In [24]:
```
import $file.hw8stdlib
import hw8stdlib._
```

```
Compiling hw8stdlib.sc
```

Out[24]:
```
import $file.$

import hw8stdlib._
```

In [25]:
```
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] =
(ss : List[S]) =>
{
    val p_res = p(ss)
    val q_res = q(ss)
    append(p_res, q_res)
}
```

Out[25]:
```
defined type Parser
defined function char
defined function success
defined function failure
defined function choose
```

Homework 8

In this week's homework we will build up more interesting parsers by creating some new combinators using `bind`

RunParser

`runParser` is a useful function for running a parser on an input. You will see it used in some of the examples below.

In [26]:
```
def runParser[D](p : Parser[Char, D], s : String) : Maybe[D] = p(stri
ng_to_list(s)) match {
    case Empty                    => None
    case Cons((x, Empty), Empty) => Just(x)
    case _                        => None
}
```

Out[26]:
```
defined function runParser
```

Problem 1 - `satisfies` (10 points)

For the following type:

$$\text{satisfies} : \underbrace{(S \to \mathbb{B})}_{\text{predicate}} \to \textbf{Parser}\ S\ \textbf{Char}$$

Implement `satisfies` such that it successfully parses a symbol that the predicate is true for and fails if the predicate is false.

hint The implementation of `char` above may be helpful as a starting point.

In [27]:
```
def satisfies[S](p : (S => Bool)) : Parser[S, S] = (xs : List[S]) =>
xs match{
    case Empty => Empty
    case Cons(s, ss) => p(s) match{
        case True => singleton((s, ss))
        case False => Empty
    }
}
```

Out[27]:
```
defined function satisfies
```

In [28]:
```
// Tests empty input
def pt(x: Char): Bool = True
def pf(x: Char): Bool = False
assert(runParser(satisfies(pt), "") == None)
assert(runParser(satisfies(pf), "") == None)
passed(5)
```

```
*** Tests Passed (5 points) ***
```

Out[28]:
```
defined function pt
defined function pf
```

In [29]:
```
// Tests all true or false
assert(satisfies(pt)(string_to_list("a")) == Cons(('a',Empty),Empty))
assert(satisfies(pt)(string_to_list("abc")) == Cons(('a',Cons('b',Con
s('c',Empty))),Empty))
assert(satisfies(pf)(string_to_list("abc")) == Empty)
passed(3)
```

```
*** Tests Passed (3 points) ***
```

In [30]:
```
// Tests specific true or false
assert(satisfies[Char](char_eq(_, '0'))(string_to_list("0bc")) == Cons(('0',Cons('b',Cons('c',Empty))),Empty))
assert(satisfies[Char](char_eq(_, '0'))(string_to_list("abc")) == Empty)
passed(4)
```

```
*** Tests Passed (4 points) ***
```

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:

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

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`. Don't worry too much about how it works. We will get an intuition for it through some examples.

In [31]:
```
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)(ss2)}
      concatMap(join, p(ss))
  }
```

```
Out[31]: defined function bind
```

Example 1 - a then b

In this example we will write a parser which recognizes the sequence of characters `"ab"`. We can begin by defining the parser for the first letter, `'a'`. For this we just need to use `char`:

In [32]:
```
def parseLetterA : Parser[Char,Char] = char('a')
```

```
Out[32]: defined function parseLetterA
```

`parseLetterA` will be the first argument to `bind`. The next thing we need is a function of type:
$D_1 \rightarrow \textbf{Parser}\ S\ D_2$

We will start by defining the parser for just the letter B:

```
In [33]: def parseLetterB : Parser[Char, Char] = char('b')
Out[33]: defined function parseLetterB
```

We would like our parse result to be a list of characters that contains "ab"(which makes $D_2 = \textbf{List Char}$). We also know that the result of the first parser is a character(This means $D_1 = \textbf{Char}$) This specializes this type to: $\textbf{Char} \rightarrow \textbf{Parser}\ S\ (\textbf{List Char})$.

We will say that this is a parser that takes a letter and then parses a `'b'`. After doing this it combines them into a list of characters.

```
In [34]: def parseLetterThenB(x : Char) : Parser[Char, List[Char]] =
             bind(
                 parseLetterB,
                 (letterb : Char) => success(Cons(x, Cons(letterb, Empty)))
             )
Out[34]: defined function parseLetterThenB
```

Above we provided bind a parser for the letter `'b'` and a lambda-function which takes the result of that parse and combines it with the letter `x` that was passed in as an argument.

Now that we have this we can combine it all into our final parser for `"ab"`:

```
In [35]: def parseAThenB : Parser[Char, List[Char]] =
             bind(parseLetterA, parseLetterThenB)
Out[35]: defined function parseAThenB
```

If we test this we will notice it behaves as we would expect:

```
In [36]: parseAThenB(string_to_list("ab"))
Out[36]: res35: List[(List[Char], List[Char])] = Cons((Cons(a,Cons(b,Empty)),E
         mpty),Empty)
```

Example 2 - *abc*

Writing parsers as a bunch of small functions like this often becomes cumbersome so we prefer to write them as a single definition using several calls to `bind` at once. Here is an example of this for the string `"abc"`:

In [37]:
```
def parseABC : Parser[Char, List[Char]]  = bind(char('a'),
                            (p_a : Char) => bind(char('b'),
                            (p_b : Char) => bind(char('c'),
                            (p_c : Char) => success(Cons(p_a, Cons(p_b
, Cons(p_c, Empty)))))))
```

Out[37]:
```
defined function parseABC
```

Here is a quick proof that it works:

In [38]:
```
runParser(parseABC, "abc")
runParser(parseABC, "xyz")
```

Out[38]:
```
res37_0: Maybe[List[Char]] = Just(Cons(a,Cons(b,Cons(c,Empty))))
res37_1: Maybe[List[Char]] = None
```

To read the function above just take each call to `bind` to mean that we are calling a parser and using its result. So on the second line we have p_a which is the character that resulted from parsing $\mathrm{char}('a')$. In the next bind we have access to both p_a and p_b. In the third we have all three results and can combine them into the list we wanted. We need to wrap this list up in `success` so that it has the necessary type($\mathbf{Parser\ Char\ (List\ Char)}$)

Note on Layout We strongly recommend laying out calls to bind on multiple lines for readability. It need not be in the same manner as we have done here though. For instance the lambda terms could have been on the same line as their bind call. Any kind of spacing that helps read the code is a good idea here.

`string` - A New Combinator

The two examples above are examples of a more general class of parsers where we wish to parse some specific combination of characters, or a string. We can write a special parser generator which takes a string and creates a parser for that string:

In [39]:
```
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_list(str))
```

Out[39]:
```
defined function stringL
defined function string
```

In [40]:
```
runParser(string("abc"), "abc")
```

Out[40]:
```
res39: Maybe[List[Char]] = Just(Cons(a,Cons(b,Cons(c,Empty))))
```

Problem 2 - Lettuce Keywords (10 points)

Write a parser, using `string` and `choice` which accepts the strings `"let"`, `"in"`, `"function"`, and `"rec"`

In [41]:
```
def parseKeywords : Parser[Char, List[Char]] = choose(string("let"),
                                                      choose(string("in"),
                                                      choose(string("functi
on"),
                                                             string("rec"
))))
```

Out[41]:
```
defined function parseKeywords
```

If your solution is correct the following examples should be parsed successfully:

In [42]:
```
assert(runParser(parseKeywords, "let") == Just(string_to_list("let"
)))
assert(runParser(parseKeywords, "rec") == Just(string_to_list("rec"
)))
assert(runParser(parseKeywords, "function") == Just(string_to_list("f
unction")))
assert(runParser(parseKeywords, "in") == Just(string_to_list("in")))
passed(5)
```

```
*** Tests Passed (5 points) ***
```

and the following examples should fail:

In [43]:
```
assert(runParser(parseKeywords, "x") == None)
assert(runParser(parseKeywords, "or") == None)
assert(runParser(parseKeywords, "functio") == None)
assert(runParser(parseKeywords, "ni") == None)
passed(5)
```

```
*** Tests Passed (5 points) ***
```

Problem 3 - `mapParser` (10 points)

One thing we may want to do to a parser is apply a function to its results. A function that does this could have the type:

$$\text{mapParser} : (A \to B) \to \textbf{Parser}\ S\ A\ \to \textbf{Parser}\ S\ B$$

We do something very much like this by applying cons to the result of our parsers above. Using `bind` write the `mapParser` function below:

In [43]:
```
def mapParser[S,A,B](f : (A => B), p : Parser[S,A]) : Parser[S,B] = b
ind(p,
```

```
<console>:1: not a legal formal parameter.
Note: Tuples cannot be directly destructured in method or function pa
rameters.
      Either create a single parameter accepting the Tuple1,
      or consider a pattern matching anonymous function: `{ case (par
am1, param1) => ... }
def mapParser[S,A,B](f : (A => B), p : Parser[S,A]) : Parser[S,B] = b
ind(p, (q: A)) => success(f(q))

^
```

In [21]:
```
// Tests Empty input
assert(runParser(mapParser(list_to_string, string("")), "") == Just(
""))
passed(3)
```

```
scala.NotImplementedError: an implementation is missing
  scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  $sess.cmd19Wrapper$Helper.mapParser(cmd19.sc:3)
  $sess.cmd20Wrapper$Helper.<init>(cmd20.sc:1)
  $sess.cmd20Wrapper.<init>(cmd20.sc:307)
  $sess.cmd20$.<init>(cmd20.sc:223)
  $sess.cmd20$.<clinit>(cmd20.sc:-1)
```

In [22]:
```
// Tests non empty success
assert(runParser(mapParser(list_to_string, string("abc")), "abc") ==
Just("abc"))
assert(runParser(mapParser(reverse[Char], string("abcd")), "abcd") ==
Just(string_to_list("dcba")))
passed(4)
```

```
scala.NotImplementedError: an implementation is missing
  scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  $sess.cmd19Wrapper$Helper.mapParser(cmd19.sc:3)
  $sess.cmd21Wrapper$Helper.<init>(cmd21.sc:1)
  $sess.cmd21Wrapper.<init>(cmd21.sc:311)
  $sess.cmd21$.<init>(cmd21.sc:223)
  $sess.cmd21$.<clinit>(cmd21.sc:-1)
```

In [23]:
```
// Tests failure
assert(runParser(mapParser(list_to_string, string("abc")), "xyz") ==
None)
passed(3)
```

```
scala.NotImplementedError: an implementation is missing
  scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  $sess.cmd19Wrapper$Helper.mapParser(cmd19.sc:3)
  $sess.cmd22Wrapper$Helper.<init>(cmd22.sc:1)
  $sess.cmd22Wrapper.<init>(cmd22.sc:307)
  $sess.cmd22$.<init>(cmd22.sc:223)
  $sess.cmd22$.<clinit>(cmd22.sc:-1)
```