford, CA: Center for the Study of Language & Information); reprinted in Charles Dunlop & Rob Kling (eds.), *Computerization and Controversy* (San Diego: Academic Press, 1991): 632–646; reprinted in Timothy R. Colburn, James H. Fetzer, & Terry L. Rankin (eds.), *Program Verification: Fundamental Issues in Computer Science* (Dordrecht, Holland: Kluwer Academic Publishers, 1993): 275–293.

Smith, B. C. (1987). The correspondence continuum. Technical Report CSLI-87-71, Center for the Study of Language & Information, Stanford, CA.

Smith, B. C. (1996). *On the Origin of Objects*. Cambridge, MA: MIT Press.

Smith, B. C. (2002). The foundations of computing. In M. Scheutz (Ed.), *Computationalism: New Directions*, pp. 23–58. Cambridge, MA: MIT Press.

Smith, P. (4 December 2010). Answer to 'What good is it to study philosophy?'. *AskPhilosophers.com*, http://www.askphilosophers.org/question/3710.

Smith, R. D. (2000). Simulation. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, 4th Edition*, pp. 1578–1587. London: Nature Publishing Group.

Soames, S. (2016, 7 March). Philosophy's true home. *New York Times*. http://opinionator.blogs.nytimes.com/2016/03/07/philosophys-true-home/.

Soare, R. I. (1999). The history and concept of computability. In E. Griffor (Ed.), *Handbook of Computability Theory*, pp. 3–36. Amsterdam: North-Holland. Page references are to the preprint at http://www.people.cs.uchicago.edu/~soare/History/handbook.pdf.

Soare, R. I. (2009). Turing oracle machines, online computing, and three displacements in computability theory. *Annals of Pure and Applied Logic 160*, 368–399. Preprint at http://www.people.cs.uchicago.edu/~soare/History/turing.pdf; published version at http://ac.els-cdn.com/S0168007209000128/1-s2.0-S0168007209000128-main.pdf?_tid=8258a7e2-01ef-11e4-9636-00000aab0f6b&acdnat=1404309072_f745d1632bb6fdd95f711397fda63ee2. A slightly different version appears as Soare 2013a.

Soare, R. I. (2012). Formalism and intuition in computability. *Philosophical Transactions of the Royal Society A 370*, 3277–3304. doi:10.1098/rsta.2011.0335.

Soare, R. I. (2013a). Interactive computing and relativized computability. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 203–260. Cambridge, MA: MIT Press. A slightly different version appeared as Soare 2009.

Soare, R. I. (2013b). Turing and the art of classical computability. In S. B. Cooper and J. van Leeuwen (Eds.), *Alan Turing: His Work and Impact*, pp. 65–70. Elsevier. http://www.people.cs.uchicago.edu/~soare/Art/.

Soare, R. I. (2016). *Turing Computability: Theory and Applications*. Berlin: Springer.

Soni, J. and R. Goodman (2017, 12 July). A man in a hurry: Claude Shannon's New York years. *IEEE Spectrum*. http://spectrum.ieee.org/geek-life/history/a-man-in-a-hurry-claude-shannons-new-york-years.

Sparrow, R. (2007). Killer robots. *Journal of Applied Philosophy 24*(1), 62–77. http://wmpeople.wm.edu/asset/index/cvance/sparrow.

Sprevak, M. (2010, September). Computation, individuation, and the received view on representation. *Studies in History and Philosophy of Science 41*(3), 260–270.

Sprigman, C. J. (2015, May). *Oracle v. Google:* a high-stakes legal fight for the software industry. *Communications of the ACM 58*(5), 27–29.

Spruit, L. and G. Tamburrini (1991). Reasoning and computation in Leibniz. *History and Philosophy of Logic 12*, 1–14.

Squires, R. (1970, October). On one's mind. *Philosophical Quarterly 20*(81), 347–356.

Srihari, S. N. (2010, 29 November). Beyond C.S.I.: The rise of computational forensics. *IEEE Spectrum*. http://spectrum.ieee.org/computing/software/beyond-csi-the-rise-of-computational-forensics.

Stairs, A. (2014). Response to question about the definition of 'magic'. http://www.askphilosophers.org/question/5735.

Standage, T. (1998). *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers*. New York: Walker Publishing. Reviewed in Alden 1999.

Stanley, J. and T. Williamson (2001, August). Knowing how. *Journal of Philosophy 98*(8), 411–444. http://www.thatmarcusfamily.org/philosophy/Course_Websites/Readings/Stanley%20and%20Williamson%20-%20Knowing%20How.pdf.

Staples, M. (2014). Critical rationalism and engineering: Ontology. *Synthese 191*(10), 2255–2279. http://www.nicta.com.au/pub?doc=7397.

Staples, M. (2015). Critical rationalism and engineering: Methodology. *Synthese 192*(1), 337–362. http://www.nicta.com.au/pub?doc=7747.

Steed, S. (2013, February). Harnessing human intellect for computing. *Computing Research News 25*(2). http://www.humancomputation.com/2013/CRN-Feb2013.pdf.

Stein, D. K. (1984, Autumn). Lady Lovelace's notes: Technical text and cultural context. *Victorian Studies 28*(1), 33–67.

Stein, D. K. (1985). *Ada: A Life and a Legacy*. Cambridge, MA: MIT Press. Reviewed in Kidder 1985.

Stepney, S., S. L. Braunstein, J. A. Clark, A. Tyrrell, A. Adamatzky, R. E. Smith, T. Addis, C. Johnson, J. Timmis, P. Welch, R. Milner, and D. Partridge (2005, March). Journeys in non-classical computation I: A grand challenge for computing research. *International Journal of Parallel, Emergent and Distributed Systems 20*(1), 5–19. https://www-users.cs.york.ac.uk/susan/bib/ss/nonstd/ijpeds20-1.pdf.

Stern, H. and L. Daston (1984, 26 April). Turing & the system. *New York Review of Books*, 52–53. http://www.nybooks.com/articles/archives/1984/apr/26/turing-the-system/.

Sternberg, R. J. (1990). *Metaphors of Mind: Conceptions of the Nature of Intelligence*. New York: Cambridge University Press.

Stevens, Jr., P. (1996). Magic. In D. Levinson and M. Ember (Eds.), *Encyclopedia of Cultural Anthropology*, pp. 721–726. New York: Henry Holt.

Stewart, I. (1994, September). A subway named Turing. *Scientific American*, 104, 106–107.

Stewart, I. (2000, January). Impossibility theorems. *Scientific American*, 98–99.

Stewart, I. (2001, March). Easter is a quasicrystal. *Scientific American*, 80, 82–83. http://www.whydomath.org/Reading_Room_Material/ian_stewart/2000_03.html.

Stewart, N. (1995, March). Science and computer science. *ACM Computing Surveys 27*(1), 39–41. Longer version at http://www.iro.umontreal.ca/~stewart/science_computerscience.pdf.

Stoll, C. (2006, May). When slide rules ruled. *Scientific American 294*(5), 80–87.

Stork, D. G. (1997). *HAL's Legacy:* 2001*'s Computer as Dream and Reality*. Cambridge, MA: MIT Press.

Strasser, C. and G. A. Antonelli (2015). Non-monotonic logic. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Fall 2015 edition)*. Stanford University. http://plato.stanford.edu/entries/logic-nonmonotonic/.

Strawson, G. (2012). Real naturalism. *Proeedings and Addresses of the American Philosophical Association 86*(2), 125–154.

Strevens, M. (2013, 24 November). Looking into the black box. *New York Times Opinionator*. http://opinionator.blogs.nytimes.com/2013/11/24/looking-into-the-black-box.

Strevens, M. (2019, 28 March). The substantiality of philosophical analysis. *The Brains Blog*, http://philosophyofbrains.com/2019/03/28/the--substantiality--of--philosophical--analysis.aspx.

Suber, P. (1988). What is software? *Journal of Speculative Philosophy 2*(2), 89–119. Revised version at http://www.earlham.edu/~peters/writing/software.htm.

Suber, P. (1997a). Formal systems and machines: An isomorphism. http://www.earlham.edu/~peters/courses/logsys/machines.htm.

Suber, P. (1997b). The Löwenheim-Skolem theorem. https://legacy.earlham.edu/~peters/courses/logsys/low-skol.htm.

Suber, P. (1997c). Turing machines. http://www.earlham.edu/~peters/courses/logsys/turing.htm.

Suber, P. (1997d). Turing machines II. http://www.earlham.edu/~peters/courses/logsys/turing2.htm.

Suber, P. (2002). Sample formal system S. http://www.earlham.edu/~peters/courses/logsys/sys-xmpl.htm.

Suits, D. B. (2005, July 1989). Out of the chinese room. *Computers & Philosophy Newsletter* (4:1+4:2), 1–7. https://people.rit.edu/dbsgsh/Out%20of%20the%20Chinese%20Room.htm.

Sullivan, E. (2019). Understadning from machine learning models. *British Journal for the Philosophy of Science*. https://philpapers.org/go.pl?id=SULUFM&u=https%3A%2F%2Fphilpapers.org%2Farchive%2FSULUFM.pdf.

Swade, D. D. (1993, February). Redeeming Charles Babbage's mechanical computer. *Scientific American*, 86–91.

Swoyer, C. (1991, June). Structural representation and surrogative reasoning. *Synthese 87*(3), 449–508.

Talmy, L. (2000). *Toward a Cognitive Semantics*. Cambridge, MA: MIT Press.

Tam, W. C. (1992, March). Teaching loop invariants to beginners by examples. *SIGSCE Bulletin 24*(1), 92–96.

Tanenbaum, A. S. (2006). *Structured Computer Organization, Fifth Edition*. Upper Saddle River, NJ: Pearson Prentice Hall. http://e-book.az/download?id=821.

Tarski, A. (1969, June). Truth and proof. *Scientific American*, 63–70, 75–77. https://cs.nyu.edu/mishra/COURSES/13.LOGIC/Tarski.pdf.

Tedre, M. (2007). Know your discipline: Teaching the philosophy of computer science. *Journal of Information Technology Education 6*, 105–122.

Tedre, M. (2008, September+October). What should be automated? *ACM Interactions 15*(5), 47–49. https://homepages.dcc.ufmg.br/~loureiro/cm/082/WhatShouldBeAutomated.pdf.

Tedre, M. (2015). *The Science of Computing: Shaping a Discipline*. Boca Raton, FL: CRC Press/Taylor & Francis.

Tedre, M. and P. J. Denning (2016). The long quest for computational thinking. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, New York, pp. 120–129. ACM. http://denninginstitute.com/pjd/PUBS/long-quest-ct.pdf.

Tedre, M. and N. Moisseinen (2014). Experiments in computing: A survey. *Scientific World Journal 2014*. Article ID 549398, http://dx.doi.org/10.1155/2014/549398.

Tedre, M. and E. Sutinen (2008, September). Three traditions of computing: What educators should know. *Computer Science Education 18*(3), 153–170.

Tenenbaum, A. M. and M. J. Augenstein (1981). *Data Structures using Pascal*. Englewood Cliffs, NJ: Prentice-Hall.

Teuscher, C. and M. Sipper (2002, August). Hypercomputation: Hype or computation? *Communications of the ACM 45*(8), 23–30. http://www.cs.bgu.ac.il/~sipper/papabs/hypercomp.pdf.

Thagard, P. (1978). Why astrology is a pseudoscience. *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association 1*, 223–234.

Thagard, P. (1984). Computer programs as psychological theories. In O. Neumaier (Ed.), *Mind, Language and Society*, pp. 77–84. Vienna: Conceptus-Studien.

Thagard, P. (2006). *Hot Thought: Mechanisms and Applications of Emotional Cognition*. Cambridge, MA: MIT Press.

Thagard, P. (2007, January). Coherence, truth, and the development of scientific knowledge. *Philosophy of Science 74*, 28–47.

Thagard, P. (4 December 2012). Eleven dogmas of analytic philosophy. *Psychology Today*, http://www.psychologytoday.com/blog/hot-thought/201212/eleven-dogmas-analytic-philosophy.

Thatcher, M. E. and D. E. Pingry (2007, October). Software patents: The good, the bad, and the messy. *Communications of the ACM 50*(10), 47–52.

The Economist (2013, 19 October). Unreliable research: Trouble at the lab. *The Economist*. http://www.economist.com/news/briefing/21588057-scientists-think-science-self-correcting-alarming-degree-it-not-trouble.

Thomason, R. H. (2003). Dynamic contextual intensional logic: Logical foundations and an application. In P. Blackburn (Ed.), *CONTEXT 2003: Lecture Notes in Artificial Intelligence 2680*, pp. 328–341. Berlin: Springer-Verlag. http://link.springer.com/chapter/10.1007/3-540-44958-2_26#page-1.

Thompson, C. (2019, 13 February). The secret history of women in coding. *New York Times Magazine*. https://www.nytimes.com/2019/02/13/magazine/women-coding-computer-programming.html.

Thurston, W. P. (1994, April). On proof and progress in mathematics. *Bulletin of the American Mathematical Society 30*(2), 161–177.

Tingley, K. (2013, 25 November). The body electric. *The New Yorker*, 78–80, 82, 86–86. https://www.newyorker.com/magazine/2013/11/25/the-body-electric.

Toulmin, S. (1984, 19 January). Fall of a genius. *New York Review of Books*, 3–4, 6. http://www.nybooks.com/articles/archives/1984/jan/19/fall-of-a-genius/.

Touretzky, D. (2008). Gallery of css descramblers. http://www.cs.cmu.edu/~dst/DeCSS/Gallery/.

Toussaint, G. (1993, Summer). A new look at Euclid's second proposition. *The Mathematical Intelligencer 15*(3), 12–23. http://cgm.cs.mcgill.ca/~godfried/publications/euclid.pdf.

Traub, J. (2011, January). What is computation? what is the right computational model for continuous scientific problems? *Ubiquity 2011*(January). Article 2, http://ubiquity.acm.org/article.cfm?id=1925842.

Tucker et al., A. (2003). A model curriculum for K–12 computer science: Final report of the ACM K–12 Task Force Curriculum Committee, second edition. Computer Science Teachers Association and Association for Computing Machinery, http://www.acm.org/education/education/curric_vols/k12final1022.pdf.

Tukey, J. W. (1958, January). The teaching of concrete mathematics. *American Mathematical Monthly 65*(1), 1–9.

Turing, A. M. (1936). On computable numbers, with an application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society, Ser. 2, Vol. 42*, 230–265. https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf.

Turing, A. M. (1937, December). Computability and λ-definability. *Journal of Symbolic Logic 2*(4), 153–163.

Turing, A. M. (1938). On computable numbers, with an application to the *Entscheidungsproblem*: A correction. *Proceedings of the London Mathematical Society, Ser. 2 43*(1), 544–546. https://www.wolframscience.com/prizes/tm23/images/Turing2.pdf.

Turing, A. M. (1939). Systems of logic based on ordinals. *Proceedings of the London Mathematical Society S2-45*(1), 161–228. Reprinted with commentary in Copeland 2004b, Ch. 3.

Turing, A. M. (1947). Lecture to the London Mathematical Society on 20 february 1947. In B. J. Copeland (Ed.), *The Essential Turing*, pp. 378–394. Oxford: Oxford University Press (2004). Editorial commentary on pp. 362–377; online at https://www.academia.edu/34875977/Alan_Turing_LECTURE_TO_THE_LONDON_MATHEMATICAL_SOCIETY_1947_.

Turing, A. M. (1948). Intelligent machinery. In B. J. Copeland (Ed.), *The Essential Turing*, pp. 410–432. Oxford: Oxford University Press (2004). Editorial commentary on pp. 395–409. Typescript at: http://www.alanturing.net/intelligent_machinery/.

Turing, A. M. (1949). Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 67–69. Cambridge, UK: University Mathematics Lab. Reprinted in Morris and Jones 1984; manuscript online at http://www.turingarchive.org/browse.php/b/8.

Turing, A. M. (1950, October). Computing machinery and intelligence. *Mind 59*(236), 433–460. http://phil415.pbworks.com/f/TuringComputing.pdf.

Turing, A. M. (1951, 1996). Intelligent machinery, a heretical theory. *Philosophia Mathematica 4*(3), 256–260. http://viola.informatik.uni-bremen.de/typo/fileadmin/media/lernen/Turing-_Intelligent_Machinery.pdf; typescripts and further bibliographic information at http://www.turing.org.uk/sources/biblio1.html.

Turing, A. M. (1953). Chess. In B. J. Copeland (Ed.), *The Essential Turing*, pp. 562–575. Oxford: Oxford University Press (2004).

Turing, A. M. (1954). Solvable and unsolvable problems. *Science News 31*, 7–23. Page references are to the reprint in Copeland 2004b, Ch. 17, http://www.ivanociardelli.altervista.org/wp-content/uploads/2018/04/Solvable-and-unsolvable-problems.pdf.

Turner, R. (2010). Programming languages as mathematical theories. In J. Vallverdú (Ed.), *Thinking Machines and the Philosophy of Computer Science: Concepts and Principles*, pp. 66–82. IGI Global. http://www.irma-international.org/viewtitle/62539/.

Turner, R. (2011). Specification. *Minds and Machines 21*(2), 135–152. https://www.academia.edu/456275/SPECIFICATION.

Turner, R. (2018). *Computational Artifacts: Towards a Philosophy of Computer Science*. Berlin: Springer.

Turner, R. (2019). Correctness, explanation and intention. In F. Manea, B. Martin, D. Paulusma, and G. Primiero (Eds.), *Computing with Foresight and Industry. CiE 2019*, pp. 62–71. Cham, Switzerland: Springer.

Turner, R. and A. H. Eden (2007). Towards a programming language ontology. In G. Dodig-Crnkovic and S. Stuart (Eds.), *Computation, Information, Cognition: The Nexus and the Liminal*, pp. 147–159. Cambridge, UK: Cambridge University Press. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.82.194&represent=rep1&type=pdf.

Turner, Z. (2015, 30 March). Beautiful code. *The New Yorker*, 21–22. http://www.newyorker.com/magazine/2015/03/30/beautiful-code.

Tversky, A. and D. Kahneman (1974, 27 September). Judgment under uncertainty: Heuristics and biases. *Science 185*(4157), 1124–1131. http://psiexp.ss.uci.edu/research/teaching/Tversky_Kahneman_1974.pdf.

Tymoczko, T. (1979, February). The four-color problem and its philosophical significance. *Journal of Philosophy 76*(2), 57–83. http://www.thatmarcusfamily.org/philosophy/Course_Websites/Math_S08/Readings/tymoczko.pdf.

Tymoczko, T. and S. Goodhart (1986, March). From logic to computers: A suggestion for logic teachers. *Teaching Philosophy 9*(1), 15–33.

Uebel, T. (2012). Vienna circle. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Summer 2012 Edition)*. Stanford University. http://plato.stanford.edu/archives/sum2012/entries/vienna-circle/.

Uglow, J. (2010, 24 June). The other side of science. *New York Review of Books*, 30–31, 34.

Uglow, J. (2018, 22 November). Stepping out of byron's shadow. *New York Review of Books 65*(18), 30–32.

Unger, P. (1979a). I do not exist. In G. F. Macdonald (Ed.), *Perception and Identity*. Ithaca, NY: Cornell University Press.

Unger, P. (1979b). Why there are no people. In *Studies in Metaphysics (Midwest Studies in Philosophy, Vol. 4)*, pp. 177–222. Minneapolis: University of Minnesota Press. http://www.thatmarcusfamily.org/philosophy/Course_Websites/Readings/Unger%20-%20No%20People.pdf.

Vahid, F. (2003, April). The softening of hardware. *IEEE Computer 36*(4), 27–34. http://www.cs.ucr.edu/~vahid/pubs/comp03_softerhw.pdf.

van Fraassen, B. C. (1989). *Laws and Symmetry*. Oxford: Clarendon Press. http://joelvelasco.net/teaching/120/vanfraassen_-Laws_and_Symmetry.pdf.

van Fraassen, B. C. (2006, December). Representation: The problem for structuralism. *Philosophy of Science 73*, 536–547. http://www.princeton.edu/~fraassen/abstract/docs-publd/PSA04_Structure.pdf.

van Leeuwen, J. and J. Wiedermann (2000). The Turing machine paradigm in contemporary computing. In B. Engquist and W. Schmid (Eds.), *Mathematics Unlimited—2001 and Beyond*, pp. 1139–1155. Berlin: Springer-Verlag. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.3925&rep=rep1&type=pdf.

van Leeuwen, J. and J. Wiedermann (2013). The computational power of Turing's non-terminating circular a-machines. In S. B. Cooper and J. van Leeuwen (Eds.), *Alan Turing: His Work and Impact*, pp. 80–85. Amsterdam: Elsevier.

Vardi, M. Y. (2010, September). Science has only two legs. *Communications of the ACM 53*(9), 5.

Vardi, M. Y. (2011a, December). Computing for humans. *Communications of the ACM 54*(12), 5.

Vardi, M. Y. (2011b, July). Solving the unsolvable. *Communications of the ACM 54*(7), 5. http://www.inf.unibz.it/~calvanese/teaching/tc/material/solving-the-unsolvable-CACM-2011-07.pdf.

Vardi, M. Y. (2012, March). What is an algorithm? *Communications of the ACM 55*(3), 5. http://cacm.acm.org/magazines/2012/3/146261-what-is-an-algorithm/fulltext.

Vardi, M. Y. (2013, January). Who begat computing? *Communications of the ACM 56*(1), 5. http://cacm.acm.org/magazines/2013/1/158780-who-begat-computing/fulltext.

Vardi, M. Y. (2014, March). Boolean satisfiability: Theory and engineering. *Communications of the ACM 57*(3), 5. http://cacm.acm.org/magazines/2014/3/172516-boolean-satisfiability/fulltext.

Vardi, M. Y. (2016, May). The moral imperative of Artificial Intelligence. *Communications of the ACM 59*(5), 5. http://delivery.acm.org/10.1145/2910000/2903530/p5-vardi.pdf.

Vardi, M. Y. (2017, November). Would Turing have won the Turing award? *Communications of the ACM 60*(11), 7. https://cacm.acm.org/magazines/2017/11/222163-would-turing-have-won-the-turing-award/fulltext.

Veblen, T. (1908). The evolution of the scientific point of view. *The University of California Chronicle: An Official Record 10*(4), 395–416. http://archive.org/details/universitycalif08goog.

Venkatasubramanian, S. (2018, 10 May). Mr. Spock has left the building: The computational and ethical ramifications of automated decision-making in society. Talk given to the SUNY Buffalo Department of Computer Science & Engineering, https://www.youtube.com/watch?v=d5NXF5y1VlQ.

Vera, A. (2018, 30 April). Social animals. *New Yorker*, 3. Letter to the editor, https://www.newyorker.com/magazine/2018/04/30/letters-from-the-april-30-2018-issue.

Verity, J. W. (1985, 15 February). Bridging the software gap. *Datamation*, 84–88.

von Neumann, J. (1945, (1993)). First draft report on the EDVAC. *IEEE Annals of the History of Computing 15*(4), 27–75. Michael D. Godfrey (ed.); page references to online version at http://virtualtravelog.net.s115267.gridserver.com/wp/wp-content/media/2003-08-TheFirstDraft.pdf.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. Urbana, IL: University of Illinois Press. Arthur W. Burks (ed.).

von Tunzelmann, A. (2014, 20 November). The imitation game: Inventing a new slander to insult Alan Turing. *The Guardian*. http://www.theguardian.com/film/2014/nov/20/the-imitation-game-invents-new-slander-to-insult-alan-turing-reel-history.

Wade, N. (2017, 1 June). You look familiar. now scientists know why. *New York Times*. https://www.nytimes.com/2017/06/01/science/facial-recognition-brain-neurons.html.

Wadler, P. (1997, September). How to declare an imperative. *ACM Computing Surveys 29*(3), 240–263. https://wiki.ittc.ku.edu/lambda/images/3/3b/Wadler_-_How_to_Declare_an_Imperative.pdf.

Wagner, A. R. and R. C. Arkin (2011). Acting deceptively: Providing robots with the capacity for deception. *International Journal of Social Robotics 3*(1), 5–26. http://www.cc.gatech.edu/~alanwags/pubs/Acting-Deceptively-Final.pdf.

Wainer, H. (2007, May-June). The most dangerous equation. *American Scientist 95*(3), 249ff. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.504.3301&rep=rep1&type=pdf.

Wainer, H. (2012, September-October). The survival of the fittists. *American Scientist 100*, 358–361. https://www.americanscientist.org/article/the-survival-of-the-fittists.

Waldrop, M. M. (2001, December). The origins of personal computing. *Scientific American*, 84–91.

Wallach, W. and C. Allen (2009). *Moral Machines: Teaching Robots Right from Wrong*. New York: Oxford University Press.

Wallas, G. (1926). *The Art of Thought*. Kent, UK: Solis Press (2014).

Wallich, P. (1997, April). Cracking the U.S. code. *Scientific American*, 42.

Walsh, T. (2014, November-December). Candy Crush's puzzling mathematics. *American Scientist 102*(6), 430–433. https://www.americanscientist.org/article/candy-crushs-puzzling-mathematics.

Wang, H. (1957, January). A variant to Turing's theory of computing machines. *Journal of the ACM 4*(1), 63–92.

Wang, Z., J. R. Busemeyer, H. Atmanspacher, and E. M. Pothos (Eds.) (2013). *Topics in Cognitive Science 5(4) (October): The Potential of Using Quantum Theory to Build Models of Cognition*. Cognitive Science Society.

Wangsness, T. and J. Franklin (1966, April). "Algorithm" and "formula". *Communications of the ACM 9*(4), 243.

Wartofsky, M. W. (1966). The model muddle: Proposals for an immodest realism. In M. W. Wartofsky (Ed.), *Models: Representation and the Scientific Theory of Understanding*, pp. 1–11. Dordrecht, The Netherlands: D. Reidel, 1979.

Wartofsky, M. W. (1979). Introduction. In M. W. Wartofsky (Ed.), *Models: Representation and the Scientific Theory of Understanding*, pp. xiii–xxvi. Dordrecht, The Netherlands: D. Reidel.

Weatherson, B. (18 July 2012). What could leave philosohy? http://tar.weatherson.org/2012/07/18/what-could-leave-philosophy/.

Weatherson, B. and D. Marshall (2018). Intrinsic vs. extrinsic properties. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2018 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2018/entries/intrinsic-extrinsic/.

Wegner, P. (1976). Research paradigms in computer science. In *ICSE '76 Proceedings of the 2nd International Conference on Software Engineering*, pp. 322–330. Los Alamitos, CA: IEEE Computer Society Press.

Wegner, P. (1995, March). Interaction as a basis for empirical computer science. *ACM Computing Surveys 27*(1), 45–48.

Wegner, P. (1997, May). Why interaction is more powerful than algorithms. *Communications of the ACM 40*(5), 80–91. http://www.cs.brown.edu/people/pw/papers/ficacm.ps.

Wegner, P. (1999, January). Towards empirical computer science. *The Monist 82*(1), 58–108. http://www.cs.brown.edu/people/pw/papers/monist.ps.

Wegner, P. (2010, November). What is computation? the evolution of computation. *Ubiquity 2010*(November). Article 2, http://ubiquity.acm.org/article.cfm?id=1883611.

Wegner, P. and D. Goldin (1999). Interaction, computability, and Church's thesis. http://cs.brown.edu/~pw/papers/bcj1.pdf.

Wegner, P. and D. Goldin (2003, April). Computation beyond Turing machines. *Communications of the ACM 46*(4), 100–102. http://www.pld.ttu.ee/~vadim/AIRT/8_computation_beyond_turing_machines.pdf.

Wegner, P. and D. Goldin (2006a, March). Forum. *Communications of the ACM 49*(3), 11.

Wegner, P. and D. Goldin (2006b, July). Principles of problem solving. *Communications of the ACM 49*(7), 27–29. https://www.researchgate.net/publication/220420866_Principles_of_problem_solving.

Weinberg, J. (2019, 15 May). Did a story about a computer made of humans scoop Searle's"Chinese room' by 20 years? *Daily Nous*. http://dailynous.com/2019/05/15/story-computer-made-humans-scoop-searles-chinese-room-20-years/.

Weinberg, S. (2002, 24 October). Is the universe a computer? *New York Review of Books 49*(16). http://www.nybooks.com/articles/2002/10/24/is-the-universe-a-computer/.

Weinberg, S. (2017, 19 January). The trouble with quantum mechanics. *New York Review of Books 64*(1), 51–53. http://www.nybooks.com/articles/2017/01/19/trouble-with-quantum-mechanics/.

Weinberger, D. (2012, 27 April). Shift happens. *The Chronicle [of Higher Education] Review*, B6–B9.

Weiner, J. (2017, 15 March). The magician who wants to break magic. *New York Times Magazine*. https://www.nytimes.com/2017/03/15/magazine/derek-delgaudio-the-magician-who-wants-to-break-magic.html.

Weir, A. (2015). Formalism in the philosophy of mathematics. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Spring 2015 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/spr2015/entries/formalism-mathematics/.

Weizenbaum, J. (1966, January). ELIZA—A computer program for the study of natural language communication between man and machine. *Communications of the ACM 9*(1), 36–45. http://web.stanford.edu/class/linguist238/p36-weizenabaum.pdf; reprinted in *Communications of the ACM* 26(1) (January 1983: 23–28; be sure to read the correspondence between Weizenbaum and Arbib on p. 28 of the reprint.

Weizenbaum, J. (1967, August). Contextual understanding by computers. *Communications of the ACM 10*(8), 474–480.

Weizenbaum, J. (1976). *Computer Power and Human Reason*. New York: W.H. Freeman.

Welch, P. D. (2007). Turing unbound: Transfinite computation. In S. Cooper, B. Löwe, and A. Sorbi (Eds.), *CiE 2007*, pp. 768–780. Berlin: Springer-Verlag Lecture Notes in Computer Science 4497.

Wells, B. (2003). The architecture of Colossus, the first PC (abstract). http://www.stanford.edu/class/ee380/Abstracts/040204.html.

Wells, B. (2004). Hypercomputation by definition. *Theoretical Computer Science 317*, 191–207. http://research.cs.queensu.ca/home/akl/cisc879/papers/PAPERS_FROM_THEORETICAL_COMPUTER_SCIENCE/05051011123316076.pdf.

Wheeler, D. L. (1997, 3 October). An ancient feud: Who invented the computer? *Chronicle of Higher Education*, B2.

Wheeler, G. (2013, April). Models, models, and models. *Metaphilosophy 44*(3), 293–300. http://philsci-archive.pitt.edu/9500/1/Models.pdf.

White, T. I. (2007). *In Defense of Dolphins: The New Moral Frontier*. Oxford: Blackwell.

White, T. I. (2013). A primer on nonhuman personhood, cetacean rights and 'flourishing'. http://us.whales.org/issues/primer-on-non-human-personhood-and-cetacean-rights and http://indefenseofdolphins.com/wp-content/uploads/2013/07/primer.pdf.

Whitemore, H. (1966). *Breaking the Code*. Samuel French Inc., 2010.

Whitemore, H. (1988, March/April). The Enigma: Alan Turing confronts a question of right and wrong. *The Sciences 28*(2), 40–41.

Wiebe, J. M. and W. J. Rapaport (1986). Representing *de re* and *de dicto* belief reports in discourse and narrative. *Proceedings of the IEEE 74*, 1405–1413. http://www.cse.buffalo.edu/~rapaport/Papers/wieberapaport86.pdf.

Wiedermann, J. (1999, September). Simulating the mind: A gauntlet thrown to computer science. *ACM Computing Surveys 31*(3es), Paper No. 16.

Wiener, N. (1961). *Cybernetics, or Control and Communication in the Animal and the Machine, 2nd Edition*. Cambridge, MA: MIT Press.

Wiener, N. (1964). *God and Golem, Inc.: A Comment on Certain Points Where Cybernetics Impinges on Religion*. Cambridge, MA: MIT Press. http://www.scribd.com/doc/2962205/God-and-Golem-Inc-Wiener and http://simson.net/ref/1963/God_And_Golem_Inc.pdf.

Wiesner, J. (1958, October). Communication sciences in a university environment. *IBM Journal of Research and Development 2*(4), 268–275.

Wigner, E. (1960, February). The unreasonable effectiveness of mathematics in the natural sciences. *Communications in Pure and Applied Mathematics 13*(1). https://www.dartmouth.edu/~matc/MathDrama/reading/Wigner.html, http://www.maths.ed.ac.uk/~aar/papers/wigner.pdf, and http://nedwww.ipac.caltech.edu/level5/March02/Wigner/Wigner.html.

Wilford, J. N. (2006, 30 November). Early astronomical 'computer' found to be technically complex. *New York Times*.

Wilford, J. N. (2008, 31 July). A device that was high-tech in 100 B.C. (discovering how greeks computed in 100 B.C.). *New York Times*, A12. https://www.nytimes.com/2008/07/31/science/31computer.html.

Wilkes, M. (1953, May). Can machines think? *Discovery 14*, 151. Reprinted in *Proceedings of the Institute of Radio Engineers* 41(10) (October): 1230–1234.

Wilks, Y. (1974, January). One small head—models and theories in linguistics. *Foundations of Language 11*(1), 77–95. Revised version in Partridge and Wilks 1990, pp. 121–134.

Williams, B. (1998, 19 November). The end of explanation? *The New York Review of Books 45*(18), 40–44.

Williamson, T. (2007). *The Philosophy of Philosophy*. Oxford: Blackwell.

Williamson, T. (2011, 4 September). What is naturalism? *New York Times Opinionator: The Stone*. http://opinionator.blogs.nytimes.com/2011/09/04/what-is-naturalism/.

Willingham, D. T. (2011, May). Trust me, I'm a scientist. *Scientific American*. http://www.scientificamerican.com/article.cfm?id=trust-me-im-a-scientist.

Wilson, D. G. and J. Papadopoulos (2004). *Bicycling Science, Third Edition*. Cambridge, MA: MIT Press.

Wilson, E. and E. Frenkel (2013). Two views: How much math do scientists need? *Notices of the AMS 60*(7), 837–838. http://dx.doi.org/10.1090/noti1032.

Wing, J. M. (2006, March). Computational thinking. *Communications of the ACM 49*(3), 33–35. https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf.

Wing, J. M. (2008a). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A 366*, 3717–3725. http://www.cs.cmu.edu/~wing/publications/Wing08a.pdf.

Wing, J. M. (2008b, January). Five deep questions in computing. *Communications of the ACM 51*(1), 58–60. http://www.cs.cmu.edu/~wing/publications/Wing08.pdf.

Wing, J. M. (2010, 17 November). Computational thinking: What and why? *The Link* (Carnegie-Mellon University), https://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf.

Winkler, J. F. H. (2012, October). Konrad zuse and floating-point numbers. *Communications of the ACM 55*(10), 6–7. (Letter to the Editor).

Winograd, T. (1983). *Language as a Cognitive Process; Vol. 1: Syntax*. Reading, MA: Addison-Wesley.

Winograd, T. and F. Flores (1987). *Understanding Computers and Cognition: A New Foundation for Design*. Reading, MA: Addison-Wesley.

Winston, P. H. (1977). *Artificial Intelligence*. Reading, MA: Addison-Wesley.

Wirth, N. (1971, April). Program development by stepwise refinement. *Communications of the ACM 14*(4), 221–227. http://oberoncore.ru/_media/library/wirth_program_development_by_stepwise_refinement2.pdf, http://sunnyday.mit.edu/16.355/wirth-refinement.html, and http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.9376&rep=rep1&type=pdf.

Wittgenstein, L. (1921). *Tractatus Logico-Philosophicus*. New York: Humanities Press. Second Edition, with corrections, 1972.

Wittgenstein, L. (1958). *Philosophical Investigations, Third Edition*. New York: Macmillan.

Wittgenstein, L. (1980). *Remarks on the Philosophy of Psychology, Vol. I*. Chicago: University of Chicago Press.

Woit, P. (2014, 19 November). The imitation game. *Not Even Wrong*. http://www.math.columbia.edu/~woit/wordpress/?p=7365.

Wolfram, S. (2002a). Introduction to *A New Kind of Science*. http://www.stephenwolfram.com/publications/introduction-to-a-new-kind-of-science/.

Wolfram, S. (2002b). *A New Kind of Science*. Wolfram Media. http://www.wolframscience.com/nks/.

Wright, A. (2008, 17 June). The Web time forgot. *New York Times*, F1, F4. http://www.nytimes.com/2008/06/17/science/17mund.html.

Wu, T. (2017, 15 July). Please prove you're not a robot. *New York Times Sunday Review*. https://www.nytimes.com/2017/07/15/opinion/sunday/please-prove-youre-not-a-robot.html.

Wulf, W. (1995, March). Are we scientists or engineers? *ACM Computing Surveys 27*(1), 55–57.

Young, J. O. (2018). The coherence theory of truth. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Fall 2018 Edition)*. http://plato.stanford.edu/archives/fall2018/entries/truth-coherence/.

Zach, R. (2019). Hilbert's program. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Summer 2019 ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2019/entries/hilbert-program/.

Zeigler, B. P. (1976). *Theory of Modeling and Simulation*. New York: Wiley Interscience.

Zemanek, H. (1971, August). Was ist Informatik? (What is informatics?). *Elektronische Rechenanlagen (Electronic Computing Systems) 13*(4), 157–171.

Zenil, H. and F. Hernández-Quiroz (2007). On the possible computational power of the human mind. In C. Gershenson, D. Aerts, and B. Edmonds (Eds.), *Worldviews, Science and Us: Philosophy and Complexity*, pp. 315–337. Singapore: World Scientific Publishing. Page references to preprint at http://arxiv.org/abs/cs/0605065.

Zimmer, C. (2016, 21 July). In brain map, gears of mind get rare look. *New York Times*, A1. http://www.nytimes.com/2016/07/21/science/human-connectome-brain-map.html.

Zipes, J. (1995). Breaking the disney spell. In M. Tatar (Ed.), *The Classic Fairy Tales: Texts, Criticism*, pp. 332–352. New York: W.W. Norton (1999).

Zobrist, A. L. (2000). Computer games: Traditional. In A. Ralston, E. D. Reilly, and D. Hemmendinger (Eds.), *Encyclopedia of Computer Science, 4th edition*, pp. 364–368. New York: Grove's Dictionaries.

Zremski, J. (2009, 18 February). Perspectives differ on autopilot, icing. *Buffalo News*, A1–A2. http://www.buffalonews.com/Perspectives_differ_on_autopilot_icing_____Amid_probe_of_crash_no_federal_consensus_for_deadly_problem.html.

Zupko, J. (2011). John Buridan, §7. In E. N. Zalta (Ed.), *Stanford Encyclopedia of Philosophy (Fall 2011 Edition)*. Stanford University. http://plato.stanford.edu/archives/fall2011/entries/buridan/.

Zylberberg, A., S. Dehaene, P. R. Roelfsema, and M. Sigman (2011, July). The human Turing machine: A neural framework for mental programs. *Trends in Cognitive Science 15*(7), 293–300. https://neuro.org.ar/sites/neuro.org.ar/files/Zylberberg%202011%20TiCS.pdf.