In [1]:
```
import $file.rec8stdlib
import rec8stdlib._
```
Out[1]:
```
import $file.$

import rec8stdlib._
```

Recitation N

In this recitation we will get some practice with the basic parser combinators.

The Parser Type

We define parsers below with the following Grammar:

$$\text{type } \mathbf{Parser}\ s\ d = (\ (\mathbf{List}\ s) \rightarrow\ \ \mathbf{List}\ (d, (\mathbf{List}\ s))\ )$$

We could also write the $\mathbf{List}$ type as square brackets. So we would write $\mathbf{List}\ A$ as $[A]$ which may be easier to read:

$$\text{type } \mathbf{Parser}\ s\ d = (\ [s] \rightarrow\ \ [(d, [s])]\ )$$

As a refresher, let's break down each part of this type.

$$\text{type } \mathbf{Parser}\ s\ d = (\ [s] \rightarrow\ \ [(d, [s])]\ )$$

- s - The type of the symbols the parser is reading in. Usually this will be characters but could also be any other data type.
- d - The type of the data we are returning from the parse. This is the structure we are trying to build up. Later this will be Lettuce Expressions.
- $[s] \rightarrow\ \ [(d, [s])]$ - The parsing function. This is any function that takes a list of symbols(such as a string) and returns a list of success parses. A successful parse is any tuple of the parsed structure d and the rest of the list that still needs to be parsed

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

The Primitives

We will define three primitive parsers that we will use to build up all of the other parsers we will need. These are:

- `char` Takes a character as an argument and parses that character
- `success` Takes any element of type D and returns a parser of that type
- `failure` Unsuccesful parses

Char

The `char` primitive is a parser for the provided character `c`. This can be any character that is included in Scala's definition. We will use this as the primary building block for all of our parsers going forward.

In [3]:
```
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
    }
}
```

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

Success

This is a fairly simples primitive that acts as a pass-through. It will just wrap up its argument into a successful parse. This will be useful when returning results inside of the `bind` combinator.

In [4]:
```
def success[S, D](x : D) : Parser[S, D] =
  (ss : List[S]) => singleton((x, ss))
```

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

Failure

This is the dual of `success` and is used any time we have an unsuccesful parse

In [5]:
```
def failure[S, D]() : Parser[S,D] = (ss : List[S]) => Empty
```

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

The Combinators

We will use these primitives with *combinators*(A name that only a mathmetician could make up) to create our parsers. We will build up most of our parsers with two main combinators. We will explore the second next week, for now let's take a look at:

Choice

`Choice` represents a case where you have two parsers and want to combine them in an either/or way. If you have a parser that recognizes numbers and another that recognizes words you could combine them to recognize both numbers and words.

In [6]:
```
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)
}
```

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

Examples

Note: To convert a string to a list of characters call the `string_to_list` function

1

Write a parser that accepts either a string beginning with `'a'` or `'z'`

In [7]:
```
val p1 = choose(char('a'), char('z'))

val ex1 = string_to_list("abc")
val ex2 = string_to_list("zyx")
val ex_bad = string_to_list("dog")
```

Out[7]:
```
p1: List[Char] => List[(Char, List[Char])] = <function1>
ex1: List[Char] = Cons(a,Cons(b,Cons(c,Empty)))
ex2: List[Char] = Cons(z,Cons(y,Cons(x,Empty)))
ex_bad: List[Char] = Cons(d,Cons(o,Cons(g,Empty)))
```

In [8]:
```
p1(ex1)
p1(ex2)
p1(ex_bad)
```

Out[8]:
```
res7_0: List[(Char, List[Char])] = Cons((a,Cons(b,Cons(c,Empty))),Empt
y)
res7_1: List[(Char, List[Char])] = Cons((z,Cons(y,Cons(x,Empty))),Empt
y)
res7_2: List[(Char, List[Char])] = Empty
```

2

Write a parser that accepts any digit (0-9)

In [9]:
```
val p01 = choose(char('0'), char('1'))
val p23 = choose(char('2'), char('3'))
val p45 = choose(char('4'), char('5'))
val p67 = choose(char('6'), char('7'))
val p89 = choose(char('8'), char('9'))

val p03 = choose(p01, p23)
val p05 = choose(p03, p45)
val p07 = choose(p05, p67)

val pdigits = choose(p07, p89)

// OR a little prettier
val digits = string_to_list("0123456789")
val digit_parsers = map(char, digits)
val pdigits_alt = fold( choose[Char,Char], failure[Char,Char], digit_par
```

Out[9]:
```
p01: List[Char] => List[(Char, List[Char])] = <function1>
p23: List[Char] => List[(Char, List[Char])] = <function1>
p45: List[Char] => List[(Char, List[Char])] = <function1>
p67: List[Char] => List[(Char, List[Char])] = <function1>
p89: List[Char] => List[(Char, List[Char])] = <function1>
p03: List[Char] => List[(Char, List[Char])] = <function1>
p05: List[Char] => List[(Char, List[Char])] = <function1>
p07: List[Char] => List[(Char, List[Char])] = <function1>
pdigits: List[Char] => List[(Char, List[Char])] = <function1>
digits: List[Char] = Cons(0,Cons(1,Cons(2,Cons(3,Cons(4,Cons(5,Cons(6,C
ons(7,Cons(8,Cons(9,Empty))))))))))
digit_parsers: List[List[Char] => List[(Char, List[Char])]] = Cons(<fun
ction1>,Cons(<function1>,Cons(<function1>,Cons(<function1>,Cons(<functi
on1>,Cons(<function1>,Cons(<function1>,Cons(<function1>,Cons(<function1
>,Cons(<function1>,Empty))))))))))
pdigits_alt: List[Char] => List[(Char, List[Char])] = <function1>
```

In [10]:
```
assert(pdigits(string_to_list("4")) == Cons(('4', Empty),Empty))
assert(pdigits(string_to_list("9sd")) == Cons(('9', Cons('s', Cons('d',
assert(pdigits(string_to_list("214")) == Cons(('2', Cons('1', Cons('4',
assert(pdigits(string_to_list("d")) == Empty)
assert(pdigits(string_to_list("d3443")) == Empty)
```

3

Write 4 parsers which do the following when given the string `"abcd"`

1. One that fails to parse
2. One that produces a single successful parse
3. One that produces 3 successful parses (it's ok if they're the same as long as there are 3 results in the list)
4. One that produces 32 results (think about how to do this efficiently)

In [11]:
```scala
val x = string_to_list("abcd")

// Have to specify types for failure cause Scala defaults it to `Nothing
val parser1 = failure[Char, Nothing]() // Could also do char(/* anything
val parser2 = char('a') // Could use success
val parser3 = choose(choose(char('a'), char('a')), char('a'))

val p2 = choose(char('a'), char('a'))
val p4 = choose(p2, p2)
val p8 = choose(p4, p4)
val p16 = choose(p8, p8)
val parser4 = choose(p16, p16)
```

Out[11]:
```
x: List[Char] = Cons(a,Cons(b,Cons(c,Cons(d,Empty))))
parser1: List[Char] => List[Tuple2[Nothing, List[Char]]] = <function1>
parser2: List[Char] => List[(Char, List[Char])] = <function1>
parser3: List[Char] => List[(Char, List[Char])] = <function1>
p2: List[Char] => List[(Char, List[Char])] = <function1>
p4: List[Char] => List[(Char, List[Char])] = <function1>
p8: List[Char] => List[(Char, List[Char])] = <function1>
p16: List[Char] => List[(Char, List[Char])] = <function1>
parser4: List[Char] => List[(Char, List[Char])] = <function1>
```

In [12]:
```scala
assert(length(parser1(x)) == Zero)
assert(length(parser2(x)) == one)
assert(length(parser3(x)) == three)
assert(length(parser4(x)) == nat_pow(five, two))
```