Lec 09 - Monadic Parsing

CS 1XA3

March 13^{th}, 2018

What is a Parser?!

- The act of parsing refers to the processing of a String (by some syntactic analysis) into a desired data structure
- One might write a parser for a single digit like so

```
parseDigit :: String -> Maybe Int
parseDigit (d:_) = case d of
      '0' -> Just 0
      '1' -> Just 1
       ...
       _ -> Nothing
parseDigit "" = Nothing
```

- Note: because we may fail to parse, we use a Maybe to wrap our result

Parsing Multiple Characters

If we wanted to continue parsing digits, it would be useful to return the rest of the unparsed String. So a better type for the previous example would be

```
parseDigit :: String -> Maybe (Int,String)
parseDigit (c:cs) = let
    char2Digit c = case c of
                     '0' -> Just 0
                     ...
                     _   -> Nothing
  in fmap (\x -> (x,cs)) (char2Digit c)
parseDigit [] = Nothing
```

Parsinge Multiple Characters

- Using the previous definition, we can take advantage of the Maybe Monad

```
parse2Digits :: String -> Maybe (Int,Int,String)
parse2Digits ss = parseDigit ss
 >>= (\(d1,ss') ->
   fmap (\(d2,ss'') -> (d1,d2,ss''))
                       (parseDigit ss') )
```

- Alternatively written using the do syntax

```
parse2Digits ss = do (d1,ss') <- parseDigit ss
                     (d2,ss'') <- parseDigit ss'
                     return (d1,d2,ss'')
```

Creating A Parser Type

- In a more general setting, we wouldn't want to restrict parsing to an Int but to any data type thats most appropriate, i.e

```
parse :: String -> Maybe (a,String)
```

- This doesn't work too well in a function defintion, however we can wrap this as a datatype of Parsers like so

```
data Parser a = Parser (String -> Maybe (a,String))

parse :: Parser a -> String
parse (Parse p) ss = p ss
```

To Be Continued ..

Now that we have a Parser datatype we can make it a

- Functor
- Applicative
- Alternative
- Monad