Monadic Parser Combinators

Or: How to Impress your Friends with Big Programming Words

Matthew McQuaid

Parser? I Hardly Know Her!

Parsing is a classic problem in computer science/software engineering. Essentially, a parser is a program which takes some input, usually text, and makes the text's *implicit* structure into *explicit* structure, usually something like a tree. A classic use case of parsers is turning program text into an abstract syntax tree. To illustrate, we consider the untyped λ-calculus.

In concrete syntax, we might write $\lambda x.x$ for the identity function, or $\lambda x.\lambda y.x$ for the K combinator. They're easy to write and read, but if we wanted to write an interpreter for our language, suddenly those text representations aren't so useful. If I have an expression like $(\lambda x.x)(\lambda y.y)$, I want to immediately know whether I'm dealing with an application or not, but this is not at all clear from the concrete syntax. In fact, we'd have to traverse the whole expression to figure this out. To solve this, we turn to abstract syntax. Here is one type of Abstract Syntax Tree (AST) for untyped λ-calculus:

```
datatype term = VAR of string
              | LAM of string * term
              | AP of term * term
```

In abstract syntax, $(\lambda x.x)(\lambda y.y)$ is written `AP(LAM("x",VAR "x"),LAM ("y",VAR "y"))`. The AST makes the structure of the term explicit: now we can do things like pattern match on terms. A parser for our language will transform a string of concrete syntax into an AST of type `term`. Our end goal for these notes will be to construct a fairly elegant and concise parser for the untyped λ-calculus

A Monad is like a ~~Burrito~~

So we want to write a parser. There are many, many techniques for writing parsers. As proves to frequently be the case in computer science, the fastest, most powerful parsers are complicated and difficult to implement. We're going to go with a less powerful, but pleasantly simple to implement technique called *monadic parser combinators*. The concept of parser combinators is that we provide a library of small combinators and primitive parsers that allow a user to construct more complex parsers through the use of said combinators. We will we accomplishing this by treating our parsers as a *monad*. Monads are notorious for being difficult to explain, and for the unending supply of blog posts and poor metaphors that try to explain them. A common metaphor is that a monad is like a container, or burrito, for some data. This has some intutive merit, but as we'll see, it fails to capture the full breadth of what a monad can be. To get straight to the point, a monad is just any type `m` that allows us to implement the following signature[1]:

```
signature MONAD =
sig
    type 'a m
    val return : 'a -> 'a m
    val >>= : 'a m -> ('a -> 'b m) -> 'b m
end
```

[1]There are a couple equivalent, alternative signatures

>>= is pronounced "bind". Some common datatypes are monads! Here are few examples:

```
infix >>=
structure ListM : MONAD =
struct
    type 'a m = 'a list
    val return = fn x => [x]
    fun x >>= f = List.concat (List.map f x)
end
stucture OptionM : MONAD =
struct
    type 'a m = 'a option
    val return = SOME
    fun x >>= f =
        case x of
          SOME a => f a
        | NONE => NONE
end
```

Essentially, we need a way to lift an `'a` into an `'a m`, and a way to extract that value so we can apply a function that creato it.

Parsers as Monads

Oh right, we wanted to treat parsers like monads. This is a perfect example of a case where the common metaphor of a monad as a container fails. A parser is a function: to call it a "container" would certainly be a stretch. What type should a parser be anyway? Well, it should take in a string, and then it should produce some kind of structured output.

If we're writing a parser for λ-calculus, a first try might be `type parser = string -> term`. But should we really constrain the type of a `parser` to return a `term`? There are many other things we might want to parse from a string. To this end, we'll parametrize the output type: `type 'a parser = string -> 'a`. Ok, but parsing can fail. What if I try to parse `"abc"` into an integer? We'll encode the possibility of failure using the option type: `type 'a parser = string -> 'a option`. We're getting pretty close now. The final concern is that sometimes we might want to parse only part of a string, leaving the rest to be handled by some other parser or function. So we'd like to return the remaining, unparsed text: `type 'a parser = string -> ('a * string) option`. The final change we'll make is to switch `string` to `char list`, because things like pattern matching to extract the first element of a string are much easier on lists, giving us: `type 'a parser = char list -> ('a * char list) option`. Now all that's left is to define `return` and `>>=`:

```
structure ParserM : MONAD =
struct
    type 'a parser = char list -> ('a * char list) option
    type 'a m = 'a parser

    fun return x = fn i => SOME (x,i)
    fun p >>= f =
        fn i => case p i of
                SOME (x,r) => f x r
              | NONE => NONE
end
```

These two functions are all we need to qualify a `parser` as monad, but we'll define some other basic functions to help us to start writing actually useful parsers:

```
fun zero _ = NONE
fun item [] = NONE
  | item (c::cs) = SOME (c,cs)
```

We can see that the two most basic parsers are one that always fails, and one that always succeeds, not processing its input at all, and returning some constant value. So
`return 1 ['a','b'] ==> SOME (1,['a','b'])`[2] and `zero ['a','b'] ==> NONE`. `item` is a `char parser` which consumes the first character of the list it's given, and leaves the rest unparsed. Then bind allows us to sequentially compose parsers. We can use it to write a parser which consumes a single character that satisfies a predicate:

```
(* sat : (char -> bool) -> char parser *)
fun sat p = item >>= (fn x => if p x then return x else zero)
```

Intuitively (and literally), the function which we bind `item` with is taking the thing that `item` parsed as input, and then returning a parser which is applied to the remaining, unparsed text.

A Set of Combinators

Now we set out to build a library of useful combinators that we can use to construct complex parsers. Our parsers work with `char lists`, but it's much easier to write `strings` in SML, and IO functions tend to return `strings`. For ease of use, we'll define a function that lets us apply a parser to a `string`:

```
infix %
fun p % s = p (String.explode s)
```

So we can sequentially compose parsers, but there might be situations where we'd like try two different parsers, and take the results of the first one that succeeds. To this end, we implement `++`:

```
infix ++
(* ++ : 'a parser * 'a parser -> 'a parser *)
fun p ++ q =
    fn i => case p i of
              NONE => q i
            | r => r
```

Now we can implement some parsers for single characters that satisfy some common properties:

```
fun in_range (x,y) c = x <= c andalso c <= y
val digit = sat (in_range ('0','9'))
val lower = sat (in_range ('a','z'))
val upper = sat (in_range ('A','Z'))
val alpha = lower ++ upper
val alphanum = digit ++ alpha
fun char c = sat (fn x => x = c)
```

[2] I use `'a'` instead of `#"a"` to indicate characters because it's easier to write

For convenience, we'll define a couple combinators that allow us to parse a character and then ignore it:

```
infix >> <<
fun p >> q = p >>= (fn _ => q)
fun p << q = p >> (fn x => q >> return x)
```

`>>` allows us to apply `p`, throw out the result, and then apply `q`. `<<` allows us to apply `p`, then apply `q`, throw out the result, and give back the result of `p`.

So far we've only implemented parsers that consume single characters. Surely we'll need some more complicated parsers. What if we want to parse an entire word from our input text?

```
(* word : char list parser *)
fun word [] = return []
  | word (c::cs) =
    char c  >>= (fn x =>
    word cs >>= (fn xs =>
    return (x::xs)
    ))
```

Given an empty word, we can parse an empty word out of any text, and given a non-empty word, we parse it's first character, then recursively parse the rest, and ultimately return the fully parsed out word. If you'd prefer to pass in a string and produce the parsed word as string, we can define

```
(* str : string parser *)
fun str s = (word (String.explode s)) >>= (return o String.implode)
```

There are many cases where we'd like apply a parser 1 or more times, and receive a list of all its output. Maybe we want consume and then ignore a bunch of whitespace in some program text, or we want to parse a list of digits out of a string representing a number (foreshadowing!). Let's define a combinator that does this:

```
(* many1 : 'a parser -> 'a list parser *)
fun many1 p =
    p                    >>= (fn x =>
    many1 p ++ return [] >>= (fn xs =>
    return (x::xs)
    ))
```

First, `p` must succeed at least once, then, we either parse a list of things from `p`, or if there is nothing left for `p` to parse, we return an empty list. Then, we combine our single parsed element with whatever else we were able to extract. A simple extension to apply a parser 0 or more times would be:

```
(* many : 'a parser -> 'a list parser *)
fun many p = many1 p ++ return []
```

Now we can implement a parser for positive integers!

```
(* nat : int parser *)
val nat =
let
    fun toNum c = Char.ord c - Char.ord '0'
    val eval = foldl (fn (c,i) => toNum c + 10 * i) 0
in
    many1 digit >>= (fn ds =>
    return (eval ds)
    )
end
```

What if we want to be able to parse negative numbers too?

```
(* int : int parser *)
val int =
(char '-' >> return ~) ++ return (Fn.id) >>= (fn f =>
nat                                      >>= (fn n =>
return (f n)
))
```

We're being a bit clever here. First we parse either a '-' or nothing, and from that we produce either the negation function, or the identity function, which we can then apply to the plain number parsed by `nat`.

Ok, we can parse a number, but what about a list of numbers? Generally a list is expressed in text by separating its elements using some separator character/string. In SML, we separate list elements with commas: "[1,2,3]". We also delimit them with brackets, but we'll get to that next. Now we implement a function that takes an element parser and a separator parser, and produces a parser for a non-empty series of separated elements:

```
(* sepby1 : 'a parser -> 'sep parser -> 'a list parser
fun sepby1 p sep =
    p >>=           (fn x =>
    many (sep >> p) (fn xs =>
    return (x::xs)
    ))
```

Once again we can extend this to potentially empty sequences:

```
fun sepby p sep = sepby1 p sep ++ return []
```

As mentioned, SML lists (and many other forms of sequence) are delimited by brackets. So we'll write a combinator to combine delimiter parsers with a parser for the delimited object:

```
(* delim : 'd1 parser -> 'a parser -> 'd2 parser -> 'a parser *)
fun delim d1 p d2 = d1 >> p << d2
```

Now we can write a parser for, say, a list of ints:

```
(* int_list : int list parser *)
val int_list =  delim (char '[') (sepby int (char ',')) (char ']')
```

Writing BNF Grammars with Parser Combinators

At this point, it's worth noting that our parser looks an awful lot like a BNF grammar. For instance, a BNF grammar for int lists might look something like this:

$$
\begin{aligned}
int &::= 1, -1, 2, -2, ... \\
seq &::= \cdot \mid int\ ,\ seq \\
list &::= [\ seq\ ]
\end{aligned}
$$

int corresponds to `int`, *seq* corresponds to `sepby int (char ',')`, and *list* corresponds to `int_list`. It's kind of amazing that we can get quite close to directly embedding the grammar of another language into SML! Can we take this further? What about the grammar for say, integer arithmetic with addition and subtraction?

$$
\begin{aligned}
int &::= 1, -1, 2, -2, ... \\
expr &::= expr\ addop\ factor \mid factor \\
addop &::= + \mid - \\
factor &::= int \mid (\ expr\ )
\end{aligned}
$$

We might try to express our grammar in SML as follows:

```
fun expr i =
    (expr   >>= (fn e =>
     addop  >>= (fn f =>
     factor =>> (fn x =>
     return (f(e,x))
    )))) i
and addop i = ((char '+' >> return op+) ++ (char '-' >> return op-)) i
and factor i = (int ++ delim (char '(') expr (char ')')) i
```

Note that this parser doesn't turn an arithmetic expression into an AST, it actually *evaluates* it. If we wished, we could easily modify it to produce an AST that some other program then evaluates. Unfortunately, there's a problem with these functions. `expr` will actually loop forever! It immediately calls itself, and has no base case. This is a problem that arises when trying to encode *left-recursive* grammars using parser combinators. A left-recursive grammar is one that describes an element of a language as being composed of an element of that language and some suffix. So when we say an *expr* can be an *expr* followed by some stuff, we are making our grammar left-recursive. There are a couple ways around this. There is an algorithm to remove left-recursion from any grammar that contains it, but it makes the grammar less clear, and harder to read. Instead, we'll use that fact that left-recursion can frequently be replaced with *iteration*. In this case, note that an *expr* is essentially a chain of *factor*s separated by an *addop*. Let's reformulate our parser to make use of this:

```
fun expr i =
    (factor                                                    >>= (fn x   =>
     many (addop >>= (fn f => factor >>= (fn y => return (f,y)))) >>= (fn fys =>
     return (foldl (fn ((f,y),z) => f(z,y)) x fys)
    ))) i
and addop i = ((char #"+" >> return op+) ++ (char #"-" >> return op-)) i
and factor i = (int ++ delim (char #"(") expr (char #")")) i
```

This is undeniably kind of ugly. This idea of replacing left-recursion with iteration seems pretty general, what if we define a combinator to describe these "chains"?

```
(* chainl1 : 'a parser -> ('a * 'a -> 'a parser) -> 'a parser
fun chainl1 terms ops =
    terms                                                    >>= (fn x   =>
    many (ops >>= (fn f => terms >>= (fn y => return (f,y)))) >>= (fn fys =>
    return (foldl (fn ((f,y),z) => f(z,y)) x fys)
    ))
```

The idea is essentially that we first parse a term, then we parse a possibly empty list of operators and terms, keeping them together in tuples, and finally we fold over the list, starting with the first term we parsed, combining each element together with each operator. Notably, `chainl1` parses operators as left associative, and does not accept empty chains. A potential inefficiency of this approach is that we first construct a list, and then fold over it. Could we just construct the answer directly?

```
fun chainl1 terms ops =
let
    fun rest x =
        ops   >>= (fn f =>
        terms >>= (fn y =>
        rest (f(x,y))
        )) ++ return x
in
    terms >>= rest
end
```

The `rest` function looks pretty weird. As before, we start by parsing at least one term, but then we bind it with `rest`. `rest` takes a term, and then tries to parse an operator and another term, which it then applies to our two terms, and then uses it recursively calculate the next term. If there are no more operators, it just returns the term we gave it. It's also possible to write a chain combinator for right associative operators, but we leave this as an exercise to the reader. Alright, now we can write our arithmetic parser as follows:

```
fun expr i = chainl1 factor addop i

and factor i = (int ++ delim (char '(') expr (char ')')) i

and addop i = ((char '+' >> return op+) ++ (char '-' >> return op-)) i
```

That looks pretty nice! Everything except for `expr` is basically straight out of our formal grammar, and `expr` isn't difficult to read.

Dealing with Whitespace Characters

As a final step before we write our λ-calculus parser, we'll write some combinators to handle leading and trailing whitespace characters:

```
(* consume : 'a parser -> unit parser *)
fun consume p = p >> return ()

(* whitespace : char parser *)
val whitespace = (char ' ') ++ (char '\t') ++ (char '\n')

(* ignore : unit parser *)
val ignore = consume (many whitespace)

(* pre : 'a parser -> 'a parser *)
fun pre p = ignore >> p

(* token : 'a parser -> 'a parser *)
fun token p = p << ignore
```

`consume` allows us to apply a parser, and then throw away the result, returning `unit` instead. `pre` is meant to handle any whitespace before a parser, and `token` handles any whitespace after a parser. Now we're ready to write our λ-calculus parser!

A Parser for the Untyped λ-Calculus

Where arithmetic was simple enough that we could actually evaluate it as we parsed it, λ-calculus is complex enough that we'll just have our parser do the tradition job of converting concrete syntax into abstract syntax. Recall the type of our λ-calculus AST:

```
datatype term = VAR of string
              | LAM of string * term
              | AP of term * term
```

First we'll define a function that gives us parsers for a symbol followed by any amount of whitespace, and a parser for any string we want to consider a variable.

```
val symbol = token o str
val variable = token (many1 alpha >>= return o String.implode)
```

Note that λ-calculus can be described using the chain concept that we discussed previously. An expression is essentially a chain of terms joined by an invisible application operator. So our expression parser is going to use `chainl1`:

```
fun expr i = chainl1 term (return AP) i

and term i = (var ++ paren ++ lam) i

and var i = (variable >>= return o VAR) i

and paren i = delim (symbol "(") expr (symbol ")") i

and lam i =
    (symbol "\\" >>
     variable    >>= (fn v =>
     symbol "."  >>
     expr        >>= (fn e =>
     return (LAM (v,e))
    ))) i
```

And there we are! Now we can parse any lambda term we like.