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
}

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))
  }

def stringL(ss : List[Char]) : Parser[Char, List[Char]] = ss match {
    case Empty      => success(Empty)
    case Cons(s,ss) =>        bind(char(s),
         (c : Char)        => bind(stringL(ss),
         (cs : List[Char]) => success(Cons(c,cs)) ))
}

def string(str : String) : Parser[Char, List[Char]] = stringL(string_to_
```
Out[2]:
```
defined type Parser
defined function char
defined function success
defined function failure
defined function choose
defined function runParser
defined function bind
defined function stringL
defined function string
```

Recitation 9

In this recitation we will develop the last few tools we will need to write our Lexer and Parser.

Adendum: You're teacher made an oopsie

Dear Class,

I hope this finds you well. I'm writing you from a table at Panera Bagels amidst a small rain storm. I have just made the observation that I have made a grave error when teaching you this very capturing and awe-inspiring material. There are in fact two different combinators that capture the notion of "choice". The first is the `choose` combinator that we have been using. This is actually a slightly less than ideal option. We often would actually prefer the `option` function defined here:

In [3]:
```
def option[S, D](p : Parser[S, D], q : Parser[S, D]) : Parser[S, D] = (s
  p(ss) match {
      case Empty => q(ss)
      case res   => res
  }
```

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

As you will quickly notice, this parser combinator will parse using the first parser `p` and if `p` is unsuccesful it will attempt parsing with `q`. This is the idea of *exclusive or* that we are familiar with from logic. The `choose` combinator will explore all possible paths. We will often prefer `option` because of its treatment of precedence with our parsers. Please take a second to note the differences between the two and understand when one might want to use one over the other.

I hope you enjoy today's recitation as much as I enjoyed writing it. Behold, the joy of functional programming!

Sincerely,

John Martin, Instructor of the Dark Arts(of Programming)

Parsing Whitespace

Parsing whitespace will be very important for building out any lexer or parser. We will treat whitespace as the delimiter between different tokens in our lexical grammar, and so it is very important we have a working parser for whitespace characters. We can write a basic parser for the different whitespace characters like so:

In [4]:
```
def whitespaceChar : Parser[Char,Char] = choose(char(' '),
                                         choose(char('\n'),
                                         choose(char('\r'),
                                                char('\t'))))
```

Out[4]:
```
defined function whitespaceChar
```

The only problem with this parser is that it doesn't have any way of handling multiple whitespace characters. In general, we don't have a way of applying a parser a multitude of times. We will need to develop this combinator, it is called `many` .

`many`

`many` will be a combinator that takes a parser and uses that parser on the input string as many times as possible.

In [5]:
```
def many[S,D](p : Parser[S,D]) : Parser[S, List[D]] =
option(
    bind(p, (res : D) => bind(many(p), (many_res : List[D]) => success(C
```

Out[5]:
```
defined function many
```

We can use many to create a parser that parses all of the whitespace characters in a string.

In [6]:
```
def whitespace : Parser[Char, List[Char]] = many(whitespaceChar)
```

Out[6]:
```
defined function whitespace
```

And we can see with the examples below that it works as we would expect:

In [7]:
```
runParser(whitespace, " ")
runParser(whitespace, " ")
runParser(whitespace, "\n")
runParser(whitespace, "        ")
runParser(whitespace, "    \n \r   ")
```

Out[7]:
```
res6_0: Maybe[List[Char]] = Just(Cons( ,Empty))
res6_1: Maybe[List[Char]] = Just(Cons( ,Empty))
res6_2: Maybe[List[Char]] = Just(Cons(
,Empty))
res6_3: Maybe[List[Char]] = Just(Cons( ,Cons( ,Cons( ,Cons( ,Cons( ,Con
s( ,Cons( ,Empty))))))))
res6_4: Maybe[List[Char]] = Just(Cons( ,Cons( ,Cons( ,Cons( ,Cons(
,Cons( ,Cons( ,Cons( ,Empty)))))))))
```

Note that in the examples above the whitespace characters did not need to all be the same. They just had to be successfully parsed by the `whitespaceChar` parser.

Parsing Numbers

One of the trickier parses we will need to perform is on numbers. We will need to *look-ahead* a bit to see if the number has ended or not, and if it has we will need some way of converting it into an Integer or Natural Number.

Let's begin by defining a parser for the digits:

In [8]:
```
def digit : Parser[Char, Char] =
    option(char('0'),
    option(char('1'),
    option(char('2'),
    option(char('3'),
    option(char('4'),
    option(char('5'),
    option(char('6'),
    option(char('7'),
    option(char('8'),
    (char('9'))))))))))
```

Out[8]:
```
defined function digit
```

Now that we have this we need a way of parsing many digits. It turns out that we have just that, the `many` parser will perform this perfectly.

In [9]:
```
def numString : Parser[Char, List[Char]] = many(digit)
```

Out[9]:
```
defined function numString
```

Now the last step will be to convert the string of digits into a natural number. Write the function, `stringToNat` below. It may be helpful to write a function that converts digits to nats first.

In [10]:
```
def digitToNat(c : Char) : Nat = c match{
    case '0' => Zero
    case '1' => one
    case '2' => two
    case '3' => three
    case '4' => four
    case '5' => five
    case '6' => six
    case '7' => seven
    case '8' => eight
    case '9' => nine
}

def stringToNat(xs : List[Char]) : Nat =
    fold((x : Nat, acc : Nat) => nat_plus(x, nat_mult(acc, ten)), Zero, 
```

Out[10]:
```
defined function digitToNat
defined function stringToNat
```

Now use `bind` to apply `stringToNat` to the result of a parsed number string. This will be our final parser called `number`.

In [11]:
```
def number : Parser[Char, Nat] = bind(numString, (ss: List[Char]) => suc
```

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

If this was done properly, the next line should give successful parses:

In [12]:
```
runParser(number, "5")
runParser(number, "12")
runParser(number, "100")
```

Out[12]:
```
res11_0: Maybe[Nat] = Just(Succ(Succ(Succ(Succ(Succ(Zero))))))
res11_1: Maybe[Nat] = Just(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Zero)))))))))))))
res11_2: Maybe[Nat] = Just(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ
(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Zer
o)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

In [ ]: