Monads by Example

Izaak Weiss

Contents

1 What are Monads?

It is entirely reasonable that this is the first question that anyone learning Monads asks, and it is also entirely reasonable that anyone who is teaching Monads answers. However, Monads are a complex concept that cannot be explained in a single sentence or even a single paragraph; to understand Monads you must simultaneously understand the problem they are trying to solve, their implementation, the interface for working with them, and the theoretical computational background. Therefore, I will not try to answer this question in a single phrase; my explanation of what Monads are is the entirety of this paper.

I would like to take a few moments to clear up one possible misconception. Monads are not special. They are a data structure, just like a Linked List or a Dictionary. They have methods that you can call, and they store data in the same way. They do not have a common sounding name, so they seem scary, and people have a tendency to define them using complex math or weird analogies, but I firmly believe that Monads are not actually any more complicated than the run of the mill data structures that programmers use every day.

1.1 Bad Explanations

Of all the one liner explanations of Monads, two stand out as being slightly useful.

- Monads are Containers (We will see the truth of this in Section 2)
- Monads are Computations (We will see the truth of this in Section 3)

Those sentences, while true, are useless to the first time user of Monads, because they have no experience with actual Monads. Hopefully, by the end of this paper, the reader will be able to understand these analogies.

In contrast, several other explanations range from useless to wrong.

- Monads are monoids in the category of endofunctors
- Monads are the the totality of all beings

The first definition is correct, and if you already know what a monoid, a category, and an endofunctor is, congratulations! You probably are working on, or have, your PhD, and you can use that definition to help inform the rest of this essay. If you have no idea what those words mean, don't worry about trying to interpret them. They're mathy words to describe what we're going to talk about later in more plain terms.

The second definition is wrong. It refers to a completely different concept in philosophy and theology that happens to have the same name. Ignore it.

Finally, we have definitions that are absurd:

- Monads are like burritos

I have seen people make fun of this definition a lot, but I've never seen it actually used. However, I doubt that it will be useful to anyone who doesn't know what Monads are.

Finally, I want to remark upon how varied and different Monads truly are; This essay intends to introduce a few common monads, and provide a framework for thinking about them, but you will still come across Monads which are foreign to you. By an analogy; consider this paper an introduction to music, where I talk about classical music, jazz, and rock and roll. That barely covers many genres of music; a reader of that paper would be confounded upon hearing rap for the first time.

2 Our First Monads

In this section, we will explore the common problem of how to report errors to the caller of a function, and provide a solution to the problem using one of the simplest Monads.

2.1 Error Handling in Plain Python

Python usually uses Exception raising and catching to report errors that happen during execution. It is a desirable feature to have in a scripting language, but it is less useful in systems languages like C, Go, or Rust, because exceptions are expensive in terms of memory and CPU time. It's also less useful in functional languages like Haskell or Scala, which use types and abstract data structures to make code more predictable and safer, a goal which is undermined when code can throw exceptions that crash the whole program. In the following section, I'll be exploring ways to handle errors in Python without throwing exceptions and without using Monads.

2.1.1 Division

```
def division(x, y):
    return x / y
```

Consider the above code fragment. This is a very simple function; one that is so simple it hardly deserves to exist. However, if `y` is zero, this function can throw a `ZeroDevisionError`. It's possible that we want to recover from this error gracefully: check the inputs and see if an error will occur, and return some error code instead of raising an exception.

However, we must decide what the error code should be. We cannot choose `0.0`, because that is correctly returned by `division(0.0,1.0)`. We cannot choose `-1.0`, because that is correctly returned by `division(-1.0,1.0)`. In fact, this function can return any possible floating point number, so we can't choose a floating point number as our error code. One solution is to return a float on success, but `None` on failure.

```
def division(x, y):
    if y == 0:
        return None
    return x / y
```

Now we have written a function that checks whether or not division is possible, and performs division if it is, but returns an error code if it is not.

2.1.2 Indexing

```
def index(ls, i):
    return ls[i]
```

The above code is similar to the last example; it will perform an index lookup into a list and return the item from the list if it can. If it cannot, we are still left with the problem: it throws an exception if the index is out of bounds. Let's try to do the same thing as above; rewrite this function so that it does not throw an error but instead uses an error code to signal something has gone wrong.

Our first guess might be to have our error code be the same as above, and just return `None`. However, this causes false positives, because the code `index([None], 0)` would also return `None`. In fact, python lets any value be inside of a list; there is no possible error code we can return that cannot also be in the list. Luckily, python lets us use multiple return values.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return False, None
    return True, ls[i]
```

This allows us to actually check whether or not this function has failed, without worrying about receiving an exception. This allows us to handle errors from outside of the function in a logical way:

```
ok, value = index([1,2,3],0)
if not ok:
    print("Oh no, we failed")
else:
    print(value)
```

2.1.3 Combining The Above

```
def inverse_element(ls, i):
    return 1/ls[i]
```

Now, we have combined the two operations in python which might lead to an exception, and we have done it in a way that allows for three different operations to result in an exception. We can rewrite this function so that it will not throw any errors by checking after each step for an error code.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]

def division(x, y):
    if y == 0:
        return None
    return x / y

def inverse_element(ls, i):
    failure, value = index(ls, i)
    if failure:
        return None

    return division(1, value)
```

This code works nicely; you can throw two types of errors at it, and it returns `None` when either occurs.

2.1.4 Other Programming Languages

Python has been very nice to us so far. In Python, it is easy to write a function that returns two values, or returns different types in different scenarios. Python also has other features, such as Exceptions, which make this rewriting we've been doing sort of useless. The most Pythonic way of writing the above would probably be to catch the exceptions:

```
def inverse_element(ls, i):
    try:
        return 1/ls[i]
    except (IndexError, ZeroDivisionError):
        return None
```

Other languages have their own ways of dealing with the errors we discussed above, and they all have their own benefits. In C, the common pattern is to have

the actual return value of the function be a number that indicates whether an error occurred, and if one did, what the error was. To get the meaningful result from the function, you pass a pointer to a block of memory into the function, and that function writes the answer you want into that block of memory. This is much faster and simpler than having to write all of the infrastructure required to deal with throwing exceptions and allowing someone above you to catch that exception.

In many modern languages, including Rust, Scala, and Haskell, the solution of choice is to use Monads.

2.2 The Option Monad

The Option Monad, also called the Maybe Monad in many programming languages, is a way of representing the result of a function or computation that might result in an error and produce no meaningful output. We call an Option Monad that has a value and represents a successful computation Some, and we call one that does not have a value and represents a failed computation Nothing.

These Monads are really easy to write in programming languages like Haskell, Scala, or Rust, but I'm going to write an implementation of the Option Monad in Python to help avoid any confusion about its inner workings. In the end, all Monads are just objects, like trees, lists, or dictionaries.

```
class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Nothing'
        else:
            return 'Some({})'.format(self._value)

    def is_some(self):
        if self._failed:
            return False
        return True

    def is_none(self):
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise Exception('This Option has no value')
```

```
        else:
            return self._value

    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

This is the longest piece of code we have had so far, so let me break it down bit by bit.

```
class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Nothing'
        else:
            return 'Some({})'.format(self._value)
```

The `__init__` function is the constructor or initializer function for classes in python. Here all we do is create a boolean that indicates whether or not we have failed the computation. If we have not failed it, we store the result of the computation in `_value`. Note that if `_failed` is `True`, then we don't care what is in `_value`, because the computation has failed and that value has no meaning. I will note that the users of this class will probably never call `__init__` themselves, as we will later write alternate constructors that are easier for people to use.

The `__repr__` function simply tells python how this object should be printed.

```
    def is_some(self):
        if self._failed:
            return False
        return True

    def is_none(self):
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise Exception('This Option has no value')
```

```
        else:
            return self._value
```

These three functions are the meat of the Option Monad; these are the ways we interact with it. The `is_some` function returns `True` when there is a meaningful return value, and `False` if the computation failed. `is_none` does the opposite. `unwrap` returns the value of the Option Monad *if there is a value to be returned*, otherwise it throws an error. In order to use the `unwrap` function without an error, you must first check to see whether the computation succeeded or failed.

```
    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

These functions, decorated with `@classmethod`, aren't methods of the object. Instead, they are methods that exist as part of the class itself; here, we use them as alternate constructors.

At this point, let me rewrite our exception-free code from above using the Option Monad.

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def inverse_element(ls, i):
    res = index(ls, i)
    if res.is_none():
        return Option.none()

    return division(1, res1.unwrap())
```

The above code is the exact same length in lines; and already has some benefits. First, these functions have a return type that can be determined just be looking at the code, is more useful in statically typed languages than dynamically typed languages (like Python). Second, we do not have to remember the convention for every function. Before, we had to remember that `division` returned `None` for an

error, but `index` returned `False, None` for an error. Despite these benefits, the code is still filled with checks for errors and clustered with temporary variables.

And that's because we haven't yet implemented the most important function for a Monad. This is the most important but also the most complicated part of the Option Monad, so I am going to give it its own section.

2.3 The Bind and Fmap Functions

Currently, to operate on a value inside of an Option Monad, we need to manually unwrap it first. Instead of having to do this every time, we can instead define a new method on our Option Monad to do this for us.

```
    # function operates on the value inside of our Option Monad
    def fmap(self, function):
        if self.is_none():
            # self is an Option Monad
            return self

        val = self.unwrap()
        newval = function(val)
        # We create a new Monad here to surround the new value
        return Option.some(newval)
    # fmap returns an Option Monad.
```

The fmap function is a higher order function. That means it is a function that takes another function as an argument, and does something with that function. If you look in the end of the previous section, in order to pass the result of `index` into `division`, we had to use this code as boilerplate; and if we wanted to keep passing our value through more and more functions, we would have to continue repeating this block of code.

```
    res = index(ls, i)
    if res.is_none():
        return Option.none()
```

We were checking whether a function had successfully computed a value or whether an error had occurred. If the value existed, we later passed that value into a function. If the value did not exist, then we simply returned the indication of failure, an `Option.none()` object.

Looking at bind, we can see it performs a similar operation. `res.fmap(function)` checks whether or not `res` is a successfully computed value, in which case it passes that value into `function`, or if it is a failed computation, in which case it simply returns itself, passing the failed computation forward.

But then it wraps the return value of the function into an Option Monad. Why? Well, for starters, it's just consistent. We want to be able to predict that `fmap`

will return an Option Monad, as opposed to having to check the type every time it is returned. Secondly, it enables us to chain the `fmap` operation multiple times.

```
Option.some(-62).fmap(abs).fmap(chr)
# Some('<')
```

The above code is an example of chaining `fmap`; we start with some option value, `Some(-62)`, and we `fmap` the `abs` function, which computes the absolute values, and the `chr` function, which turns a number into its corresponding character. In this case, `62` corresponds to `'<'`.

At a higher level, `fmap` unwraps the value in our Monad, and passes that value through a function. However, `fmap` does not do so blindly. It takes care to maintain all of the Monad's internal context for the value. In this case, that context is simple; all `fmap` has to do is return early with Nothing if the Monad `fmap` is called on is Nothing, and wrap the result of the function back into an Option Monad otherwise.

However, we can't quite use this function to fix the problem we had earlier. Let's see what happens if we try to use `fmap` in that case:

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def inverse_element(ls, i):
    res = index(ls, i)
    return res.fmap(lambda x: division(1,x))
```

Aside: In Python, you can create a simple one-line function by writing `lambda` followed by the list of arguments, then a colon, and then an expression which will be returned by the function.

```
add = lambda x, y: x + y
add(2,3) #5

add_three = lambda x: 3 + x
add_three(7) # 10
```

Our above solution using `fmap` looks a lot nicer, but it doesn't quite work. We no longer have to do any manual unwrapping, but if we run this, we get a weird result:

```
root_element([1,2,3],1)
# Some(Some(1.4142135623730951))
```

Instead of having what we want, which is `Some(1.4142135623730951)`, we have our value wrapped in an extra Monadic layer. This is because our `root` function returns a Monad, and `fmap` wraps the result of `root` in a Monad. This is an annoying problem, and we can write a function to flatten it if we want, but instead, we usually write another function; `bind`.

```
    # function returns an Option Monad
    def bind(self, function):
        if self.is_none():
            # self is an Option Monad
            return self

        val = self.unwrap()
        # function returns an Option Monad
        return function(val)
    # bind returns an Option Monad.
```

`bind` does essentially the same thing as `fmap`, but we expect that the function we pass to `bind` to be a monadic function; it needs to return a Monad. If we do this, we can use `bind` and `fmap` to chain function application on an Option Monad, and we can rewrite our above code again.

```
import math

def root(x):
    if x < 0:
        return Option.none()
    return Option.some(math.sqrt(x))

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def root_element(ls, i):
    return index(ls, i).bind(root)
```

Now, we finally have a worthy example of how to use Option Monads. We have written two functions which use Option Monads to handle errors, and when we want to write a new function that uses both of those functions, we can completely ignore checking for errors or unwrapping values; we just use `bind` and let the Option Monad handle everything.

You may still think this sort of thing is useless; and in python, for such a simple

example, it kinda is! But as we continue to explore Monads, we will encounter some examples that get more and more complex without Monads, but that Monads make simpler. Oh hey look, that's the next section.

2.4 A More Complex Example

In order to give a more illustrative example of where the Option Monad can be more useful, consider the following problem; open a file, and read the first whitespace-terminated word from the beginning of the file, and parse it into an integer if possible. This problem is fairly easy to do with built-in Python functions, but the Option Monad can make error handling easier. However, none of Python's built-in functions use the Option Monad, so we will have to rewrite them so that they do. In languages with the Option Monad as a star player, such as Rust, Haskell, or Scala, this is not an issue.

```
def option_open(filename, mode='r'):
    try:
        fd = Option.some(open(filename, mode=mode))
    except Exception:
        fd = Option.none()
    return fd

def option_read(fd):
    try:
        data = Option.some(fd.read())
    except Exception:
        data = Option.none()
    return data

import re

def option_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        match = Option.some(match)
    else:
        match = Option.none()
    return match

def option_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None
```

```
    if g == None:
        g = Option.none()
    else:
        g = Option.some(g)

    return g

def option_int(s):
    try:
        i = Option.some(int(s))
    except Exception:
        i = Option.none()

    return i
```

These functions perform the exact same operations as their Python counterparts, but they return Some if the computation succeeds and Nothing if it fails, instead of throwing an error or using some other return code. This will allow us to use `bind` to chain these functions together.

```
result = (
    # We create a new Option Monad holding the string 'text.txt'
    Option.some('text.txt')
    # We then bind our file opening function.
    # The value returned from this function is
    # an Option Monad holding a file object (or Nothing)
    .bind(option_open)
    # We then bind our file reading function
    # the value returned from this function is an
    # Option Monad holding the contents of the file (or Nothing)
    .bind(option_read)
    # We then bind a lambda to match the first word in the file.
    # the value returned from this function is an
    # Option Monad holding a regex match object (or Nothing)
    .bind(lambda x: option_match(r'\s*(\S*)', x))
    # We then bind a lambda to get the string from the match object.
    # the value returned from this function is an
    # Option Monad holding the first word in the file (or Nothing)
    .bind(lambda x: option_get_group(x, 1))
    # We then bind our integer casting function.
    # the value returned from this function is an
    # Option Monad holding the integer it is cast to (or Nothing)
    .bind(option_int))
```

Side note: although whitespace is significant in Python, it is ignored inside of parenthesis, so if you ever need to split an expression onto multiple lines, you

can surround it in parenthesis, as I have above.

This code opens a text file, reads the entire file from it, looks at the first word in the file, and tries to read it in as an integer. Using the Option Monad is useful because if an error happens at any time during the computation, it just passes a Nothing Option through the rest of the bind functions.

We can make this look cooler by choosing an infix operator to overload. By convention, `>>=` is used, but that's hard to to do in python, so we are going to use `>>`, a right shift. We can override that operator in python with the following code:

```
def __rshift__(self, function):
    return self.bind(function)
```

Now, let's rewrite the above option code as the following.

```
result = (
    Option.some('text.txt')
      >> option_open
      >> option_read
      >> (lambda x: option_match(r'\s*(\S*)', x))
      >> (lambda x: option_get_group(x, 1))
      >> option_int
    )
```

Consider the same operation in regular Python. We can write it in one expression, in which case this is impossible to read, or we can split it up, over many lines, creating a bunch of temporary variables that we use once and then never again.

```
# one expression
result = int(
        re.match(
            r'\s*(\S*)',
            open('text.txt').read()
        ).group(1)
    )

# with temporary variables
temp1 = open('text.txt').read()
temp2 = re.match(r'\s*(\S*)', temp1).group(1)
result = int(temp2)
```

The first example is unreadable. The functions used have no meaningful order, so it becomes an act of mental gymnastics to figure out what happens when. The functions appear in the order `int`, `match`, `open`, `read`, and `group`. `int` comes first, despite being called last, and `open`, the first function to be called, appears randomly in the middle.

The second example is the shortest version where all the functions appear in the

source code in the order they are called, and so it is pretty readable, but once again we've got the problem of errors!

This code could fail if the wrong information is passed into it, and it can fail in approximately 5 places. Even worse, because `.group()` can return `None` if it fails, and `int(None) == 0`, you can get a wrong answer from this code without an error being thrown. In order to guarantee that this code doesn't fail, we would have to rewrite it, and a solution like this would be necessary in Python.

```
try:
    temp1 = open('text.txt').read()
    temp2 = re.match(r'\s*(\S*)', temp1).group(1)
    if temp2 == None:
        result = None
    else:
        result = int(temp2)
except Exception:
    result = None
```

The above code now won't throw any errors or produce erroneous results, and will set `result` to `None` if the code fails. Now, compare that safe version without Monads to the safe version with Monads.

```
result = (
    Option.some('text.txt')
      >> option_open
      >> option_read
      >> (lambda x: option_match(r'\s*(\S*)', x))
      >> (lambda x: option_get_group(x, 1))
      >> option_int
    )
```

The Monad version is simpler, just as safe, and even a line shorter (two if you don't count the line with a single closing parenthesis). Not to mention cooler and more elegant by far.

2.5 The Result Monad

There is one major problem with the Option monad above; if our code fails, we have no way to know how or when. With the standard python example, we could print an error message to the screen or to a file that would let us know what kind of error occurred. There is another Monad, called the Result Monad, that allows us to do just that while still having the power of the Option Monad. I won't reproduce the entire code here (it is in the appendix) but I will go over a few of the changes, as we will use the Result Monad in the next section.

The Result Monad uses slightly different names; a value is Ok if it is a successful computation, and it is an Error if the computation has failed. The functions

that check this status are `self.is_ok()` and `self.is_error()`. `self.is_ok()` returns `True` if there is a value, and `False` if there is an error message. `self.is_error()` does the opposite.

```
class Result:

    def __init__(self, failed, value, message):
        self._failed = failed
        self._message = message
        self._value = value
```

Our `__init__` function now takes an additional argument; an error message. Now, we have a value that indicates whether or not our computation has failed, a value that stores the result of the computation (if the computation succeeded), and a value that stores the error message (if the computation failed).

In order to access the error message, we add a new function like `unwrap` from the Option Monad. `unwrap` still exists and behaves in about the same way.

```
    def error_msg(self):
        if self.is_error():
            return self._message
        else:
            raise Exception('This Result is Ok')
```

This function checks whether or not we have failed, and returns the error message if it is an Error. Just like `unwrap`, it throws an Exception if there is no error message.

`bind` and `fmap` have not changed at all, but it is nice to note that when you return `self` in the case of the error, the error message stays the same. This means that when we chain multiple `bind` and `fmap` calls together, the first one that fails will have its error message propagate through till the end.

```
    # function operates on the value in the monad
    def fmap(self, function):
        if self.is_error():
            # self is a Result Monad
            return self

        val = self.unwrap()
        # function(val) is a value, so we have to wrap it
        return Result.ok(function(val))
    # bind returns a Result Monad

    # function returns a Result Monad
    def bind(self, function):
        if self.is_error():
            # self is a Result Monad
```

```
            return self

        val = self.unwrap()
        # function(val) is a Result Monad
        return function(val)
    # bind returns a Result Monad
```

I'm also adding another function that is sort of like `bind`, but instead, provides a simple way for Monadic computations to check for errors, and if they've occurred, to replace the computed value with a default value. Instead of applying the function pass into `recover` to the value within the result, we call the function with no arguments, and it will return (by definition) a new Result Monad

```
    def recover(self, function):
        if self.is_error():
            return function()

        return self
```

For example, if you wanted to write an app that read settings from a configuration file, you could write a function that did so, returning an Result Monad. Then you could `recover` with a function that provided the default settings for your app. That would mean that if everything worked fine, you would run your app with the settings from the file; however, if the file was missing, if parsing failed, or if some other error occurred, the recover function would automatically be called and the default value would be returned.

```
config = (
    result_open('config.txt')
      >> result_read
      >> result_parse
    ).recover(lambda: Result.ok(10))
```

The equivalent in Python without Monads would be the following.

```
try:
    temp1 = open('config.txt').read()
    config = parse(temp1)
except Exception:
    config = 10
```

We've already seen the following constructors used above, but for completeness here are the two constructors for the Result Monad.

```
    @classmethod
    def ok(cls, val):
        return cls(False, val, None)

    @classmethod
```

```
    def error(cls, msg):
        return cls(True, None, msg)
```

3 A Parsing Monad

We're now going to talk about a Monad called the Parsing Combinator, which basically recreates and improves upon regular expressions, using Monads. Here we see the truth of the oft said but little understood aphorism that Monads represent computation; here, they represent the computation of regular expressions.

It's also worth noting that there are many python Parsing Combinator libraries on the python package index, and those will have faster, more powerful implementations than the one below.

3.1 Why?

The Parsing Combinator is going to seem a little pointless until we get to the examples, so I'd like to try and motivate it a bit. Parsing Combinators are, on the surface, similar to regular expressions, but they are more powerful in a few important ways.

First of all, while basic regular expressions can only handle regular languages, Parsing Combinators can match context sensitive languages. However, this is rarely important in practice, because most "regular expressions" in modern languages are likewise extended to be able to parse more complex languages.

Secondly, Parsing Combinators are more readable than regular expressions. Although regular expressions excel at concise and quick patterns, they can quickly become hard to read, like the following regular expression for floating point numbers: `[-+]? [0-9]* \.? [0-9]+ ([eE] [-+]? [0-9]+)?`. While this is concise and accurate, it is nevertheless hard to read, and impossible to understand if you don't already know regular expressions. And the example above is fairly simple.

Thirdly, Parsing Combinators allow for more than simple matching and finding substrings. Parsing Combinators allow you to transform the result of a match without leaving the Parsing Combinator; for instance, we could write a Parsing Combinator that found IP addresses in a text file, and then resolved the IP addresses into domain names, and then returned a list of the domain names, instead of simply returning the strings that matched.

Fourthly, Parsing Combinators can take advantage of compile time type checking in typed languages. While this isn't possible in Python, it is in many other languages. For example, the regular expression `[` is invalid; but most statically typed languages can't determine that it is invalid until runtime. Parsing

Combinators, on the other hand, don't throw runtime exceptions like that; the information they encode is within the language itself, and therefore, they are checked at compile time to be valid.

Fifthly, they can be really fast. It's rare that a high level, very abstract language can claim to be as fast as low level languages, but there are cases when parsers written with the Parser Combinator Monad can rival or even beat the speed of parsers written in custom C. Obviously, the one I wrote for this next section is optimized for readability and not speed, but the interface is basically the same.

3.2 The Code

Unlike the Option Monad, I'm not going to go over the entire codebase for the Parsing Combinator. I hope to provide enough context so that the inner workings aren't mysterious, but there is a lot of code (all in the appendix) and it is frankly quite dull at times.

```
class Parser:
    def __init__(self, function):
        # function takes a string to be parsed and returns a Result Monad
        # holding a tuple, holding (already_parsed_value, remainder_of_string)
        # or an Error
        self._function = function

    def __call__(self, text):
        return self._function(text)
```

The basic idea behind our Parsing Combinator is that it represents a function that takes an input text in, and outputs a Result Monad holding either an error, or a tuple containing the 'matched value' and the 'remaining text'. For example, a function that behaves like this would be:

```
def parse_hi(text):
    try:
        if text[0] == 'h' and text[1] == 'i':
            res = Result.ok(('hi', text[2:]))
        else:
            res = Result.error('The first two characters '
                               'were not "hi".')
    except IndexError:
        res = Result.error('The string is too short to '
                           'contain "hi".')
    return res
```

However, we want to use certain Monadic ideas like `bind` and `fmap` with our parser, so instead of just using functions like the above, we're going to wrap them with the `Parser` class, using `Parser(parse_hi)`. In order to still be able

to call our function, we're going to implement the `__call__` method, which allows instances of our class to be called like functions. I've provided a small example of that below.

```
class DefaultPrinter:
    def __init__(self, text):
        self.text = text
    def __call__(self, arg=None):
        if arg is None:
            print(self.text)
        else:
            print(arg)

print_hi = DefaultPrinter('hi')
print_hi() # prints 'hi'
print_hi('other text') # prints 'other text'
```

One of the ways we are going to interact with the Parsing Combinator is by combining a bunch of simple Parsers into one, large parser. As an example of a simple parser, let's look at one of the constructors for our parser.

```
# char is a single character
# text is a string
def match_char(char, text):
    try:
        current = text[0]
    except IndexError:
        return Result.error('End of String encountered, but ' +
            '{} is still expected'.format(repr(char)))

    if current == char:
        return Result.ok(
            (text[0],   # the character matched
             text[1:])  # the remainder of the text
        )
    else:
        return Result.error('Failed to match character {} at {}'
            .format(repr(char), repr(text)))
```

The above function takes a character and a text, checks whether the character is the first element of the text, and if it is, returns the matched character and the rest of the string as a tuple in a Result, and if not, it returns a error message in a Result. We can use this function to define an alternate constructor for the parser now:

```
    @classmethod
    def char(cls, char):
        # char represents the character that our parser is going to match
```

```
        return Parser(lambda text: match_char(char, text))
```

The function we pass into the `Parser` constructor takes a single input, `text`, which will then be passed into `match_char` to be matched against the character we passed into the constructor (`char`).

We'll encounter other basic constructors later; for now, this will be enough to do a few basic examples.

In the next section, I am going to do something that isn't going to reflect how you would actually use monads. I'm going to use `bindp` for the `bind` function of a Parser Monad, and `bindr` for the `bind` function of a Result Monad.

To impress upon you how weird this is, and why you would never do it in actual code, is that `bind` represents a common interface all monads have; it would be like naming the addition and multiplication operators something different for `unsigned int`s, `signed int`s, `long`s, and `float`s.

However, it's very hard to read for the first time if all the binds look the same.

```
    # function takes a value, and returns a Result monad holding either the
    # result of that function, or an error message.
    def bindp(self, function):

        # returned holds a tuple holding
        # (currently matched value, remainder of source text)
        def inner_function(returned):
            match = returned[0]
            remainder = returned[1]
            return function(match)
                .bindr(lambda x: Result.ok((x, remainder)))

        return Parser(lambda text: self(text).bindr(inner))
```

`bindp` is, of course, the star of the show. From a high level, it is going to take a function (that might fail), and apply that function to the matched value of the Parser.

Let's break it down step by step. First, the `inner_function`:

- `inner_function` takes a tuple containing the matched value and the remainder of the text. It's going to mostly ignore the remainder of the text.
- Then, it applies the function (passed to `bindp`) to the matched value, transforming it into some other value. However, because it might fail, it returns a Result Monad holding that other value.
- We want to return a Result Monad holding a tuple containing that other value and the remainder of the text, but the other value is in a Monad, so we need to either (a) manually pull it out, or (b) use bind.

- We can use bind with `lambda x: Result.ok((x, remainder))` to pull the value x out of the Result Monad, and stick it back into a new Result Monad as the first member of the tuple we want.

If we recall, `fmap` is very similar to `bind`, with the exception that the function passed to `fmap` is one we know isn't going to fail, and therefore doesn't return a Result Monad, just a simple value. Therefore, it requires slightly different handling. Luckily, since we know it won't fail, we can make it into a function that does return a Result Monad by simply wrapping it's return value in an `Result.Ok` monad.

```
    def fmap(self, function):
        return self.bindp(lambda x: Result.ok(function(x)))
```

We really don't have enough information yet to do anything interesting, but I'll still try to demonstrate an example of how `bind` or `fmap` might work.

```
# Here is a function that matches any character
def any_char(text):
    if len(text) == 0:
        return Result.error("The text is too short"
                            " to contain any character")
    else:
        return Result.ok((text[0], text[1:]))

any_char_parser = Parser(any_char)

print(any_char_parser(''))
# Result.Error("The text is too short to contain any character")

print(any_char_parser('hello'))
# Result.Ok(('h','ello'))
```

If I wanted to transform this value, I could define a new parser by `fmap`ing the `ord` function over it. `ord` is a built in python function that takes a single character and returns the ascii or unicode number it is associated with.

```
ord_parser = any_char_parser.fmap(ord)

print(ord_parser(''))
# Result.Error("The text is too short to contain any character")

print(ord_parser('hello'))
# Ok((104, 'ello'))
```

As we can see above, `fmap` has altered the parsed value without altering the remainder of the text to be parsed.

Now that we've seen a way to construct this Monad, and how its `bind` and `fmap`

functions work, let's look at how to combine simple Parsing Combinators into more complex ones.

```
    # self is a parser
    # other is a parser
    # function takes two values, and returns a new value
    def combine(self, other, function):

        def combine_func(returned):
            match = returned[0] # the matched value from self
            rest = returned[1] # the remaining text from self
            res = other(rest) # pass the remaining text through other

            # I could rewrite this using bind, but it
            # just results in harder to read code, which is what
            # we are trying to avoid.
            if res.is_ok():
                other_match, other_rest = res.unwrap()

                # we use function to combine the two matches into
                # a new match, and then we put the new match back into
                # context with the remaining text.
                new_match = function(match, other_match)
                return Result.ok((new_match, other_rest))
            else:
                return res

        return (
              Parser(lambda text: self(text)
                .bindr( lambda res: combine_func(res) )
              )
            )
```

`combine` is the most useful function for us; it takes two parsers (`self` and `other`) and a function. It creates a new parser that first executes `self` on the input text, and then it executes `other` on the remaining text after `self`'s match. Then it uses `function` to combine the two matches, and returns that combination along with the remaining text from `other`'s match.

Once we have `combine` defined, we're going to create a bunch of other similar functions by calling `combine` with a default function to combine the two values.

```
    def concat(self, other):
        # think string concatenation for this plus, not
        # addition of numbers.
        return self.combine(other, lambda x,y: x + y)
```

```
    def first(self, other):
        return self.combine(other, lambda x,y: x)

    def last(self, other):
        return self.combine(other, lambda x,y: y)

    def tuple(self, other):
        return self.combine(other, lambda x,y: (x,y))
```

These functions are the most common choices for what you might want to use to combine the two matches. Most common is `concat`, which simply concatenates the two strings; this is the default regex behavior. Then there is `first` and `last`, which instead of combining the two values, instead discard one of the values. We will see later why this is useful. Finally, we can put the two values into a tuple instead of concatenating them, which is mostly useful when used in conjunction with `bind`. However, I've provided a few examples of working with these functions.

```
match_a = Parser.char('a')
match_b = Parser.char('b')

# This will match the string 'ab' and return the string 'ab'
match_a.concat(match_b)

# This will match the string 'ab' and return the string 'a'
match_a.first(match_b)

# This will match the string 'ab' and return the string 'b'
match_a.last(match_b)

# This will match the string 'ab' and return the tuple ('a','b')
match_a.tuple(match_b)
```

We also have a few other important functions that we use to build more complex parsers. I haven't included their code here, because I think that understanding `combine` is the most important code here, and I don't want to get bogged down in too many details.

`choice` implements the ability to try one parser, and if it fails, recover by trying another. Be careful with this one, because it only operates locally. If you try and match **x** or **y**, and **x** succeeds in matching the text but puts you in a corner that causes failure later on, it won't backtrack and try **y**. It will only backtrack and try **y** if matching **x** fails. It is possible to alter this function so that it does backtrack, but it tends to not be required for writing powerful parsers.

`many` is perhaps the most complicated constructor. What we want to do is continually match one parser, using a function to combine the results, until

a failure occurs; but when a failure occurs, we want to ignore the failure and return that previous match. We also implement `many1` that matches one or more examples of an object, not zero or more like `many`.

There is also a variant of `many` called `many_list`, which instead of combining them by concatenating them, like `many` does, simply collects a list of all of the many matches.

`optional` tries to match the input text with `self`, but if it fails, it matches nothing and returns the empty string as the match, and the whole text as the remainder.

Finally, we add symbolic versions of many of the above functions. This is purely for ease of reading the expressions we will write; you will see that they can get pretty complex, and `this.concat(that)` is less elegant than `this + that`.

Here is a full table of which functions I have bound to which symbols:

symbol	function	description
`+`	`concat`	concatenates the parsed values
`<=`	`first`	returns the parsed value of the first parser
`>=`	`last`	returns the parsed value of the last parser
`&`	`tuple`	returns the parsed value of both parsers in a tuple
`>>`	`bind`	applies the function (which might fail) to the parsed value
`>`	`fmap`	applies the function to the parsed value
`|`	`choice`	tries the first parser, and if it fails, tries the second

There are also a few more alternate constructors available, so here's a list of them and a brief description of what they do.

constructor	description
`char`	Matches the character that is passed into it
`oneof`	Matches any one of the characters passed in as a string or list
`empty`	Matches nothing
`noneof`	Matches any character not passed in as a string or list

In order to use a parser, all we have to do is call it on some input. However, this will return a Result holding the matched value, remainder of text pair. I've also written a Parser method that will help streamline some parsing cases. Our normal parsing combinator doesn't care if it has reached the end of the input; if you have a parser that parses numbers, and you ask it to parse `'99 bottles of beer on the wall'`, it will happily parse the `99` and ignore the rest of the string; usually, we want the entire match, not just the first part. This function causes the result to be an error if the entire string isn't matched.

```
    def parse_total(self, string):

        def check_full(tup):
            # if there is no remaining text, return our matched value
            if tup[1] == '':
                return Result.ok(tup[0])
            # otherwise, there is an Error
            else:
                return Result.error('The match did not consist of the entire ' +
                    'string: {} was left over'.format(repr(tup[1])))

        return self(string).bindr(check_full)
```

If you want to look at more of the code for context, it is included in the appendices of this paper.

3.3 Using the Parser Combinator

Now we have a powerful enough Parser Monad to recreate all of the flexibility and power of regular expressions. We can combine parsers to make more complex ones, and we can transform the values within parsers, so let's try parsing something simple; let's try and write a Parser Combinator that parses numbers.

A number can be as simple as `12` or as complex as `12.00123e45`, so we're going to need to build up a complex parser. Let's start with creating a parser that parses 1 or more consecutive digits.

```
digits = Parser.oneof('0123456789').many1()
```

Now, we need to express an optional decimal place, followed by one of more digits. Remember, the `+` operator will use the first parser, and then the second parser, and concatenate their results.

```
decimal = (Parser.char('.') + digits).optional()
```

Now, we need an exponent part, which is pretty simple given the above. However, we do need one additional component, a sign. The `|` operator will parse one or the other of the two things on either side of it.

```
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = (Parser.oneof('eE') + sign + digits).optional()
```

Finally, we can put these four together to get:

```
number = sign + digits + decimal + exponent
```

Here's our finished code, and it's results on some possible inputs:

```
digits = Parser.oneof('0123456789').many1()
decimal = (Parser.char('.') + digits).optional()
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = (Parser.oneof('eE') + sign + digits).optional()
number = sign + digits + decimal + exponent

Parser.parse_total(number, '12')
    # Ok('12')
Parser.parse_total(number, '12e10')
    # Ok('12e10')
Parser.parse_total(number, '2.12345e100')
    # Ok('2.12345e100')
Parser.parse_total(number, 'hello world')
    # Error(Failed to match one of
    # ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
Parser.parse_total(number, '99 bottles of beer on the wall')
    # Error(The match did not consist of the entire string:
    # ' bottles of beer on the wall' was left over)
Parser.parse_total(number, '')
    # Error(End of String encountered)
```

Furthermore, we can use a result-oriented version of python's `float` function, along with our `bind` operator, to cast these results to be floats instead of strings.

```
def result_float(x):
    try:
        return Result.ok(float(x))
    except Exception:
        return Result.error('Failed to cast to a float')

Parser.parse_total(number, '7.22345e10') >> result_float
    # Ok(72234500000.0)
```

3.4 Going Further

Everything we've done so far can basically be done in the exact same way by regular expressions. However, because of the way we've defined the Parsing Combinator, we can use `bind` and `fmap` inside of Parsing Combinator expressions. This allows us to parse complex expressions into fully formed and entirely arbitrary python objects.

3.4.1 List of Numbers Parser

Before, we matched a string using our parser, producing a Result Monad. We then used `bind` to transform the value the Parser returned into an actual number. However, the Parser is also a Monad, so we can move the `bind` call into the parser itself, and it won't change the way it works.

```
number = (sign + digits + decimal + exponent) >> result_float

# whitespace consists of a space, a tab, or a newline
whitespace = Parser.oneof(' \t\n').many1()

# without infix operators
many_numbers = (
        (whitespace.optional().last(number))
    ).many_list()

# with infix operators
many_numbers = (
        (whitespace.optional() >= number)
    ).many_list()
```

Let me take this code and expand it into English. `many_numbers` translates to 'If there's any whitespace, match it and discard it, and parse a number following it. Then, put that number in a list, and repeat until parsing fails, appending new numbers to the list, and return that.'

```
text = '2.12345e+100 2.1e10 1223 13.5 100e100'
print(Parser.parse_total(many_numbers, text))
# Ok([2.12345e+100, 21000000000.0, 1223.0, 13.5, 1e+102])
```

3.4.2 CSV Parser

Parsing Combinators can be used to write powerful, modular, readable, and concise parsers for any format of text. One common way of representing values in text is the CSV file format; this format is used to represent tables in pure text. The columns are separated by commas, and the rows are separated by newlines. Below is a CSV parser in 5 lines of code.

```
expression = Parser.noneof(',\n').many1()
comma = Parser.char(',')
newline = Parser.char('\n')
line = (expression <= comma.optional()).many_list()
csv = (line <= newline.optional()).many_list()

# Example
```

```
text = '''1,2,3,4,5
hello world, my, good, friends, 5
0,1,2,3,4'''

print(csv.parse_total(text))
# Ok(
#     [
#         ['1', '2', '3', '4', '5'],
#         ['hello world', ' my', ' good', ' friends', ' 5'],
#         ['0', '1', '2', '3', '4']
#     ]
# )
```

First, we define a parser for the data we actually care about; the data we're talking about can be any text as long as it doesn't contain a newline or a comma.

Next, we create simple comma and newline parsers. This isn't really necessary; I just think it makes it easier to read.

Then, we will define a line parser to be an expression, followed by a comma, over and over. This Parser will match until it comes to a newline, at which point it will stop, because nothing in the Parser can deal with a newline.

In order to match the whole csv, we will match a line, followed by a newline character, over and over. And then when we pass some data into the CSV, it will parse it into a list of rows, each of which is a list of the values in those rows.

3.4.3 Abstract Syntax Tree Parser

This next section requires a bit of domain knowledge; we're going to parse basic mathematical expressions into something called an Abstract Syntax Tree, or AST. An AST is behind the interpreter or compiler for almost every language, so parsing them is a common thing to want to do. In our case, we are parsing a simple language of addition, subtraction, multiplication, and division, and using parenthesis to let our expression contain a smaller expression.

The parser that we will construct is going to return a tree like recursive data structure in Python, holding the full structure of the text we pass in. In order to do this recursively, I will not be able to do everything with the constructors and combination functions; instead, I will have to write a new function and pass that explicitly to the default Parser constructor.

```
from enum import Enum, auto

# This is just a basic Enumeration in Python
class Op(Enum):
    PLUS = auto()
```

```
    MINUS = auto()
    TIMES = auto()
    DIV = auto()

class Expr:
    def __init__(expr1, op, expr2):
        self.expr1 = expr1
        self.expr2 = expr2

        if op == '+':
            self.op = Op.PLUS
        elif op == '-':
            self.op = Op.MINUS
        elif op == '*':
            self.op = Op.TIMES
        elif op == '/':
            self.op = Op.DIV
        else:
            self.op = op

    def __repr__(self):
        return ("Expr({}, {}, {})"
            .format(self.expr1, self.op, self.expr2))

# this function surrounds a parser with optional whitespace
def pad(parser):
    return (whitespace.optional() >= parser) <= whitespace.optional()

# we create a bunch of symbols for our parser which
# all can be surrounded by whitespace
openp = pad(Parser.char('('))
closep = pad(Parser.char(')'))

plus = pad(Parser.char('+'))
minus = pad(Parser.char('-'))
times = pad(Parser.char('*'))
div = pad(Parser.char('/'))
operator = plus | minus | times | div

# this function surrounds a parser with a pair of parens
def surround(parser):
    return (openp >= parser) <= closep

def expr(text):
    recursive = Parser(expr)
```

```
    expression = surround(
        (recursive | number)
        & operator
        & (recursive | number)
    )

    # things parsed by expression will have the slightly ugly form
    # of Ok(((a,b),c)). To transform that into an Ok(Expr)
    # we will define the following function:
    weird_func = lambda weird_tuple: Expr(
                        weird_tuple[0][0],
                        weird_tuple[0][1],
                        weird_tuple[1]
                    )

    # we use fmap to apply the above function to
    # the matched value of the parser
    full = expression > weird_func

    return full(text)

text = '((1+2) * (9 - 11))'

print(Parser(expr).parse_total(text))
# Ok(
#     Expr(
#         Expr(
#             1.0,
#             Op.PLUS,
#             2.0
#         ),
#         Op.TIMES,
#         Expr(
#             9.0,
#             Op.MINUS,
#             11.0
#         )
#     )
# )
```

I decided to reproduce this sort of code using standard regular expressions; Not only did my regex version have twice as many lines, it was a much more fragile program. I didn't thoroughly test it, but I didn't even bother adding error checking if stuff went wrong; I just assumed everything would go right. Furthermore, there were nested loops, plenty of functions, and all in all complex,

messy, hard to read code. The Parsing Combinator above, however, is short and will always return an Error Result with a sensible error message if an error happens.

4 Theory of Monads

Now that we've seen a few examples of what a Monad is, we can talk about the formal definition. This is going to be the most abstract section of the text, but I'll try and keep any statements from category theory or abstract algebra from appearing here.

4.1 Defining Monads

Monads are a special type of object that contains with in it a value and a context. Our Option and Result Monads contained the result of a computation, along with contextual information regarding whether the operation had succeeded or failed. This allowed us to write programs that could detect failure elegantly. Our Parsing Monad contained the result of the parsing so far, as well as the rest of the text remaining to be parsed. Whenever you see a Monad, you can sum up its operation by asking "What is the value in this Monad, and what is the context?". Many people, when confronted with Monads, want a way to get the value out of the Monad. But this causes problems, because you've taken the value out of context, and it becomes significantly more useless.

In order to interact with the values without taking them out of their context, we have a function called `fmap`. `fmap` takes the value, and applies the function to that value, and puts the result of the function back into context. In our Option and Result Monads, it applied the function to the value if our computation was successful, or it simply bypassed the function if our computation had failed. In our parsing combinator, it applied the function to the result of our computation, while leaving the remainder of the text to be parsed alone.

However, this meant that we couldn't use functions on the values in our Monad if the functions themselves returned Monads. For our Option and Result Monads, that means that we couldn't use a function that could fail (a function that returned an Option or Result Monad) on our Monad with `fmap`. If we did that, we could end up with a recursive Monad, like `Some(None)` or `Some(Some(3))`. This is annoying, and there's two ways to fix this weirdness.

The first way, which we used in the previous section, is to use a function called `bind`. `bind` is the same as `fmap`, but instead of putting the return value of the function passed to `bind` back into the same context, it expects that `bind` will return a new value and context (a Monad of some kind).

The second way is an alternative to `bind`; you don't need both, and it's more common to have `bind` as the one to use, so I haven't bothered talking about it yet. This function is called `join`, and it takes a recursive Monad and flattens it from two layers to one layer. For example:

```
Option.some(Option.some(x)).join() == Option.some(x)
Option.some(Option.none()).join() == Option.none()
Option.none().join() == Option.none()
```

We can show that `join` isn't any less useful than `bind` by actually writing `bind` using only `join` and `fmap`.

```
def bind(monad, function):
    return monad.fmap(function).join()
```

This does the same thing as `bind` usually does; it applies the function to the inside value if the Monad isn't Nothing, and then it returns Nothing if either the function returns or the Monad is Nothing, or it returns Some(value) if the function succeeds and the Monad had a value to pass into the function.

We can also show a way to write `fmap` and `join` solely using `bind`:

```
def fmap(monad, function):
    monad.bind(lambda x: Option.some(function(x)))

def join(monad):
    monad.bind(lambda x: x.unwrap() if x.is_some() else x)
```

This means, for our purposes, for something to be a Monad, we require it to either have both `fmap` *and* `join`, or just `bind`. However, it is common for Monads to have all three available.

4.2 Monad Laws

Now, Monads have three laws, or rules, they have to follow; this is just to make sure Monads don't have any unexpected behavior, but we should go over those rules anyway.

First of all, if you `fmap` over a Monad with the identity function (a function that returns its inputs unchanged), the value in the Monad doesn't change.

```
Monad(x).fmap(lambda x: x) == Monad(x)
```

Secondly, if you `fmap` two functions in a row, it should be the same as simply `fmap`ing the function which does the equivalent of those two functions in order.

```
m.fmap(lambda x: x+1).fmap(lambda x: x+2) == m.fmap(lambda x: x+3)
```

Finally, if you apply a function to a value and then stick it in a Monad, it is the same as putting that value in a Monad and `fmap`ing that function.

```
Monad(x).fmap(f) == Monad(f(x))
```

If you want, you can put these rules into equivalent forms using `bind` instead of `fmap`.

These might seem common sense, and if they are, that's good! The only reason that we require that these rules are followed is so that somebody doesn't create a Monad that behaves weirdly and it screws up our program. They basically boil down to "Monads should behave sensibly when you fmap or bind functions over them".

5 More Monad Examples

5.1 The Zeroth Monad

Our first section was titled 'Our First Monad'. However, we are computer scientists, and therefore we start counting at zero, not at one. So let's talk about another Monad that everyone reading this document has probably used, but never noticed that it was a Monad.

Lists.

How is a list a Monad? Well, from the previous section, a Monad is really just anything with a `bind` function, or with a `fmap` and a `join` function. And while not every programming language has these functions built in, we can easily write these functions for a list.

```
# python has a built in function, 'map' that does this.
def fmap(ls, function):
    new = []
    for item in ls:
        new.append(function(item))
    return new

# this is sometimes called 'flatten'
def join(ls):
    new = []
    for sublist in ls:
        for item in sublist:
            new.append(item)
    return new

# bind can be defined entirely with the other two!
```

```
def bind(ls, function):
    return join(fmap(ls, function))
```

This is all nice and well that we now have these functions, but it explains little conceptually. So let's try and describe Monads conceptually, and see how that can be applied to lists.

Our first Monads, the Option Monad and the Result Monad, both represented some sort of computation result that required more context than a simple value; in particular, they represented a computation result that could either succeed, producing a value, or fail, producing no meaningful value.

Lists can be thought of in a similar way; instead of representing either zero or one meaningful return value, lists can represent computations that can return zero, one, or any possible number of return values. For example, consider the following contrived example.

```
def abs_less_than(x):
    if x == 0:
        return []
    ls = [0]
    for i in range(1,x):
        ls.append(i)
        ls.append(-i)
    return ls

abs_less_than(1) # [0]
abs_less_than(3) # [0,-1,1,-2,2]

from math import sqrt

def sqrts(x):
    if x < 0:
        return []
    elif x == 0:
        return [0.0]
    else:
        return [sqrt(x), -sqrt(x)]

sqrts(4) # [-2, 2]
sqrts(-4) # []

bind(
    bind(
        # we make a monad holding 3
        [3],
        # and we bind abs_less_than over it
```

```
        abs_less_than
    ),
    # and then bind sqrts over it
    sqrts
)
# [0.0, 1.0, -1.0, 1.4142135623730951, -1.4142135623730951]
```

In this case, we execute two functions in series, getting all of the valid results to our question in one list; but the number of results isn't the same for all inputs, so we need a Monad to represent this computational uncertainty.

5.2 Promises (in Javascript)

As I was writing this essay, I started working on a project in Javascript that ended up using a thing called Promises. In Javascript, it's common to call a function that will perform some action, wait for something else to respond to the action, and then respond to that response. This is traditionally done with callbacks: passing a function into another function.

```
// setTimeout waits for 3000 ms, and then
// calls the function you passed it.
setTimeout(
    function(){
        alert("Hello");
    },
    3000
);
```

However, if the function you pass in needs to call another function that uses a callback, this can quickly lead to *callback hell*.

```
// don't bother trying to understand this;
// it's just an illustrative example
function handler () {
  // validateParams takes a function
  validateParams((err) => {
    if (err) { console.log('Error:', err); return }
    // dbQuery takes a function
    dbQuery((err, dbResults) => {
      if (err) { console.log('Error:', err); return }
      // serviceCall takes a function
      serviceCall(dbResults, (err, serviceResults) => {
        if (err) { console.log('Error:', err); return }
        //do something here!
      })
    })
```

```
  })
}
```

In modern javascript, there is a new feature: Promises! Promises intend to make the above code significantly easier to read; instead of passing in a function to provide a response, the function returns a Promise object you simply call the `then` method and pass in another function that returns a Promise object.

```
function handler (done) {
  // validateParams returns a promise
  return validateParams()
    .then(dbQuery)
    .then(serviceCall)
    .then( /* do something here! */ )
    .catch((err) => {
      console.log('Error:', err)
  })
}
```

Look familiar? That's right; Promises are essentially Result Monads with `bind` renamed as `then`, `recover` renamed as `catch`, and with a different underlying implementation (one that makes it impossible to define a `unwrap` operation, but then again, it's rare to actually need that). This is intentional. The people who designed Promises knew what Monads are, and knew how they could be used to fix a problem with Javascript.

6 Conclusion

HELP!

A Code: The Result Monad

```
class Result:
    def __init__(self, failed, value, message):
        self._failed = failed
        self._message = message
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Option.error({})'.format(repr(self._message))
        else:
            return 'Option.ok({})'.format(repr(self._value))

    def __str__(self):
        if self._failed:
            return 'Error({})'.format(self._message)
        else:
            return 'Ok({})'.format(self._value)

    def is_ok(self):
        if self._failed:
            return False
        return True

    def is_error(self):
        return not self.is_ok()

    def unwrap(self):
        if self.is_ok():
            return self._value
        else:
            raise Exception('This Result is an Error')

    def error_msg(self):
        if self.is_error():
            return self._message
        else:
            raise Exception('This Result is Ok')

    def bind(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
```

```
        return function(val)

    def fmap(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
        return Result.ok(function(val))

    def recover(self, function):
        if self.is_error():
            return function()

        return self

    def __rshift__(self, function):
        return self.bind(function)

    @classmethod
    def ok(cls, val):
        return cls(False, val, None)

    @classmethod
    def error(cls, msg):
        return cls(True, None, msg)

# The following are built in functions
# rewritten to work with the Result Monad

def result_open(filename, mode='r'):
    try:
        fd = Result.ok(open(filename, mode=mode))
    except Exception:
        fd = Result.error("Failed to open the file")
    return fd

def result_read(fd, size=-1):
    try:
        data = Result.ok(fd.read(size))
    except Exception:
        data = Result.error("Failed to read from the file")
    return data

import re
```

```
def result_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        match = Result.ok(match)
    else:
        match = Result.error("Failed to match the pattern")
    return match

def result_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        g = Result.error("Failed to get the group from the match")
    else:
        g = Result.ok(g)

    return g

def result_int(s):
    try:
        i = Result.ok(int(s))
    except Exception:
        i = Result.error("Failed to parse into an integer")
    return i


result = (
    Result.ok('text.txt')
      >> result_open
      >> result_read
      >> (lambda x: result_match(r'\s*(\S*)', x))
      >> (lambda x: result_get_group(x, 1))
      >> result_int
    )

print(result)
```

B Code: The Parsing Combinator

```
class Parser:
    def __init__(self, function):
        self._function = function

    def __call__(self, text):
        x = self._function(text)
        return x

    def __repr__(self):
        return '<Parsing Combinator>'

    def bind(self, function):

        def bind_func(result):
            return function(result[0]).bind(lambda x: Result.ok((x, result[1])))

        return Parser(lambda text: self(text).bind(bind_func))

    def fmap(self, function):
        return self.bind(lambda x: Result.ok(function(x)))

    def combine(self, other, function):

        def combine_func(match, rest):
            res = other(rest)
            if res.is_ok():
                other_match, rest = res.unwrap()
                new_match = function(match, other_match)
                return Result.ok((new_match, rest))
            else:
                return res

        return Parser(lambda text: self(text).bind(lambda res: combine_func(*res)))

    def concat(self, other):
        return self.combine(other, lambda x, y: x + y)

    def choice(self,other):

        def choice_func(text):
            return self(text).recover(lambda: other(text))

        return Parser(choice_func)
```

```
    def many(self, function=lambda x,y: x + y):

        def repeat_func(text):
            res = self(text)

            if res.is_error():
                return Result.ok(('',text))

            match = res.unwrap()[0]
            rest = res.unwrap()[1]

            res = self(rest)

            while res.is_ok():
                match = function(match, res.unwrap()[0])
                rest = res.unwrap()[1]
                res = self(rest)

            return Result.ok((match, rest))

        return Parser(repeat_func)

    def many_list(self):

        def repeat_func(text):
            res = self(text)

            if res.is_error():
                return Result.ok(('',text))

            match = [res.unwrap()[0]]
            rest = res.unwrap()[1]

            res = self(rest)

            while res.is_ok():
                match = match + [res.unwrap()[0]]
                rest = res.unwrap()[1]
                res = self(rest)

            return Result.ok((match, rest))

        return Parser(repeat_func)

    def many1(self, function=lambda x,y: x + y):
```

```
        return self.combine(self.many(function), function)

    def many1_list(self):
        return self.combine(self.many_list(function), function)

    def optional(self):
        return self | Parser.empty()

    def first(self, other):
        return self.combine(other, lambda x,y: x)

    def last(self, other):
        return self.combine(other, lambda x,y: y)

    def tuple(self, other):
        return self.combine(other, lambda x,y: (x,y))

    def __rshift__(self, function):
        return self.bind(function)

    def __gt__(self, function):
        return self.fmap(function)

    def __ge__(self, other):
        return self.last(other)

    def __le__(self, other):
        return self.first(other)

    def __add__(self, other):
        return self.concat(other)

    def __or__(self, other):
        return self.choice(other)

    def __and__(self, other):
        return self.tuple(other)

    @classmethod
    def char(cls, val):

        def match_char(text):
            try:
                current = text[0]
            except IndexError:
```

```
                return Result.error('End of String encountered, but ' +
                    '{} is still expected'.format(repr(val)))

            if current == val:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Failed to match character {} at {}'
                    .format(repr(val), repr(text)))

        return Parser(match_char)

    @classmethod
    def empty(cls):

        def match_empty(text):
            return Result.ok(('', text))

        return Parser(match_empty)

    @classmethod
    def oneof(cls, charls):

        def match_charls(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered, but one of ' +
                '{} is still expected'.format(list(charls)))

            if current in charls:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Failed to match one of {} at {}'
                    .format(list(charls), repr(text)))

        return Parser(match_charls)

    @classmethod
    def noneof(cls, charls):

        def none_charls(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered, but none of ' +
```

```
                '{} is still expected'.format(repr(text)))

            if current not in charls:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Found one of {} at {} '
                    +'when there should be none of'.format(list(charls), repr(text)))

        return Parser(none_charls)

    def parse_prefix(self, string):
        return self(string)

    def parse_total(self, string):

        def check_full(tup):
            if tup[1] == '':
                return Result.ok(tup[0])
            else:
                return Result.error('The match did not consist of the entire ' +
                    'string: {} was left over'.format(repr(tup[1])))

        return self(string) >> check_full
```