chapter 8

Functional parsers

In this chapter we show how Haskell can be used to program simple parsers. We start by explaining what parsers are and why they are useful, show how parsers can naturally be viewed as functions, define a number of basic parsers and higher-order functions for combining parsers in various ways, and conclude by developing a parser for arithmetic expressions.

8.1 Parsers

A *parser* is a program that takes a string of characters, and produces some form of tree that makes the syntactic structure of the string explicit. For example, given the string $2 * 3 + 4$, a parser for arithmetic expressions might produce a tree of the following form, in which the numbers appear at the leaves of the tree, and the operators appear at the branches:

$$\begin{tikzcd}
 & & + \arrow[dl] \arrow[dr] & \\
 & * \arrow[dl] \arrow[dr] & & 4 \\
2 & & 3 &
\end{tikzcd}$$

The structure of this tree makes explicit that $+$ and $*$ are operators with two arguments, and that $*$ has higher priority than $+$.

Parsers are an important topic in computing, because most real-life programs use a parser to pre-process their input. For example, a calculator program parses numeric expressions prior to evaluating them, the Hugs system parses Haskell programs prior to executing them, and a web browser parses hypertext documents prior to displaying them. In each case, making the structure of the input explicit considerably simplifies its further processing. For example, once a numeric expression has been parsed into a tree structure such as in the example above, evaluating the expression is straightforward.

8.2 The parser type

In Haskell, a parser can naturally be viewed directly as a function that takes a string and produces a tree. Hence, given a suitable type *Tree* of trees, the notion of a parser can be represented as a function of type *String* → *Tree*, which we abbreviate as *Parser* using the following declaration:

```
type Parser  =  String → Tree
```

In general, however, a parser might not always consume its entire argument string. For example, a parser for numbers might be applied to a string comprising a number followed by a word. For this reason, we generalise our type for parsers to also return any unconsumed part of the argument string:

```
type Parser  =  String → (Tree, String)
```

Similarly, a parser might not always succeed. For example, a parser for numbers might be applied to a string comprising a word. To handle this, we further generalise our type for parsers to return a list of results, with the convention that the empty list denotes failure, and a singleton list denotes success:

```
type Parser  =  String → [(Tree, String)]
```

Returning a list also opens up the possibility of returning more than one result if the argument string can be parsed in more than one way. For simplicity, however, we only consider parsers that return at most one result.

Finally, different parsers will likely return different kinds of trees, or more generally, any kind of value. For example, a parser for numbers might return an integer value. Hence, it is useful to abstract from the specific type *Tree* of result values, and make this into a parameter of the *Parser* type:

```
type Parser a  =  String → [(a, String)]
```

In summary, this declaration states that a parser of type a is a function that takes an input string and produces a list of results, each of which is a pair comprising a result value of type a and an output string. Alternatively, the parser type can also be read as a rhyme in the style of Dr Seuss!

A parser for things
Is a function from strings
To lists of pairs
Of things and strings

8.3 Basic parsers

We now define three basic parsers that will be used as the building blocks for all other parsers. First of all, the parser *return v* always succeeds with the result value v, without consuming any of the input string:

```
return    ::  a → Parser a
return v  =   λinp → [(v, inp)]
```

This function could equally well be defined by *result v inp* = [(*v*, *inp*)]. However, we prefer the above definition in which the second argument *inp* is shunted to the body of the definition using a lambda expression, because it makes explicit that *return* is a function that takes a single argument and returns a parser, as expressed by the type *a* → *Parser a*.

Whereas *return v* always succeeds, the dual parser *failure* always fails, regardless of the contents of the input string:

```
failure  ::  Parser a
failure  =   λinp → [ ]
```

Our final basic parser is *item*, which fails if the input string is empty, and succeeds with the first character as the result value otherwise:

```
item  ::  Parser Char
item  =   λinp → case inp of
                   [ ] → [ ]
                   (x : xs) → [(x, xs)]
```

The **case** mechanism of Haskell used in this definition allows pattern matching to be used in the body of a definition, in this example by matching the string *inp* against two patterns to choose between two possible results. The **case** mechanism is not used much in this book, but can sometimes be useful.

Because parsers are functions, they could be applied to a string using normal function application, but we prefer to abstract from the representation of parsers by defining our own application function:

```
parse          ::  Parser a → String → [(a, String)]
parse p inp    =   p inp
```

Using *parse*, we conclude this section with some examples that illustrate the behaviour of the three basic parsers defined above:

```
> parse (return 1) "abc"
[(1, "hello")]

> parse failure "abc"
[]

> parse item ""
[]

> parse item "abc"
[('a', "bc")]
```

8.4 | Sequencing

Perhaps the simplest way of combining two parsers is to apply one after the other in sequence, with the output string returned by the first parser becoming the input string to the second. But how should the result values from the two parsers be handled? One approach would be to combine the two values as a pair, using a sequencing operator for parsers with the following type:

```
Parser a → Parser b → Parser (a, b)
```

In practice, however, it turns out to be more convenient to combine the sequencing of parsers with the processing of their result values, by means of a sequencing operator ≫= (read as "then") defined as follows:

```
(≫=)    ::  Parser a → (a → Parser b) → Parser b
p ≫= f  =   λinp → case parse p inp of
                      [] → []
                      [(v, out)] → parse (f v) out
```

That is, the parser $p \gg= f$ fails if the application of the parser p to the input string fails, and otherwise applies the function f to the result value to give a second parser, which is then applied to the output string to give the final result. In this manner, the result value produced by the first parser is made directly available for processing by the second.

A typical parser built using ≫= has the following structure:

```
p1 ≫= λv1 →
p2 ≫= λv2 →
⋮
pn ≫= λvn →
return (f v1 v2 ... vn)
```

That is, apply the parser *p1* and call its result value *v1*; then apply the parser *p2* and call its result value *v2*; . . . ; then apply the parser *pn* and call its result value *vn*; and, finally, combine all the results into a single value by applying the function *f*. Haskell provides a special syntax for such parsers, allowing them to be expressed in the following, more appealing, form:

```
do v1 ← p1
   v2 ← p2
   ⋮
   vn ← pn
   return (f v1 v2 ... vn)
```

As with list comprehensions, the expressions $vi \leftarrow pi$ are called *generators*. If the result value produced by a generator $vi \leftarrow pi$ is not required, the generator can be abbreviated simply by *pi*. Note also that the layout rule applies to the **do** notation for sequencing parsers, in the sense that each parser in the sequence must begin in precisely the same column.

For example, a parser that consumes three characters, discards the second, and returns the first and third as a pair can now be defined as follows:

```
p  ::  Parser (Char, Char)
p  =   do x ← item
          item
          y ← item
          return (x, y)
```

Note that *p* only succeeds if every parser in its defining sequence succeeds, which requires at least three characters in the input string:

```
> parse p "abcdef"
[(('a', 'c'), "def")]

> parse p "ab"
[]
```

8.5 Choice

Another natural way of combining two parsers is to apply the first parser to the input string, and if this fails to apply the second instead. Such a choice operator +++ (read as "or else") can be defined as follows:

```
(+++)    :: Parser a → Parser a → Parser a
p +++ q  =  λinp → case parse p inp of
                     [] → parse q inp
                     [(v, out)] → [(v, out)]
```

For example:

```
> parse (item +++ return 'd') "abc"
[('a', "bc")]

> parse (failure +++ return 'd') "abc"
[('d', "abc")]

> parse (failure +++ failure) "abc"
[]
```

8.6 Derived primitives

Using the three basic parsers together with sequencing and choice, we can now define a number of other useful parsing primitives. First of all, we define a parser *sat p* for single characters that satisfy the predicate *p*:

```
sat    :: (Char → Bool) → Parser Char
sat p  =  do x ← item
             if p x then return x else failure
```

Using *sat* and appropriate predicates from the standard prelude, we can define parsers for single digits, lower-case letters, upper-case letters, arbitrary letters, alphanumeric characters, and specific characters:

```
digit     :: Parser Char
digit     =  sat isDigit

lower     :: Parser Char
lower     =  sat isLower

upper     :: Parser Char
upper     =  sat isUpper

letter    :: Parser Char
letter    =  sat isAlpha

alphanum  :: Parser Char
alphanum  =  sat isAlphaNum

char      :: Char → Parser Char
char x    =  sat (== x)
```

For example:

```
> parse digit "123"
[('1',"23")]

> parse digit "abc"
[]

> parse (char 'a') "abc"
[('a',"bc")]

> parse (char 'a') "123"
[]
```

In turn, using *char* we can define a parser *string xs* for the string of characters *xs*, with the string itself returned as the result value:

```
string          :: String → Parser String
string []       =  return []
string (x : xs) =  do char x
                      string xs
                      return (x : xs)
```

Note that *string* is defined using recursion, and only succeeds if the entire target string is consumed. The base case states that the empty string can always be parsed. The recursive case states that a non-empty string can be parsed by parsing the first character, parsing the remaining characters, and returning the entire string as the result value. For example:

```
> parse (string "abc") "abcdef"
[("abc","def")]

> parse (string "abc") "ab1234"
[]
```

Our next two parsers, *many p* and *many1 p*, apply a parser *p* as many times as possible until it fails, with the result values from each successful application of *p* being combined as a list. The difference between these two repetition primitives is that *many* permits zero or more applications of *p*, whereas *many1* requires at least one successful application:

```
many    :: Parser a → Parser [a]
many p  =  many1 p +++ return []

many1   :: Parser a → Parser [a]
many1 p =  do v ← p
              vs ← many p
              return (v : vs)
```

Note that *many* and *many1* are defined using mutual recursion. In particular, the definition for *many p* states that *p* can either be applied at least once or not at all, while the definition for *many1 p* states that *p* can be applied once and then zero or more times. For example:

```
> parse (many digit) "123abc"
[("123", "abc")]

> parse (many digit) "abcdef"
[("", "abcdef")]

> parse (many1 digit) "abcdef"
[]
```

Using *many* and *many1*, we can define parsers for *identifiers* (variable names) comprising a lower-case letter followed by zero or more alphanumeric characters, natural numbers comprising one or more digits, and spacing comprising zero or more space, tab, and newline characters:

```
ident :: Parser String
ident =  do x ← lower
            xs ← many alphanum
            return (x : xs)

nat   :: Parser Int
nat   =  do xs ← many1 digit
            return (read xs)

space :: Parser ()
space =  do many (sat isSpace)
            return ()
```

For example:

```
> parse ident "abc def"
[("abc", " def")]

> parse nat "123 abc"
[(123, " abc")]

> parse space "    abc"
[((), "abc")]
```

tions of *p*,

n particular,
t once or not
ed once and

fiers (variable
alphanumeric
spacing com-

Note that *space* returns the empty tuple () as a dummy result value, reflecting the fact that the details of spacing are not usually important.

8.7 | Handling spacing

Most real-life parsers allow spacing to be freely used around the basic *tokens* in their input string. For example, the strings 1+2 and 1 + 2 are both parsed in the same way by Hugs. To handle such spacing, we define a new primitive that ignores any space before and after applying a parser for a token:

```
token   :: Parser a → Parser a
token p  = do space
              v ← p
              space
              return v
```

Using *token*, it is now easy to define parsers that ignore spacing around identifiers, natural numbers, and special symbols:

```
identifier :: Parser String
identifier  = token ident

natural    :: Parser Int
natural     = token nat

symbol     :: String → Parser String
symbol xs   = token (string xs)
```

For example, a parser for a non-empty list of natural numbers that ignores spacing around tokens can be defined as follows:

```
p :: Parser [Int]
p  = do symbol "["
        n ← natural
        ns ← many (do symbol ","
                      natural)
        symbol "]"
        return (n : ns)
```

This definition states that such a list begins with an opening square bracket and a natural number, followed by zero or more commas and natural numbers, and concludes with a closing square bracket. Note that *p* only succeeds if a complete list in precisely this format is consumed:

```
> parse p " [1,  2,  3] "
[([1, 2, 3], "")]

> parse p "[1,2,]"
[]
```

8.8 Arithmetic expressions

We conclude this chapter with an extended example. Consider a simple form of arithmetic expressions built up from natural numbers using addition, multiplication, and parentheses. We assume that addition and multiplication associate to the right, and that multiplication has higher priority than addition. For example, $2 + 3 + 4$ means $2 + (3 + 4)$, while $2 * 3 + 4$ means $(2 * 3) + 4$.

The syntactic structure of a language can be formalised using the mathematical notion of a *grammar*, which is a set of rules that describes how strings of the language can be constructed. For example, a grammar for our language of arithmetic expressions can be defined by the following two rules:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{expr} + \mathit{expr} \mid \mathit{expr} * \mathit{expr} \mid (\mathit{expr}) \mid \mathit{nat} \\ \mathit{nat} & ::= & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

The first rule states that an expression is either the addition or multiplication of two expressions, a parenthesised expression, or a natural number. In turn, the second rule states that a natural number is either zero, one, two, etc. For example, using this grammar the construction of the expression $2 * 3 + 4$ can be represented by the following *parse tree*, in which the tokens in the expression appear at the leaves, and the grammatical rules applied to construct the expression give rise to the branching structure:

$$\begin{tikzcd}
 & & \mathit{expr} \arrow[dl] \arrow[d] \arrow[dr] & \\
 & \mathit{expr} \arrow[dl] \arrow[d] \arrow[dr] & + & \mathit{expr} \arrow[d] \\
\mathit{expr} \arrow[d] & * & \mathit{expr} \arrow[d] & \mathit{nat} \arrow[d] \\
\mathit{nat} \arrow[d] & & \mathit{nat} \arrow[d] & 4 \\
2 & & 3 &
\end{tikzcd}$$

The structure of this tree makes explicit that $2 * 3 + 4$ can be constructed from the addition of two expressions, the first given by the multiplication of two further expressions which are in turn given by the numbers two and three, and the second expression given by the number four. However, the grammar also permits another possible parse tree for this example, which corresponds to the erroneous interpretation of the expression as $2 * (3 + 4)$:

$$\begin{tikzcd}
 & \mathit{expr} \arrow[dl] \arrow[d] \arrow[dr] & & \\
\mathit{expr} \arrow[d] & * & \mathit{expr} \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathit{nat} \arrow[d] & \mathit{expr} \arrow[d] & + & \mathit{expr} \arrow[d] \\
2 & \mathit{nat} \arrow[d] & & \mathit{nat} \arrow[d] \\
 & 3 & & 4
\end{tikzcd}$$

The problem is that our grammar for expressions does not take account of the fact that multiplication has higher priority than addition. However, this can easily be fixed by modifying the grammar to have a separate rule for each level of priority, with addition at the lowest level of priority, multiplication at the middle level, and parentheses and numbers at the highest level:

$$\begin{array}{lcl} expr & ::= & expr + expr \mid term \\ term & ::= & term * term \mid factor \\ factor & ::= & (expr) \mid nat \\ nat & ::= & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

Using this new grammar, $2 * 3 + 4$ indeed has a single parse tree, which corresponds to the correct interpretation of the expression as $(2 * 3) + 4$:

$$\begin{tikzcd}
 & & expr \arrow[dl] \arrow[d] \arrow[dr] & \\
 & expr \arrow[d] & + & expr \arrow[d] \\
 & term \arrow[dl] \arrow[d] \arrow[dr] & & term \arrow[d] \\
term \arrow[d] & * & term \arrow[d] & factor \arrow[d] \\
factor \arrow[d] & & factor \arrow[d] & nat \arrow[d] \\
nat \arrow[d] & & nat \arrow[d] & 4 \\
2 & & 3 &
\end{tikzcd}$$

We have now dealt with the issue of priority, but our grammar does not yet take account of the fact that addition and multiplication associate to the right. For example, the expression $2 + 3 + 4$ currently has two possible parse trees, corresponding to $(2 + 3) + 4$ and $2 + (3 + 4)$. However, this can also easily be fixed by modifying the grammatical rules for addition and multiplication to be recursive in their right argument only, rather than both arguments:

$$\begin{array}{lcl} expr & ::= & term + expr \mid term \\ term & ::= & factor * term \mid factor \end{array}$$

Using these new rules, $2 + 3 + 4$ now has a single parse tree, which corresponds to the correct interpretation of the expression as $2 + (3 + 4)$:

$$\begin{tikzcd}
 & expr \arrow[dl] \arrow[d] \arrow[dr] & & \\
term \arrow[d] & + & expr \arrow[dl] \arrow[d] \arrow[dr] & \\
factor \arrow[d] & term \arrow[d] & + & expr \arrow[d] \\
nat \arrow[d] & factor \arrow[d] & & term \arrow[d] \\
2 & nat \arrow[d] & & factor \arrow[d] \\
 & 3 & & nat \arrow[d] \\
 & & & 4
\end{tikzcd}$$

In fact, our grammar for expressions is now unambiguous, in the sense that every well-formed expression has precisely one parse tree.

Our final modification to the grammar is one of simplification. For example, consider the rule $expr ::= term + expr \mid term$, which states that an expression is either the addition of a term and an expression, or is a term. In other words, an expression always begins with a term, which can then be followed by the addition of an expression or by nothing. Hence, the rule for expressions can be simplified to $expr ::= term\ (+\ expr \mid \epsilon)$, in which the symbol ϵ denotes the empty string. Simplifying the rule for terms in a similar manner gives our final grammar for arithmetic expressions:

$$\begin{array}{lcl} expr & ::= & term\ (+\ expr \mid \epsilon) \\ term & ::= & factor\ (*\ term \mid \epsilon) \\ factor & ::= & (expr) \mid nat \\ nat & ::= & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

It is now straightforward to translate this grammar into a parser for expressions, by simply rewriting the rules using our parsing primitives. In fact, we choose to have the parser itself evaluate the expression being parsed to its integer value, rather than returning some form of tree:

```
expr :: Parser Int
expr = do t ← term
          do symbol "+"
             e ← expr
             return (t + e)
           +++ return t

term :: Parser Int
term = do f ← factor
          do symbol "*"
             t ← term
             return (f * t)
           +++ return f

factor :: Parser Int
factor = do symbol "("
            e ← expr
            symbol ")"
            return e
          +++ natural
```

For example, the parser *expr* first parses a term with value t, then parses a plus symbol followed by an expression with value e and returns the value $t + e$, or else parses nothing further and simply returns the value t. The parsers *term* and *factor* can be read in a similar manner.

Finally, using *expr* we define a function $eval :: String \rightarrow Int$ that evaluates an arithmetic expression to its integer value. To handle the cases of unconsumed and invalid input, we use the library function $error :: String \rightarrow a$ that displays an error message and then terminates the program:

```
eval    :: String → Int
eval xs  =  case parse expr xs of
               [(n, [])] → n
               [(_, out)] → error ("unused input " ++ out)
               [] → error "invalid input"
```

For example:

```
> eval "2*3+4"
10

> eval "2*(3+4)"
14

> eval "2 * (3 + 4)"
14

> eval "2*3-4"
Error : unused input - 4

> eval "-1"
Error : invalid input
```

8.9 | Chapter remarks

A library file comprising the parsing primitives from this chapter is available from the book's website. For technical reasons concerning the monadic nature of parsers, a number of the basic definitions in this library are slightly different to those given here. Further details are available in (16; 17), upon which this chapter is based. More information concerning grammars can be found in (27), and more advanced approaches to building parsers in Haskell are given in (22; 9). The reading of the parser type as a rhyme is due to Fritz Ruehr.

8.10 | Exercises

1. The library file also defines a parser *int* :: *Parser Int* for an integer. Without looking at this definition, define *int*. Hint: an integer is either a minus symbol followed by a natural number, or a natural number.
2. Define a parser *comment* :: *Parser* () for ordinary Haskell comments that begin with the symbol -- and extend to the end of the current line, which is represented by the control character `'\n'`.
3. Using our second grammar for arithmetic expressions, draw the two possible parse trees for the expression $2 + 3 + 4$.
4. Using our third grammar for arithmetic expressions, draw the parse trees for the expressions $2 + 3$, $2 * 3 * 4$ and $(2 + 3) + 4$.

5. Explain why the final simplification of the grammar for arithmetic expressions has a dramatic effect on the efficiency of the resulting parser. Hint: begin by considering how an expression comprising a single number would be parsed if this step had not been made.

6. Extend the parser for arithmetic expressions to support subtraction and division, based upon the following extensions to the grammar:

$$\begin{array}{lll} \mathit{expr} & ::= & \mathit{term}\ (+\ \mathit{expr} \mid -\ \mathit{expr} \mid \epsilon) \\ \mathit{term} & ::= & \mathit{factor}\ (*\ \mathit{term} \mid /\ \mathit{term} \mid \epsilon) \end{array}$$

7. Further extend the grammar and parser for arithmetic expressions to support exponentiation, which is assumed to associate to the right and have higher priority than multiplication and division, but lower priority than parentheses and numbers. For example, $2 \uparrow 3 * 4$ means $(2 \uparrow 3) * 4$. Hint: the new level of priority requires a new rule in the grammar.

8. Consider expressions built up from natural numbers using a subtraction operator that is assumed to associate to the left.

   (a) Define a natural grammar for such expressions.
   (b) Translate this grammar into a parser *expr* :: *Parser Int*.
   (c) What is the problem with this parser?
   (d) Show how it can be fixed. Hint: rewrite the parser using the repetition primitive *many* and the library function *foldl*.