Bind

I'm just going to go ahead and import all of the things that we need and copy over some definitions. All of this is taken from Homework 8. A lot of this notebook will look incredibly similar to the examples in that homework. This will be looking at them in a little more detail with a few more examples.

In [1]:
```
import $file.hw8stdlib
import hw8stdlib._
```

Out[1]:
```
import $file.$

import hw8stdlib._
```

In [2]:
```
type Parser[S,D] = List[S] => List[(D, List[S])]

def char(c : Char) : Parser[Char,Char] =
(ss : List[Char]) => ss match {
    case Empty      => Empty
    case Cons(s,ss) => char_eq(s,c) match {
        case True  => singleton((s, ss))
        case False => Empty
    }
}

def success[S, D](x : D) : Parser[S, D] =
  (ss : List[S]) => singleton((x, ss))

def failure[S, D]() : Parser[S,D] = (ss : List[S]) => Empty

def choose[S, D](p : Parser[S, D], q : Parser[S, D]) : Parser[S, D] = (s
{
    val p_res = p(ss)
    val q_res = q(ss)
    append(p_res, q_res)
}

def runParser[D](p : Parser[Char, D], s : String) : Maybe[D] = p(string_
    case Empty                    => Nothing
    case Cons((x, Empty), Empty) => Just(x)
    case _                        => Nothing
}
```

Out[2]:
```
defined type Parser
defined function char
defined function success
defined function failure
defined function choose
defined function runParser
```

Bind - Sequencing our Parses

`bind` allows us to sequence parses. It can be a little confusing at first but with some practice it becomes very intuitive to use.

The type for bind is:

$$\textbf{bind} : \underbrace{\textbf{Parser}\ S\ D_1}_{p_1\text{ - First Parser}} \rightarrow \underbrace{(D_1 \rightarrow \textbf{Parser}\ S\ D_2)}_{p_2\text{ - Generator for Second Parser}} \rightarrow \underbrace{\textbf{Parser}\ S\ D_2}_{p_1p_2\text{ - Composite Parser}}$$

Let's break down each part of this signature:

- p_1 is the first parser we will use on an input. It will give us a parsed result of type D_1. We will then feed this result into the next part.
- p_2 is a generator for a parser which takes, as an argument something of type D_1, and returns a parser which gives results of type D_2. `bind` takes the result of running the first parser p_1 on some input and then feeds that result into this function, producing the output type for `bind` - $\textbf{Parser}\ S\ D_2$

Here is the implementation of `bind`.

In [3]:
```
def bind[S, D, E](p : Parser[S,D], q : (D => Parser[S,E]) ) : Parser[S,E
  (ss : List[S]) => {
      val join = (res : (D, List[S])) => res match {case (d, ss2) => q(d
      concatMap(join, p(ss))
  }
```

Out[3]:
```
defined function bind
```

"so thats cool and all but, like, what does it mean?"

First of all, don't start by trying to understand the code itself. Some of you might find it useful but I promise that learning what it is *doing* first will make it so much easier to look back at what the code actually did.

`bind` lets us string together multiple parsers. It does this by first taking in a valid parser. This could be the return value of something like `char('a')`. Let's go ahead and set up two of these and some strings to be used later.

In [4]:
```
val abc = string_to_list("abc")
val aab = string_to_list("aab")
val bcd = string_to_list("bcd")

val parseA = char('a')
val parseB = char('b')
```

Out[4]:
```
abc: List[Char] = Cons(a,Cons(b,Cons(c,Empty)))
aab: List[Char] = Cons(a,Cons(a,Cons(b,Empty)))
bcd: List[Char] = Cons(b,Cons(c,Cons(d,Empty)))
parseA: List[Char] => List[(Char, List[Char])] = <function1>
parseB: List[Char] => List[(Char, List[Char])] = <function1>
```

Awesome. Now we have two simple parsers that can parse out `a` and `b`. But what if we didn't

want to just parse one character? What if we wanted to parse `a` then `b` in a string? Well, we know that running them will either give us a list of valid parses if it works or an empty list if it doesn't.

In [5]:
```
parseA(abc)
parseB(abc)
```

Out[5]:
```
res4_0: List[(Char, List[Char])] = Cons((a,Cons(b,Cons(c,Empty))),Empty)
res4_1: List[(Char, List[Char])] = Empty
```

With that in mind, the first thing that `bind` does is take the input to parse then runs it through that first parser given to it. This means that if we have something like

```
bind(parseA, PARSER_GENERATOR)(abc)
```

the first thing that it will do is parse the variable `abc` using `parseA`. When the first parser executes, it returns something of type `(S,[D])`. In this case, we would get `('a', ['b', 'c'])` as output.

Now, I've clearly glossed over something really important. What is `PARSER_GENERATOR`? We say that a parser generator is a function that returns a parser. You've actually already encountered one of these; the function `char` is a parser generator. How do we know this? Well, in our `bind` definition, we stated that its second argument should be a function that takes in a single argument and returns a parser. `char` takes a single argument, the character to parse, then returns a parser.

The next thing we need to figure out is why we need a parser generator instead of just a parser itself. This is where the magic happens. After the first parser is applied to the input, its output is passed to the parser generator. Let's break that down. If we had something like

```
bind(parseA, PARSER_GENERATOR)(aab)   // note the new input
```

the first thing that happens is the parser is applied to the input. This gives us the output `('a', ['a', 'b'])`. For right now, let's just rename those two tuple fields so we have `(SUCCESSFUl_PARSE, TO_BE_PARSED)`. `bind` then takes those values and does this with them: `PARSER_GENERATOR(SUCCESSFUL_PARSE)(TO_BE_PARSED)`. Therefore if we had

```
bind(parseA, char)(aab)
```

it first would apply `parseA` to the input giving us `('a', ['a', 'b'])`. It would then do `char('a')(['a', 'b'])` as `a` is the successful parse and the list is what still needs to be parsed.

Sweet! Let's see it in action!

In [6]:
```
bind(parseA, char)(aab)
```

Out[6]:
```
res5: List[(Char, List[Char])] = Cons((a,Cons(b,Empty)),Empty)
```

Hold up, something seems off. We threw both of those into `bind`, parsed `aab`, then got back a valid parse on only one letter `a`. That should make some sense. At the end of the day, all that

we actually returned was the result of `char('a')(['a', 'b'])`. We lost the original parse results entirely. This also should seem odd to you because, with our current parser generators, we can only parse the same character twice in a row.

Now it's time to write a new parser generator!

In [7]:
```
def parseLetterThenB(x : Char) : Parser[Char, List[Char]] = bind(parseB,
                                                                 (letterb : (
                                                                     success
```

Out[7]:
```
defined function parseLetterThenB
```

`parseLetterThenB` is a parser generator that takes in the successful parse of the last parser and returns another parser (it is ok if "parser" no longer looks like a real word). Specifically, we have another bind in there that lets us string together even more parsers!

This may seem needlessly complicated but it's how we'll capture the output of the original parser. It first takes in the result of the previous parse to hold onto for later. It then parses the new string input with `parseB` and returns a concatenation of those parse results.

What does this look like in practice? Well, if we just pass in a random character to `parseLetterThenB` and apply it to a string, we can look at its output. (I am using the letter `t` just so we can clearly see where it ends up)

In [8]:
```
parseLetterThenB('t')(bcd)
```

Out[8]:
```
res7: List[(List[Char], List[Char])] = Cons((Cons(t,Cons(b,Empty)),Cons
(c,Cons(d,Empty))),Empty)
```

See how that `t` is at the front of the parse? That's because it was the argument to the generator. Now ignore that `t` for a moment and look at the rest of the list. It looks the same as if we just ran

In [9]:
```
parseB(bcd)
```

Out[9]:
```
res8: List[(Char, List[Char])] = Cons((b,Cons(c,Cons(d,Empty))),Empty)
```

The only difference is that the `t` was prepended to the list. This should also make sense as the first step of a parser generated by `parseLetterThenB` is to apply `parseB` to the input. Aha! We can finally recover the results of the previous parse. `parseLetterThenB` lets us complete a parse, perform a `parseB` on the remainder of the input, then return all of the successful parses that happened and everything that still needs to be parsed. We can now finally parse an `a` then a `b` out of a string.

In [10]:
```
bind(parseA, parseLetterThenB)(abc)
```

Out[10]:
```
res9: List[(List[Char], List[Char])] = Cons((Cons(a,Cons(b,Empty)),Cons
(c,Empty)),Empty)
```

In this example `parseA` is applied to `abc`. Because this is successful, this gives us the initial

result of `('a', ['b', 'c'])`. This is then applied to the second argument as `parseLetterThenB('a')(['b', 'c'])`. We know that this will actually parse the input, then prepend `a` to its parse results. This is how we get our final result.

Looking Back at the Code

This is an optional section for anyone curious. It's just going to show the code you already said but with some comments showing what's happening. If you get the parts above but not this, you're fine.

In [11]:
```
def bind[S, D, E](p : Parser[S,D], q : (D => Parser[S,E]) ) : Parser[S,E
  (ss : List[S]) => {
      val join = (res : (D, List[S])) => res match { // Apply the first 
          case (d, ss2) => q(d)(ss2) // If the parse was successful, use
                                     // then pass what it still needs to
      }
      concatMap(join, p(ss)) // Combine the output of the two parsers
  }
```

Out[11]:
```
defined function bind
```

In [ ]:
```

```