What is a Parser?

A parser is a program that analyzes a piece of text to determine its structure (and, typically, returns a tree representing this structure).

The World is Full of Parsers!

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.

Functional Parsers

In Haskell, a parser is naturally viewed as a function:

```
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:

```
type Parser = String -> (Tree,String)
```
### Functional Parsers

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

Parsing an Arbitrary Character

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 ""
⇒ []
```

Parsing Nothing (Successfully)

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)])
```

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

\[ p \text{ `seqP` } q \]

is a parser that first applies \( p \) and then applies \( q \) to each result from \( p \).

\[
\text{seqP} :: \text{Parser } a \to (a \to \text{Parser } b) \to \text{Parser } b
\]

\[
p \text{ `seqP` } q = \\
\text{Parser} \ \\
(\text{\textbackslash}cs \to \text{concat} [\text{parse} (q a) cs'] \\
| (a,cs') \leftarrow \text{parse } p \ cs)
\]

Example

\[
\text{parseTwo} :: \text{Parser } (\text{Char},\text{Char})
\]

\[
\text{parseTwo} = \ \\
\text{item} \\
\text{'seqP' } \text{\textbackslash}x \to \text{item} \\
\text{'seqP' } \text{\textbackslash}y \to \text{return } (x,y)
\]

parse parseTwo "hello"
\[
\Rightarrow [(('h','e'),"llo")]
\]

parse parseTwo "h"
\[
\Rightarrow []
\]

Note that, if any parser in a sequence fails, then the whole sequence fails.

The Parser Monad

The definitions of \text{returnP} and \text{seqP} have the right types (and obey the required laws) to make \text{Parser} into a monad.

instance Monad Parser where
\[
\text{return} = \text{returnP} \\
(\text{>>=} ) = \text{seqP}
\]

The Parser Monad

Having made this instance declaration, we can use \text{do} syntax to simplify the presentation of the \text{parseTwo} function:

\[
\text{parseTwo2} :: \text{Parser } (\text{Char},\text{Char})
\]

\[
\text{parseTwo2} = \text{do } x \leftarrow \text{item} \\
y \leftarrow \text{item} \\
\text{return } (x,y)
\]

More Primitives
 Parsing Nothing (Unsuccessfully)

The parser `zeroP` always fails:

```
zeroP :: Parser a
zeroP = Parser (\cs -> [])
```

 Parsing a Character If It Satisfies Some Test

The parser `sat p` behaves like `item` if the first character on the input string satisfies the predicate `p`; otherwise it fails.

```
sat :: (Char -> Bool) -> Parser Char
sat p = do c <- item
  if p c then return c else zeroP
```

```
parse (sat (=='h')) "hello"
⇒ [('h',"ello")]
parse (sat (=='x')) "hello"
⇒ []
```

 Examples

```
char :: Char -> Parser Char
char c = sat (c ==)
alphachar :: Parser Char
alphachar = sat isAlpha
numchar :: Parser Char
numchar = sat isDigit
digit :: Parser Int
digit = do {c <- numchar; return (ord c - ord '0')}
```

(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.

chooseP :: Parser a -> Parser a -> Parser a
chooseP p q = Parser (\cs -> parse p cs ++ parse q cs)
```

```
alphanum :: Parser Char
alphanum = alphachar 'chooseP' 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"]
```

 Yet Another Example

This parser yields a function:

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

```
addop = do {char '+'; return (+)}
  'chooseP'
do {char '-' ; return (-)}
```

For example:

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

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

```
parse calc "1+2"
⇒ [(3,"" problems)]
```
Recursive Parsers

Regonizing a String

string s is a parser that recognizes (and returns) exactly the string s:

\[
\text{string} :: \text{String} \rightarrow \text{Parser String}
\]

\[
\text{string} \ "" = \text{return} \ ""
\]

\[
\text{string} \ (c:cs) = \text{do} \ \{ \text{char} \ c; \ \text{string} \ cs; \ \text{return} \ (c:cs) \}
\]

Parsing a Sequence

many :: Parser a \rightarrow\ Parser [a]

\[
\text{many} \ p = \text{many1} \ p \ "\chooseP\" \ \text{return} \ [\]
\]

many1 :: Parser a \rightarrow\ Parser [a]

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

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)

\text{"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 \text{chooseP}?

A Complete Parser

Multiplication-like Operators

As before...

mulop :: Parser (Int \rightarrow\ Int \rightarrow\ Int)

\[
\text{mulop} = \text{do} \ \{ \text{char} \ \"\"; \ \text{return} \ (**\})
\]

\text{\"chooseP\"}

\[
\text{do} \ \{ \text{char} \ \"/\"; \ \text{return} \ (\div)\}
\]

\[
\text{mulop} \end{equation}
**Complete Parser**

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

term = do x <- factor; op <- mulop; y <- term
  return (x 'op' y)
'factor'
factor = digit
  'chooseP'
do {char '('; n <- expr; char ')'; return n}
```

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

**A Little More Abstraction**

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

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

term = do x <- factor; op <- mulop; y <- term
  return (x 'op' y)
'factor'
Can we express them both as instances of a common abstraction?
```

**A Better Arithmetic Expression Parser**

```haskell
expr2, term2, factor2 :: Parser Int
expr2 = term2 'chainl1' addop
term2 = factor2 'chainl1' mulop
factor2 = digit
  'chooseP'
do {char '('; n <- expr2; char ')'; return n}
```

```haskell
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")]
```
Deterministic Choice

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 $p$, if any, and otherwise the first result from $q$.

$$
(++) :: \text{Parser } a \to \text{Parser } a \\
p +++ q = \text{Parser } (\\text{\textbackslash}cs \to \text{case parse } (p \ '\text{chooseP'} q) \ cs \ of \\
\hspace{1em}[] \to [] \\
\hspace{1em}(x:xs) \to [x])
$$

More Efficient Sequencing

We can now redefine $\text{many}$ in terms of $+++$.

$$
\text{many} :: \text{Parser } a \to \text{Parser } [a] \\
\text{many } p = \text{many1 } p +++ \text{return } []
$$

$$
\text{many1} :: \text{Parser } a \to \text{Parser } [a] \\
\text{many1 } p = \text{do } \{a \leftarrow p; \ as \leftarrow \text{many } p; \ \text{return } (a:as)\}
$$

This change ensures that $\text{many}$ always returns exactly one result.

More Efficient Chaining

Similarly, we can redefine $\text{chainl}$ and $\text{chainl1}$ in terms of $+++$.

$$
\text{chainl1} :: \text{Parser } a \to \text{Parser } (a \to a \to a) \\
\to \text{Parser } a \\
p \ '\text{chainl1'} op = \\
\hspace{1em}\text{do } \{a \leftarrow p; \ \text{rest } a\} \\
\hspace{2em}\text{where} \\
\hspace{3em}\text{rest } a = \text{do } \{f \leftarrow op; \ b \leftarrow p; \ \text{rest } (f \ a \ b)\} \\
\hspace{3em}+++ \text{return } a
$$

$$
\text{chainl} :: \text{Parser } a \to \text{Parser } (a \to a \to a) \to a \\
\to \text{Parser } a \\
\text{chainl } p \ op \ a = \\
\hspace{1em}(p \ '\text{chainl1'} \ op) +++ \text{return } a
$$

Wrap Up

More on Functional Parsing

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

The MonadPlus Class

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.

$$
class \text{Monad } m \Rightarrow \text{MonadPlus } m \ where \\
\hspace{1em}mzero :: m \ a \\
\hspace{1em}mplus :: m \ a \to m \ a \to m \ a
$$

Parsers are an instance of MonadPlus:

$$
\text{instance MonadPlus } \text{Parser} \ where \\
\hspace{1em}mzero = \text{zeroP} \\
\hspace{1em}mplus = \text{chooseP}
$$