CIS552: Advanced Programming

Handout 8
Functional Parsers
A parser is a program that analyzes a piece of text to determine its structure (and, typically, returns a tree representing this structure).
Almost every real-life program involves some kind of parsing...

- Hugs and GHC parse Haskell programs
- Unix shells (bash, sh, etc.) parse shell scripts
- Explorer, Mozilla, etc., parse HTML
- Command-line utilities parse command lines
- etc., etc.
In Haskell, a parser is naturally viewed as a function:

```haskell
type Parser = String -> Tree
```
However, a parser might not actually use up the whole string, so we also return the unused portion of the input:

\[
\text{type Parser} = \text{String} \rightarrow (\text{Tree}, \text{String})
\]
Also, a given string might be parseable in many ways (including zero!), so we generalize to a list of results:

```haskell
type Parser = String -> [(Tree, String)]
```
The result returned by a parser might not always be a tree, so we generalize once more to make the `Parser` type polymorphic:

```haskell
type Parser a = String -> [(a,String)]
```
Finally, for the sake of readability, let’s change the `type` declaration into a `newtype` and add a constructor on the right-hand side. The convenience function `parse` takes a parser and applies it to a given string.

```haskell
newtype Parser a = Parser (String -> [(a, String)])

parse :: Parser a -> String -> [(a, String)]
parse (Parser p) = p
```
Primitive Parsers
The parser `item` fails if the input is empty, and consumes the first character otherwise:

```haskell
item :: Parser Char
item = Parser (
  \cs -> case cs of
    "" -> []
    (c:cs) -> [(c,cs)]
)
```

```
parse item "hello"
⇒ [(‘h’,"ello")]
```

```
parse item ""
⇒ []
```
The parser `returnP a` always succeeds, returning the value `a` without consuming any input:

```haskell
returnP :: a -> Parser a
returnP a = Parser (\cs -> [(a,cs)])
```

```haskell
parse (returnP 5) "hello"
⇒ [(5,"hello")]
```
Putting Parsers in Sequence

\( \text{putting parsers in sequence} \)

\( p \text{ `seqP` } q \) is a parser that first applies \( p \) and then applies \( q \) to each result from \( p \).

\[
\begin{aligned}
\text{seqP} & : \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b \\
p \text{ `seqP` } q &= \\
\text{Parser} \\
(\text{\textbackslash cs} \rightarrow \text{concat } \left[ \text{parse } (q \text{ a}) \text{ cs'} \\
| (a,\text{cs'}) \leftarrow \text{parse } p \text{ cs} \right])
\end{aligned}
\]
Example

parseTwo :: Parser (Char,Char)

parseTwo =
    item
    'seqP' \x -> item
    'seqP' \y -> return (x,y)

parse parseTwo "hello"
⇒ [((‘h’,’e’),"llo")]

parse parseTwo "h"
⇒ []

Note that, if any parser in a sequence fails, then the whole sequence fails.
Parsers Are a Monad
The definitions of \texttt{returnP} and \texttt{seqP} have the right types (and obey the required laws) to make \texttt{Parser} into a monad.

\begin{verbatim}
instance Monad Parser where
  return = returnP
  (>>=)  = seqP
\end{verbatim}
Having made this instance declaration, we can use do syntax to simplify the presentation of the `parseTwo` function:

```haskell
parseTwo2 :: Parser (Char,Char)
parseTwo2   = do x <- item
                y <- item
                return (x,y)
```
More Primitives
The parser \texttt{zeroP} always fails:

\begin{verbatim}
zeroP :: Parser a

zeroP = Parser (\cs -> [])
\end{verbatim}
The parser \texttt{sat p} behaves like \texttt{item} if the first character on the input string satisfies the predicate \texttt{p}; otherwise it fails.

\begin{verbatim}
\texttt{sat} :: (Char \rightarrow \textbf{Bool}) \rightarrow \textbf{Parser} \textbf{Char}
\texttt{sat p} = do c <- \texttt{item}
    if \texttt{p c} then return c else \texttt{zeroP}
\end{verbatim}

\begin{verbatim}
parse (\texttt{sat (=='h')}) "hello"
⇒ [\texttt{('h',"ello")}]

parse (\texttt{sat (=='x')}) "hello"
⇒ []
\end{verbatim}
Examples

\[
\begin{align*}
\text{char} &::= \text{Char} \rightarrow \text{Parser Char} \\text{char} \ c \ &= \ \text{sat} \ (c ==) \\
\text{alphachar} &::= \text{Parser Char} \ 
\text{alphachar} \ &= \ \text{sat} \ \text{isAlpha} \\
\text{numchar} &::= \text{Parser Char} \ 
\text{numchar} \ &= \ \text{sat} \ \text{isDigit} \\
\text{digit} &::= \text{Parser Int} \ 
\text{digit} \ &= \ \text{do} \ \{ \ c \ <- \ \text{numchar}; \ \text{return} \ (\text{ord} \ c - \ \text{ord} \ '0') \}\)
\end{align*}
\]

(isAlpha and isDigit come from the Char module in the standard library.)
Nondeterministic Choice

\( p \ 'chooseP' \ q \) yields all the results of applying either \( p \) or \( q \) to the whole input string.

\[
\text{chooseP} :: \text{Parser } a \to \text{Parser } a \to \text{Parser } a
\]

\[
p \ 'chooseP' \ q = \text{Parser} \left( \lambda \text{cs} \to \text{parse } p \ \text{cs} \ +\ + \ \text{parse } q \ \text{cs} \right)
\]

\[
\text{alphanum} :: \text{Parser } \text{Char}
\]

\[
\text{alphanum} = \text{alphachar} \ 'chooseP' \ \text{numchar}
\]
Another Example

```
p = do { x <- item; return ("Got "+[x]) }

'chooseP' 
do { x <- item; return ("Parsed "+[x]) }

parse p "xyz"
⇒ [("Got x","yz"),("Parsed x","yz")]
```
This parser yields a function:

```haskell
addop :: Parser (Int -> Int -> Int)
addop = do {char '+'; return (+)}
         'chooseP'
         do {char '-' ; return (-)}
```

For example:

```haskell
calc = do x <- digit; op <- addop; y <- digit
         return (x `op` y)
```

```haskell
parse calc "1+2"
⇒ [(3,"")]```
Recursive Parsers
string $s$ is a parser that recognizes (and returns) exactly the string $s$:

\[
\begin{align*}
\text{string} & \quad : \quad \text{String} \rightarrow \text{Parser String} \\
\text{string } "" & \quad = \quad \text{return } ""
\text{string } (c:cs) & \quad = \quad \text{do} \{ \text{char } c; \text{ string } cs; \text{ return } (c:cs) \}
\end{align*}
\]
many :: Parser a -> Parser [a]
many p = many1 p `chooseP` return []

many1 :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}

parse (many numchar) "123ab"
⇒ [("123","ab"),("12","3ab"),("1","23ab"),("","123ab")]
A Parser for Arithmetic Expressions

calc1 = do x <- digit
          op <- addop
          y <- calc1
          return (x ‘op’ y)
‘chooseP’
digit

parse calc1 "3+4-1"
⇒ [(6,""), (7,"-1"), (3,"+4-1")]

Note that, for simplicity, we’re taking + and – to be right-associative for the moment.

Query: What happens if we exchange the arguments to chooseP?
A Complete Parser
Multiplication-like Operators

As before...

```
mulop :: Parser (Int -> Int -> Int)
mulop = do {char '*'; return (*)}
    'chooseP'
    do {char '/'; return (div)}
```
expr = do x <- term; op <- addop; y <- expr
    return (x 'op' y)
    'chooseP'
    term

term = do x <- factor; op <- mulop; y <- term
    return (x 'op' y)
    'chooseP'
    factor

factor = digit
    'chooseP'
    do {char '('; n <- expr; char ')'; return n}

parse expr "(3+4)*5"
⇒ [(35,""),(7,"*5")]

Complete Parser
Note the similarity in the definitions of `expr` and `term`.

```haskell
expr = do x <- term; op <- addop; y <- expr
    return (x `op` y) ‘chooseP’
term

term = do x <- factor; op <- mulop; y <- term
    return (x `op` y) ‘chooseP’
factor
```

Can we express them both as instances of a common abstraction?
The parser `chainl p op` consumes a non-empty sequence of `ps` from the front of the input and combines them together (in the style of `foldl`) using `op`.

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

A similar chaining function also works for empty sequences:

```haskell
chainl :: Parser a -> Parser (a -> a -> a) -> a
     -> Parser a
chainl p op a =
  (p 'chainl1' op) 'chooseP' return a
```
expr2, term2, factor2 :: Parser Int

expr2 = term2 'chainl1' addop
term2 = factor2 'chainl1' mulop
factor2 = digit
  'chooseP'  
    do {char '('; n <- expr2; char ')'; return n}
As a side-benefit, our new expression parser also makes subtraction and division (and addition and multiplication) left-associative:

```
parse expr "9-3-2" -- old
⇒ [(8,""),(6,"-2"),(9,"-3-2")]

parse expr2 "9-3-2" -- new
⇒ [(4,""),(6,"-2"),(9,"-3-2")]
```
Efficiency
Usually, we are interested in getting just one parse of the input string, not all possible parses.

The parser $p +++ q$ yields just the first result from by $p$, if any, and otherwise the first result from $q$.

```
(++++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p `chooseP` q) cs of
                    []    -> []
                    (x:xs) -> [x])
```
We can now redefine `many` in terms of `+++`.

```haskell
many :: Parser a -> Parser [a]
many p = many1 p +++ return []
```

```haskell
many1 :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```

This change ensures that `many` always returns exactly one result.
Similarly, we can redefine \texttt{chainl1} and \texttt{chainl} in terms of +++.

\texttt{chainl1 :: Parser a \rightarrow Parser (a \rightarrow a \rightarrow a) \rightarrow Parser a}

\texttt{p 'chainl1' op =}
\texttt{ do \{a <- p; rest a\}}
\texttt{ where}
\texttt{ rest a = do \{f <- op; b <- p; rest (f a b)\}}
\texttt{ +++ return a}

\texttt{chainl :: Parser a \rightarrow Parser (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow Parser a}

\texttt{chainl p op a =}
\texttt{(p 'chainl1' op) +++ return a}
Wrap Up
Parsing technology is a large and complex research area, extending back to the 1950s and still continuing today. (E.g., see many recent papers on “Generalized LR parsing,” “packrat parsing”, etc.)

Functional parsing is also an active research topic, whose surface we have just scratched here.

- further efficiency improvements
- error reporting and correction
- infix operator precedence
- support for “almost deterministic” grammars
**MonadPlus** is an extension of the **Monad** class that adds a couple of extra operations. It is not as critical as **Monad**, but there are some library functions that rely on **MonadPlus** for a few useful things.

```haskell
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

**Parsers are an instance of MonadPlus:**

```haskell
instance MonadPlus Parser where
  mzero = zeroP
  mplus = chooseP
```