15 Building a parser

A *parser* takes a concrete representation of some abstract data and identifies the abstract object. You can think of it as an inverse to *show*, and indeed there is a predefined polymorphic $read :: Read\ a \Rightarrow String \rightarrow a$. How might you write instances of *Read* for complex objects?

The sort of thing we want to be able to do is

```
*Main> apply expr "12*(3+4)"
Just 84
```

where the language of the string is described by a little grammar

```
> expr, term, factor :: Parser Int
> expr   = term   `chain` addop
> term   = factor `chain` mulop
> factor = digits <|> parens (symb "(") expr (symb ")")
```

More usually the parser might produce an abstract syntax tree for an expression, and evaluating it might be a separate job, but this will do for now.

A parser for things of type a must take a string and consume enough of the string to find a representation of an a. It should also hand back the rest of the string in case the context is looking for an a followed by something else. So perhaps a parser for a maps a *String* to a pair $(a, String)$.

However there might be some ambiguity: "$1 + 2 + 3$" might be parsed into 1 and "$+2 + 3$", or 3 and "$+3$", or 6 and "" (the empty string), so our parsers will return a list of possible parses:

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

15.1 Primitive parsers

The simplest parser consumes none of its input, and just returns a value:

```
> return :: a -> Parser a
> return x xs = [(x,xs)]
```

and an *item* is the next single character, if there is one:

```
> item :: Parser Char
> item    []  = []
> item (x:xs) = return x xs
```

More usefully we might want to accept in a parse only a character that satisfies some predicate:

```
> sat :: (Char -> Bool) -> Parser Char
> sat p = p <?> item

> (<?>) :: (a -> Bool) -> Parser a -> Parser a
> (c <?> p) xs = [ (a,ys) | (a,ys) <- p xs, c a ]
```

The (<?>) operator (read *satisfying*) generalises this idea to an analogue of *filter* on parsers other than *item*. For example

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

matches exactly the given character from the input string.

15.2 Sequencing parsers

More generally

```
> string :: String -> Parser String
> string    ""  = return ""
> string (c:cs) = char c >> string cs >> return (c:cs)
```

matches exactly the sequence of characters in a string by matching first the head, and then (by recursion) the tail.

The sequencing operator ($\gg$) runs two parsers in sequence

```
> (>>)  :: Parser a -> Parser b -> Parser b
> (p >> q) xs = [ (b,zs) | (a,ys) <- p xs, (b,zs) <- q ys ]
```

discarding the result of the first one, and returning the result of running the second on that part of the string not already consumed by the first parser. (Notice that the *string* parser returns the string that was being matched, not something composed of the parsed values.)

More generally we might want not to discard the parsed value from the first parser. This operator

```
> (>>=) :: Parser a -> (a -> Parser b) -> Parser b
> (p >>= f) xs = [ (b,zs) | (a,ys) <- p xs, (b,zs) <- f a ys ]
```

(pronounced *bind* for reasons that might become more obvious in the next lecture) parses an a using the first parser p and then applies f to a to find a second parser to produce the final result. Typically, the $f\ a$ runs a parser essentially independent of the value a, and then combines a with the result of that parse. Of course, $p \gg q = p \gg\!= const\ q$.

15.3 Alternative parsers

A choice operator has type $Parser\ a \to Parser\ a \to Parser\ a$ and produces a parse from one or the other of its arguments. For example

```
> (<++>) :: Parser a -> Parser a -> Parser a
> (p <++> q) xs = p xs ++ q xs
```

is a sort of non-deterministic choice: $p <\!\!+\!\!+\!\!> q$ will parse and return anything that can be parsed by either p or by q. Usually we will be interested only in the best parse, and if we make sure that the left argument parser is more specific than the right argument, we will prefer a deterministic choice:

```
> (<|>) :: Parser a -> Parser a -> Parser a
> (p <|> q) = take 1 . (p <++> q)
```

which parses the best match that comes from the first parser, and only resorts to the second if the first fails entirely.

For example, if *some* p parses a seqence of one or more of the things parsed by p, then

```
> many :: Parser a -> Parser [a]
> many p = some p <|> return []
```

will parse none or more, that is a potentially empty sequence of p. Of course, *some* is defined in terms of *many*.

```
> some :: Parser a -> Parser [a]
> some p = p >>= ps
>          where ps a = many p >>= done
>                       where done as = return (a:as)
```

A non-empty sequence consists of a head parsed by p and a tail parsed by ps. The nesting of the local definitions is just about labelling the value of the head as a and the tail as as. If you are familiar with lambda expressions, you might prefer

```
> some p = p >>= (\a -> many p >>= (\as -> return (a:as)))
```

however I think this is just a sign that we need an even better notation. Again, I will return to this in the next lecture.

A concrete example would be a parser that matches whitespace

```
> space :: Parser String
> space = many (sat isSpace)
```

This might be part of a parser that matches some token, followed by white space which is ignored

```
> token :: Parser a -> Parser a
> token p = p >>= done
>           where done a = space >> return a
```

for example a symbol matching a given string

```
> symbol :: String -> Parser String
> symbol xs = token (string xs)
```

For a concrete example, *symbol* "+" would match an addition operator. However this would return a '+' character as the corresponding value, and we would want to have the addition function (+).

```
> addop = ((+) <$ symbol "+") <|> ((-) <$ symbol "-")
> mulop = ((*) <$ symbol "*") <|> (div <$ symbol "/")

> (<$) :: b -> Parser a -> Parser b
> x <$ p = const x <$> p

> (<$>) :: (a -> b) -> (Parser a -> Parser b)
> (f <$> p) xs = [ (f x, ys) | (x,ys) <- p xs ]
```

The type of (<$>) is deliberately reminiscent of the type of *map* and *fmap*, and indeed if we had not carefully hidden it, (<$>) is a predefined synomym for *fmap*.

The same mechanism can translate strings of digits

```
> digits :: Parser Int
> digits = foldl shift 0 <$> token (some digit)
>          where shift n d = 10*n+d

> digit :: Parser Int
> digit = fromDigitChar <$> sat isDigit
>         where fromDigitChar x = ord x - ord '0'
```

The function *isDigit* is defined in *Data . Char*, as is *ord* which is a traditional name for the instance of *fromEnum* for the type *Char*.

Finally to complete the functions used in the example grammar

```
> parens :: Parser b -> Parser a -> Parser c -> Parser a
> parens open p close = open >> p >>= rest
>                       where rest n = close >> return n
```

parses a p between two symbols which act as parethneses. This produces the same value as p but the concrete syntax is different.

The other function represents a number of p, each pair separated by a *op*, and associating to the left.

```
> chain :: Parser a -> Parser (a->a->a) -> Parser a
> chain p op = p >>= rest
>              where rest a = (op >>= q) <|> return a
>                     where q f = p >>= done
>                            where done b = rest (f a b)
```

The left association comes from the way that *rest* recurses, reminiscent of *foldl*. As before the mechanism required to carry around the results makes this difficult to read, and a better notation would be helpful.

15.4 Running the parser

A parser is run by applying it to the string that is to be parsed; here with *leading* space ignored.

```
> apply :: Parser a -> String -> Maybe a
> apply p xs =
>     case [ v | (v,ys) <- (space >> p) xs, null ys ] of
>         [v] -> Just v
>         _   -> Nothing
```

This returns a *Maybe* in case the parse is unsuccessful.

The parse is successful if it consumes all of the string, and is unambiguous. It is unsuccessful if either there is something left over, or if there are two possible parses.