A Quick Guide to the Parser Library

The functions in the parser library are intended for constructing *parsers* easily. What, then, is a parser? According to this library, it is just a function with the type

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

That is, you can invoke a parser (call the function), supplying a string to parse as input, and the parser will then produce a *list of alternative parses*. Each parse in the list consists of a value of type a, the *value parsed*, and a string representing the unparsed part of the input. You can think of each result as a way of splitting the original input into two: something of the type the parser is looking for, and the rest. For example,

```
Calculator> number "123"
[(123,""),(12,"3"),(1,"23")]
```

Given the string "123", we can recognise 12, followed by "3", among other possibilities. (The reason for returning a *list* of alternatives is so as to be able to support ambiguous grammars).

Now that we know what a parser is, we should think of Parser a as an *abstract type*—and the parser library defines functions that let us build and use parsers without even knowing what type is used to represent them. All code that *uses* the parsing library should be written so that it will still work , if the representation type used for parsers is changed.

parse

```
parse :: Parser a -> String -> [a]
```

The parse function is used to *invoke* a parser on a string. When we *use* a parser, we generally want to parse the *entire* input string—returning the unparsed part of the input is generally of interest only inside the parsing library itself. Therefore parse keeps only alternatives that recognise the entire input string, and just returns a list of the alternative values parsed. (Usually this list will have one or no elements).

value

```
value :: a -> Parser a
```

value x creates a parser that just delivers x as its result, without actually consuming any input at all. For example, if we are trying to parse zero-or-more occurrences of something, and there are actually none of those things in the input, then we can use value [] to generate an empty list, without actually consuming any input.

satisfy

```
satisfy :: (Char -> Bool) -> Parser Char
```

satisfy p creates a parser that tries to parse *one character* satisfying the predicate p (i.e. for which the function p returns True). So satisfy isDigit tries to parse one digit. It can succeed or fail—but will never find more than one alternative parse.

exactly

```
exactly :: String -> Parser String
```

exactly s creates a parser that tries to parse exactly the string s. If the input begins with s, then exactly s succeeds, otherwise it fails.

|||

(|||) :: Parser a -> Parser a -> Parser a

p ||| q combines two parsers p and q, into a parser that tries both. p and q are tried on *the same input*, and all of the results that either of them produces becomes part of the result of p ||| q. Normally p and q are *alternatives*, so we expect that at most one will succeed—although nothing requires this.

@@

```
(@@) :: Parser (a->b) -> Parser a -> Parser b
```

p @@ q combines two parsers p and q into a parser that uses both *in sequence*. Thus @@ first uses p to parse the first part of the input, then uses q to parse what follows. This is the basic operation used to parse a sequence of things. The value that p parses is required to be a *function*, of the right type to apply to the value that q parses. The value delivered by p @@ q is then obtained by applying the function that p produces to the argument that q produces.

Suppose we want to parse three things, using parsers p, q, and r, and combine the results using the function f. We can achieve this with the following code:

```
value f @@ p @@ q @@ r
```

which is interpreted as ((value f @@ p) @@ q) @@ r. First value f generates the function f, without consuming any input. Then p generates its result, let's call it x, and the first @@ applies f to x, delivering f x. Now q parses the next bit of input, let's say generating y. The second @@ takes the value delivered by (value f @@ p), that is f x, and applies it to y, delivering f x y. Finally the third @@ invokes r to get its result, say z, and applies f x y to z, delivering f x y z—the value we wanted.

##

```
(##) :: Parser a -> Parser b -> Parser a
```

p ## q is like p @@ q, except that the value parsed by q is just thrown away. The value that p ## q delivers is the same value that p delivers. This is useful for parsing "noise" symbols that may be important in the syntax we're parsing, but are not needed for the actual parse result. For example, when we parse negative numbers, then the "-" sign needs to be recognised, but once we've done so, then we don't need to include that string in the result we deliver. So we could write a parser for negative numbers that uses ## to discard the minus sign—remembering, of course, to include a function to negate the value that we parse. Here is such a parser:

```
value negate ## exactly “-“ @@ number
```

This brackets as (value negate ## exactly “-“) @@ number, where the ## parses and discards a minus sign, delivering the result from value negate—that is the negation function. Then the @@ applies this function to the positive number following the “-“, parsed by number.

many

```
many :: Parser a -> Parser [a]
```

many p parses zero or more of the things that p parses, delivering a list of the results.

some

```
some :: Parser a -> Parser [a]
```

some p parses one or more of the things that p parses, delivering a non-empty list of the results.