Lab 10 - Monadic Parsing Cont.

CS 1XA3

March 19^{th}, 2018

Recall: Creating A Parser Type

- In the previous lecture, we defined a parsing function that returned a Maybe result, containing the parsed value and remaining string

```
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))
```

- Executing a Parser would involve unwrapping the function from the type and applying it to a String, we can automate this process with the following auxilary function

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

Primitive Parser Combinators

We'll build more complicated Parsers out of smaller simple parsers (parser combinators)

- An empty, zero parser that doesn't actually attempt to parse anything (we'll see how useful this is later)

```haskell
zero :: Parser a
zero = Parser (\_ -> Nothing)
```

- A parser item for parsing any single Char

```haskell
item :: Parser Char
item = let
   getItem ss = case ss of
         (c:cs) -> Just (c,cs)
         []     -> Nothing
  in Parser getItem
```

Revisiting Parsing Digits

- A function for parsing a single digit into an Int can be defined for this type as follows

```
parseDigit :: Parser Int
parseDigit = let
     char2Digit c  = case c of
                        '0' -> Just 0
                        ...
                        '9' -> Just 9
                        _   -> Nothing
  in Parser (\ss -> case ss of
     (c:ss') -> fmap (\x -> (x,ss')) (char2Digit c)
     []      -> Nothing )
```

- This function definition is tedious and messy, perhaps Functors,Applicatives, and Monads can help clean things up

Functor and Applicative Parser

- When defining a Functor instance, we need to wrap and unwrap inside of a new function

```
instance Functor Parser where
  fmap f p = Parser (\ss -> case parse p ss of
                Just (a,ss') -> Just (f a,ss')
                Nothing      -> Nothing )
```

- The Applicative instance is even messier, but still what you would expect

```
instance Applicative Parser where
  pure p    = Parser (\ss -> Just (p,ss))
  fp <*> p = Parser (\ss -> case parse fp ss of
                Just (f,ss1) -> case parse p ss1 of
                      Just (x,ss2) -> Just (f x,ss2)
                      Nothing      -> Nothing
                Nothing -> Nothing )
```

Parsing Using Monads!

- Functors and Applicatives are nice, but Monads will give us the real power we're looking for

```
instance Monad Parser where
  p >>= f = Parser (\ss -> case parse p ss of
                 Just (a,ss') -> parse (f a) ss'
                 Nothing      -> Nothing )
```

- There's another class that proves quite useful for parsing called Alternative, located in Control.Applicative

```
instance Alternative Parser where
  empty = zero
  p <|> q = Parser (\ss -> case parse p ss of
              Nothing -> parse q ss
              res     -> res)
```

Primitive Parser Combinators Cont.

Now that we have a Monad instance, we can continue defining primitive combinators with ease

- Parse an item under a condition

```
itemIf :: (Char -> Bool) -> Parser Char
itemIf cond = do { c <- item;
                   if cond c
                     then return c
                     else zero }
```

- A Parser for a single specified Char

```
char :: Char -> Parser Char
char c = itemIf (==c)
```

Case Study: Parsing two Characters

- Consider the following function

```
parseIF :: Parse String
parseIF = do { char 'i';
               char 'f';
               return "if" }
```

- Note: the suffix (leftover String) is passed along automatically
- Note++: if at any point Nothing is returned, the whole function returns Nothing

Revisiting: Parsing a Digit

- Now that we have the power of the Monad, we can redefine how to parse a digit

```
digit :: Parser Int
digit = do { c <- item;
             case c of
               '0' -> return 0
               ...
               '9' -> return 9
               _   -> zero }
```

- Much cleaner!

Primitive Parser Combinators Cont.

- Apply a parser over and over again until nothing is returned

```
manyP :: (Monad f, Alternative f) => f a -> f [a]
manyP p = manyP1 p <|> return []

manyP1 :: (Monad f, Alternative f) => f a -> f [a]
manyP1 p = do { a <- p;
                as <- manyP p;
                return (a:as) }
```

- Note: both functions return a List, manyP may return an empty list but manyP1 returns at least on thing or Nothing

Challenge: Write some more useful Combinators

Try defining the following combinators yourself

- A string parser, similar to the char parser, but parsers a whole String

  ```
  string :: String -> Parser String
  ```

- A spaces parser that parses zero or more occurances of spaces

  ```
  spaces :: Parser String
  ```

- A token parser that works just like string, but allows for spaces padding the String being parsed

  ```
  token :: String -> Parser String
  ```

Parsing Expressions With Infix Operators

- Consider we want to parse an expression with infix operators, like

  `"1+2+3+4+5"` or `"1+2*3"` or `"1"`

- We need a parser combinator like this one

```
chainl1 :: Parser a -> Parser (a -> a -> a) ->Parser a
p `chainl1` op = do { a <- p; rest a }
  where
    rest a = (do f <- op
                 b <- p
                 rest (f a b))
             <|> return a
```

Parsing Boolean Expressions

- Consider the following data type for Boolean Expressions

```
data BoolExpr = And BoolExpr BoolExpr
              | Or BoolExpr BoolExpr
              | Boolean Bool
  deriving Show
```

- We wish to be able to parse an expression in a String like

```
"0|1 & 0|0"
```

- into it's corresponding data type, i.e

```
And (Or (Boolean False) (Boolean True))
    Or (Boolean False) (Boolean False))
```

Parsing Boolean Expressions

- First, define a function for parsing boolean values

```
boolean :: Parser BoolExpr
boolean = let
  true = token "0" >> return (Boolean False)
  false = token "1" >> return (Boolean True)
  in true <|> false
```

- Next, for our two infix operators And and Or

```
andOp = token "&" >> return And
orOp = token "|" >> return Or
```

- Finally, we put everything together with chainl1

```
expr = term `chainl1` andOp
term = boolean `chainl1` orOp
```

Challenge

- Create a Parser that parses a string of digits into an Int

  ```
  parseInt :: Parser Int
  ```

  Hint: parse each digit into a list with manyP1, then multiply and sum the String by 10^i

- Create a parser for the simple Int expression data type

  ```
  data IntExpr = Add IntExpr IntExpr
               | Mult IntExpr IntExpr
               | Constant Int

  expr :: Parser IntExpr
  ```