Parsing

Part III: Using the ReadP package

Jim Royer

April 9, 2019

CIS 352

On to ReadP

- ReadP
  - A small, but fairly complete parsing package (shipped with GHC)
  - package docs:
    `http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-ParserCombinators-ReadP.html`
- Parsec
  - A bigger more complete parsing package
  - Unlike ReadP, it can handle errors in an OK fashion.
  - package docs:
    `http://hackage.haskell.org/package/parsec`
  - The Parsec page on the Haskell Wiki:
    `https://wiki.haskell.org/Parsec`

Primitives Repeated from Hutton's Parser.hs

- `get :: ReadP Char`
  Consumes and returns the next character. Fails on an empty input.
- `(<++) :: ReadP a -> ReadP a -> ReadP a`
  Equivalent to Hutton's `+++`. *(`+++` means something else in ReadP.)*
- `pfail :: ReadP a`
  Equivalent to Hutton's `fail`.
- `satisfy :: (Char -> Bool) -> ReadP Char`
  Equivalent to Hutton's `sat`.
- `char :: Char -> ReadP Char`
  Same as in Hutton's
- `string :: String -> ReadP String`
  Same as in Hutton's

First Examples

```
getLetter, openClose :: Parser Char
getLetter = satisfy isLetter

openClose = do { char '('
               ; char ')'
               }
```

```
anbn :: Parser ()
anbn = do { char 'a'
          ; anbn
          ; char 'b'
          ; return ()
          }
       <++ return ()
```

- `getLetter`
  parses the language
  $\{a, b, \ldots, z, A, B, \ldots, Z\}$.
- `openClose`
  parses the language $\{()\}$.
- `anbn`
  parses the language $\{\, a^n b^n \mid n \geq 0 \,\}$?
  (Actually, there are problems.)

Digression: Running your parser

- `readP_to_S :: ReadP a -> String -> [(a,String)]`
  `(readP_to_S p str)` runs parser `p` on `str` and returns the results.

samples.hs

```
    ⋮
sample, openClose :: ReadP Char
sample = satify isLetter

openClose
  = do { char '(' ; char ')' }
    ⋮
```

After loading `samples.hs`

```
*Main> readP_to_S openClose "()"
[(')',"")]

*Main> readP_to_S openClose "(]"
[]
    ⋮
```

In our parser files, we'll usually introduce the alias

```
parse = readP_to_S
```

Two Handy Definitions

```
parse :: ReadP a -> String -> [(a,String)]
parse = readP_to_S
```

```
parseWith :: ReadP a -> String -> a
parseWith p s
    = case [a | (a,t) <- parse p s, all isSpace t] of
        [a] -> a
        []  -> error "no parse"
        _   -> error "ambiguous parse"
```

ReadP's (+++)

- `(+++) :: ReadP a -> ReadP a -> ReadP a`
  `(p1 +++ p2)` runs parses `p1` and `p2` *"in parallel"* and returns the list of results.

  (Not the same as Hutton's `(+++)`!)

  Recall that `(p1 <++ p2)` trys `p1`, and if that fails, trys `p2`.

  Examples

```
*Main> parse (string "ask" +++ string "as") "ask him"
[("as","k him"),("ask"," him")]
```

```
*Main> parse (string "ask" <++ string "as") "ask him"
[("ask"," him")]
```

```
*Main> parse (string "as" <++ string "ask") "ask him"
[("as","k him")]
```

When we mix (+++) and recursion, things get interesting.

as1, as2 :: ReadP String

```
as1 = do { c <- char 'a'
         ; cs <- as1
         ; return (c:cs)
         }
      +++ return ""

as2 = same as as1 but with <++.
```

After loading `samples.hs`

```
*Main> parse as1 "aaaxxx"
[("","aaaxxx"),
 ("a","aaxxx"),
 ("aa","axxx"),
 ("aaa","xxx")]

*Main> parse as2 "aaaxxx"
[("aaa","xxx")]
```

Primitives beyond Hutton's, `munch`, `munch1`

- `many :: (ReadP a) -> (ReadP [a])`

  Parses zero or more occurrences of the given parser
- `many1 :: (ReadP a) -> (ReadP [a])`

  `Parses one or more occurrences of the given parser`
- `munch, munch1 :: (Char -> Bool) -> ReadP String`

  `(munch tst)` is a *greedy* variant of `(many (satisfy tst))`.

  For example:

```
> parse (many (char 'a')) "aaaa"
[("","aaaa"), ("a","aaa"), ("aa","aa"),
 ("aaa","a"), ("aaaa","")]
```

```
> parse (munch (=='a')) "aaaa"
[("aaaa","")]
```

Primitives beyond Hutton's, `munch`, `munch1`

- Greedy ≈ parses as much of the string as possible.
- `munch` and `munch1` use `(<++)`.
- `many` and `many1` use `(+++)`.

Adding Semantics, An Example

```
nesting :: Parser Int
nesting = do { char '('
             ; n <- nesting
             ; char ')'
             ; m <- nesting
             ; return (max (n+1) m)
             }
        +++ return 0
```

[Try `(parse nesting "(())")`, `(parse nesting "()((()())())")`, etc.]

A Few Combinators, 1

Things to look up in the ReadP docs:

- `skipMany` (and friends)
- `between`
- `sepBy` (and friends)
- `endBy` (and friends)

URL: `https://hackage.haskell.org/package/base-4.11.0.0/docs/Text-ParserCombinators-ReadP.html`

A Few Combinators, 2

Simple sentence parsing

```
word :: ReadP String
word = munch1 isLetter

oneOf :: [Char] -> ReadP Char
oneOf cs
  = choice [char c | c <- cs]

separator :: ReadP ()
separator
  = skipMany1 (oneOf " ,")
```

Simple sentence parsing (continued)

```
sentence :: ReadP [String]
sentence
  = do { words <- sepBy1
                    word
                    separator
       ; oneOf ".?!"
       ; return words
       }
```

```
*Main> parse sentence "traffic lights are red, blue, and green."
["traffic","lights","are","red","blue","and","green"]
```

Parsing CSV Files

A CSV parser (from *Real World Haskell*)

CSV: Comma-separated values

A simple file format used by spreadsheets and databases.

See: http://en.wikipedia.org/wiki/Comma-separated_values

A sample

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""","",5000.00
1996,Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

- Commas separate “cells”.
- Unquoted commas are in red.
- Inside quoted text "" is a quoted quote.
- Lines normally end with a newline, but quoted text can cross line boundries.

A Grammar for CSV

$$
\begin{aligned}
\langle \text{file} \rangle &::= \langle \text{line} \rangle^* \\
\langle \text{line} \rangle &::= ((\langle \text{cell} \rangle,)^* \langle \text{cell} \rangle)^? \langle \text{newline} \rangle \\
\langle \text{cell} \rangle &::= \langle \text{character} \rangle^+ \mid \langle \text{quotedCell} \rangle \\
\langle \text{quotedCell} \rangle &::= "\langle \text{quotedChar} \rangle^*" \\
\langle \text{quotedChar} \rangle &::= \langle \text{notQuote} \rangle \mid "" \\
\langle \text{notQuote} \rangle &::= \textit{everything but } " \\
\langle \text{newline} \rangle &::= \backslash n \backslash r \mid \backslash r \backslash n \mid \backslash n \mid \backslash r \\
\langle \text{character} \rangle &::= a \mid b \mid \ldots
\end{aligned}
$$

Note: $A^? \equiv A \mid \epsilon \equiv$ 0 or 1 copies of A

[Stage direction: Copy the grammar to the board.]

A parser for CSV, 1

$$\langle \text{file} \rangle ::= \langle \text{line} \rangle^*$$
$$\langle \text{newline} \rangle ::= \backslash n \backslash r \mid \backslash r \backslash n \mid \backslash n \mid \backslash r$$

```
csvFile :: ReadP [[String]]
csvFile = endBy line eol
```

```
eol :: ReadP String
eol = (string "\n\r")
      <++ (string "\r\n")
      <++ (string "\n")
      <++ (string "\r")
```

A parser for CSV, 2

$$\langle \text{cell} \rangle ::= \langle \text{character} \rangle^{+} \mid \langle \text{quotedCell} \rangle$$
$$\langle \text{character} \rangle ::= a \mid b \mid \ldots$$

```
line :: ReadP [String]
line = sepBy cell (char ',')
```

```
cell :: ReadP String
cell = quotedCell
       <++ munch (`notElem` ",\n\r")
```

A parser for CSV, 3

$$\langle \text{quotedCell} \rangle ::= \text{"}\langle \text{quotedChar} \rangle^*\text{"}$$
$$\langle \text{quotedChar} \rangle ::= \langle \text{notQuote} \rangle \mid \text{""}$$
$$\langle \text{notQuote} \rangle ::= \textit{everything but "}$$

```
quotedCell :: ReadP String
quotedCell = between (char '"')
                     (char '"')
                     (many quotedChar)
```

```
quotedChar :: ReadP Char
quotedChar =
    satisfy (/= '"')
    +++ (string "\"\"" >> return '"')
```

A parser for CSV, 4

All on one page

```
csvFile :: ReadP [[String]]
csvFile = endBy line eol
```

```
line :: ReadP [String]
line = sepBy cell (char ',')
```

```
eol :: ReadP String
eol = (string "\n\r")
      <++ (string "\r\n")
      <++ (string "\n")
      <++ (string "\r")
```

```
cell :: ReadP String
cell =
    quotedCell
    <++ munch (`notElem` ",\n\r")
```

```
quotedCell :: ReadP String
quotedCell =
  between (char '"')
          (char '"')
          (many quotedChar)
```

```
quotedChar :: ReadP Char
quotedChar =
    satisfy (/= '"')
    +++ (string "\"\"" >> return '"')
```

Parser combinators (other than <++ and +++) are in **bold**.

A parser for CSV, 5

```
parseCSV ::  String -> [([[String]], String)]
parseCSV input = parse csvFile input
```

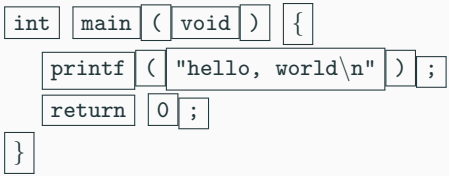
```
parseFile :: FilePath -> IO ()
parseFile name =
    do c <- readFile name
       mapM_ print (parseWith csvFile c)
```

Tokens

Token based parsing

Tokens ≈ Variable names, numerals, operators key-words, . . .

```
int main(void) {
    printf("hello, world\n");
    return 0;
}
```



- Sometimes white space is needed to separate tokens
  Example: ”`return 0`” versus ”`return0`”
- Otherwise, there can be any amount of space between tokens.

Parsing strategy

- Start with the first non-space character
- Repeatedly grab a token and then skip any following whitespace.

A Tour of `Parser1.hs` which parses

$$expr ::= aexpr \mid aexpr\,?\,aexpr\ :\ expr$$

$$aexpr ::= term \left\{ \{+ \mid -\}\, term \right\}^*$$

$$term ::= factor \left\{ \{* \mid /\}\, factor \right\}^*$$

$$factor ::= num \mid (expr)$$

Things to look up in the ReadP docs:

- `option`
- `chainl1` (and friends)

[See:
`http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-ParserCombinators-ReadP.html`]

A Tour of `LCparserP.hs` which parses LC

Phases	P	$::=$	$A \mid B \mid C$
Arithmetic Expressons	A	$::=$	$n \mid !\ell \mid A \circledast A \quad (\circledast \in \{+, -, *, \dots\})$
Boolean Expressons	B	$::=$	$b \mid A \circledast A \quad (\circledast \in \{=, <, >=, \dots\})$
Commands	C	$::=$	$\textbf{skip} \mid \ell := A \mid C; C$
		$\mid$	$\textbf{if } B \textbf{ then } C \textbf{ else } C \mid \textbf{while } B \textbf{ do } C$

Integers	$n \in \mathbb{Z}$	$= \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
Booleans	$b \in \mathbb{B}$	$= \{\text{tt}, \text{ff}\}$
Locations	$\ell \in \mathbb{L}$	$= \{x0, x1, x2, \dots\}$

Things to look up in the ReadP docs:

- `choice`

Now you are ready to parse a *(close-to)* real programming language.