Lec 10 - The Parsec Library

CS 1XA3

March 14^{th}, 2018

Recap: Monadic Parsing!

- The last few lectures, we designed a Monadic Parser and used it to parse simple boolean expressions
- The general idea behind all Monadic Parsers was taken from this popular paper, I encourage you to review it `http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf`
- Although building your own parser library is a good exercise, you should use a more robust library for real work. Parsec is the go too Haskell parsing library `https://hackage.haskell.org/package/parsec`

Using Parsec: Including Parsec in your Code

- First, make sure your packages are up to date and install the Parsec package with cabal

```
cabal update
cabal install parsec
```

- To include all the basic Parsec functions in your code, add the following import

```
import Text.Parsec
import Text.Parsec.String
```

- Almost all the basic parsing functions we defined for our own parser are included in Text.Parsec, see https://hackage.haskell.org/package/parsec-3.1.13.0/docs/Text-Parsec.html

Parsec Vs Our Parser

- Our Parser from last lecture

```
data Parser a = Parser { unParser ::
                         (String -> Maybe (a,String)) }
```

- Parsec: definition is a little bit more complicated

```
data ParsecT s u m a
    = ParsecT {unParser :: forall b . State s u
               -> (a -> State s u -> ParseError -> m b)
               -> (ParseError -> m b)
               -> (a -> State s u -> ParseError -> m b)
               -> (ParseError -> m b)
               -> m b }
type Parsec s u = ParsecT s u Identity
```

Parsec: Why So Complicated?

- I introduced parsing only in the context of Strings, however Parsec is capable of parsing more general things
- However, for most purposes we want to just stick to String. The Text.Parsec.String package contains simpler definitions for this

```
type Parser = Parsec String ()
```

- Using this definition, we can define an example Parser like so

```
parseOne :: Parser Char
parseOne = char '1'
```

Using Parsec: Running A Parser

- Given a parser (i.e a function that returns a ParsecT or Parser type), such as

```
parseOnes :: Parser String
parseOnes = many1 $ char '1'
```

- Use the parse function to execute the parser on a String (just like before)

```
parse :: Stream s Identity t =>
              Parsec s () a   -- the parser
           -> SourceName      -- error file (String)
           -> s               -- input (String)
           -> Either ParseError a -- result
-- example
parse parseOnes "" "111222"
```

Core Parsec Combinators

Most of the combinators we defined last lectures and many more are defined in Text.Parsec. Some very noteworthy ones include

```
(<|>) :: Parser a -> Parser a -> Parser a
-- choice combinator, executes second operand only if
-- first fails WITHOUT CONSUMING ANY INPUT

try :: Parser a -> Parser a
-- allows you to execute a parser and pretend no input
-- has been consumed if it fails (USE WITH <|>)

many :: Parser a -> Parser [a]
-- applies the parser zero or more times

sepBy ::  Parser a -> Parser String -> Parser [a]
-- apply first parser seperated by second parser
-- Example: commaSep p = p `sepBy` (symbol ",")
```

Parsec Char Combinators

Other core combinators that operate specifically on Chars are defined in Text.Parser.Char

```
spaces :: Parser ()
-- skips zero or more spaces (returns nothing)
char :: Char -> Parser Char
-- parses the specified character or fails
anyChar :: Parser Char
-- like char but parse any character
string :: String -> Parser
-- parses the specified whole string or fails
```

See more at https://hackage.haskell.org/package/parsec-3.1.13.0/docs/Text-Parsec-Char.html

Useful Combinators Parsec Should but Doesn't Include

There are a few combinators Parsec arguably should include in the standard library but doesn't. Thankfully they're easy to define

```
symbol :: String -> Parser String
symbol ss = do { spaces;
                 ss' <- string ss;
                 spaces;
                 return ss' }
```

```
parens :: Parser a -> Parser a
parens p = do { char '(';
                cs <- p;
                char ')';
                return cs }
```

Useful Combinators Parsec Should but Doesn't Include

```
digits :: Parser Integer
digits = many1 digit

negDigits :: Parser String
negDigits = do neg <- symbol "-"
               dig <- digits
               return (neg ++ dig)

integer :: Parser Integer
integer = fmap read $ try negDigits <|> digits
```

Challenge: define a Parser for Float

Parsing an Integer Expression

Start by defining Parsers for different operations we want to suppport

```
mulop :: Parser (Integer -> Integer -> Integer)
mulop   =   do{ symbol "*"; return (*)   }
        <|> do{ symbol "/"; return (div) }

addop :: Parser (Integer -> Integer -> Integer)
addop   =   do{ symbol "+"; return (+) }
        <|> do{ symbol "-"; return (-) }
```

Parsing an Integer Expression

We can then compute the expression with the assistance of chainl

```
expr :: Parser Integer
expr    = term   `chainl1` addop

term :: Parser Integer
term    = factor `chainl1` mulop

factor :: Parser Integer
factor  = (parens expr) <|> integer
```