Chapter 11

Parsing

A *parser* is a function that analyses a piece of text to determine its logical structure. The text is a string of characters describing some value of interest, such as an arithmetic expression, a poem or a spreadsheet. The output of a parser is a representation of the value, such as a tree of some kind for an arithmetic expression, a list of verses for a poem, or something more complicated for a spreadsheet. Most programming tasks involve decoding the input in some way, so parsing is a pervasive component of computer programming. In this chapter we will describe a monadic approach to parsing, mainly designing simple parsers for expressions of various kinds. We will also say a little more about the converse process of encoding the output as a string; in other words, more about the type class `Show`. This material will be used in the final chapter.

11.1 Parsers as monads

Parsers return different values of interest, so as a first cut we can think of a parser as a function that takes a string and returns a value:

```
type Parser a = String -> a
```

This type is basically the same as that of the standard prelude function

```
read :: Read a => String -> a
```

Indeed, `read` is a parser, though not a very flexible one. One reason is that all the input must be consumed. Thus:

```
ghci> read "123" :: Int
123
```

```
ghci> read "123+51" :: Int
*** Exception: Prelude.read: no parse
```

With `read` there is no obvious way of reading two or more things in sequence. For example, in a parser for arithmetic expressions we may want to look in the input stream for a numeral, then an operator and then another numeral. The first parser for a numeral will consume some prefix of the input, the parser for an operator some prefix of the remaining input, and the third parser yet more input. A better idea is to define a parser as a function that consumes a prefix of the input and returns both a value of interest and the unconsumed suffix:

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

We are not quite there yet. It can happen that a parser may *fail* on some input. It is not a mistake to construct parsers that can fail. For example, in a parser for arithmetic expressions, we may want to look for either a numeral or an opening parenthesis. One or either of these subsidiary parsers will certainly fail. Failure should not be thought of as an error that terminates the parsing process; rather it acts like an identity element for an operation that chooses between alternatives. More generally, a parser may find a number of different ways that some prefix of the input can be structured. Failure then corresponds to the particular case of the empty sequence of parses. In order to handle these various possibilities, we change our definition yet again and define

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

The standard prelude provides exactly this type synonym, except that it is called `ReadS`, not `Parser`. And it also provides a function

```
reads :: Read a => ReadS a
```

as a subsidiary method in the type class `Read`. For example,

```
ghci> reads "-123+51" :: [(Int,String)]
[(-123,"+51")]
ghci> reads "+51" :: [(Int,String)]
[]
```

As with the function `read` you have to tell `reads` the type you are expecting. The second example fails, returning no parses, because a Haskell integer can be preceded by an optional minus sign but not by an optional plus sign. By definition, a parser is *deterministic* if it returns an empty or singleton list of parses in all possible cases. In particular, instances of `reads` ought to be deterministic parsers.

There is one further change we have to make to the definition of `Parser`. We would like to install this type as an instance of the `Monad` class, but that is not possible. The reason is that `Parser` is declared as a type synonym, and type synonyms cannot be made members of any type class: they inherit whatever instances are declared for the underlying type. A type synonym is there simply to improve readability in type declarations; no new types are involved and we cannot construct two different type class instances for what is essentially the same type.

One way to construct a new type is by a data declaration:

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

The identifier `Parser` on the right is a constructor, while on the left it is the name of a new type. Most people are happy with the pun; others would rename the constructor as something like `MkParser` or just `P`.

There is a better way to create a new type for `Parser` and that is to use a `newtype` declaration:

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

We have not needed `newtype` declarations up to now, so let us digress a little to explain them. The price paid for using a `data` declaration for `Parser` is that operations to examine parsers have to be constantly unwrapped and rewrapped with the constructor `Parser`, and this adds to the running time of parser operations. In addition there is an unwanted element of `Parser`, namely `Parser undefined`. In other words, `Parser a` and `String -> [(a,String)]` are not *isomorphic* types. Recognising this, Haskell allows a `newtype` declaration for types defined with a *single* constructor taking a *single* argument. It differs from a type synonym in that it creates a genuinely new type whose values must be expressed using the `Parser` wrapper. But these coercions, though they have to appear in the program text, do not add to the execution time of the program because the Haskell compiler eliminates them before evaluation begins. The values of the new type are systematically replaced by the values in the underlying type. Consequently, `Parser a` and `String -> [(a,String)]` describe isomorphic types, and `Parser undefined` and `undefined` are isomorphic values sharing the same representation. New types, as distinct from synonym types, can be made members of type classes in different ways from the underlying type.

With either kind of declaration we have to provide some way of applying the parsing function, so we define

```
apply :: Parser a -> String -> [(a,String)]
apply (Parser p) s = p s
```

The functions `apply` and `Parser` are mutual inverses and witness the isomorphism.

We also define

```
parse :: Parser a -> String -> a
parse p = fst . head . apply p
```

The function `parse p` returns the first object of the first parse, causing an error if the parser `p` fails. This is the only place an error might occur.

Now we can define

```
instance Monad Parser where
  return x = Parser (\s -> [(x,s)])
  p >>= q  = Parser (\s -> [(y,s'')
                           | (x,s') <- apply p s,
                             (y,s'') <- apply (q x) s'])
```

In the definition of `p >>= q` the parser `p` is applied to an input string, producing a list of possible parses each of which is paired with the corresponding unconsumed portion of the input. The parser `q` is then applied to each parse to produce a list of results whose concatenation provides the final answer. One should also show that the three monad laws hold, a task we will leave as an exercise.

11.2 Basic parsers

Perhaps the simplest basic parser is

```
getc :: Parser Char
getc = Parser f
       where f []     = []
             f (c:cs) = [(c,cs)]
```

This parser returns the first character of the input if there is one. It plays exactly the same role for parsers as `getChar` does for the input–output monad of the previous chapter.

Next, here is a parser for recognising a character that satisfies a given condition:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do {c <- getc;
            if p c then return c
            else fail}
```

where `fail` is defined by

```
fail = Parser (\s -> [])
```

The parser `fail` is another basic parser that returns no parses. The parser `sat p` reads a character and, if it satisfies `p`, returns the character as the result. The definition of `sat` can be written more briefly by using a little combinator called `guard`:

```
sat p = do {c <- getc; guard (p c); return c}

guard :: Parser ()
guard True  = return ()
guard False = fail
```

To see that these two definitions are the same, observe that if `p c` is false, then

```
guard (p c) >> return c = fail >> return c = fail
```

Note the use of the law `fail >> p = fail`, whose proof we leave as an exercise. If `p c` is true, then

```
 guard (p c) >> return c
= return () >> return c
= return c
```

Using *sat* we can define a number of other parsers; for instance

```
char :: Char -> Parser ()
char x = do {c <- sat (==x); return ()}

string :: String -> Parser ()
string []     = return ()
string (x:xs) = do {char x; string xs; return ()}

lower :: Parser Char
lower = sat isLower

digit :: Parser Int
digit = do {d <- sat isDigit; return (cvt d)}
        where cvt d = fromEnum d - fromEnum '0'
```

The parser `char x` looks for the specific character `x` as the next item in the input string, while `string xs` looks for a specific string; both parsers return `()` if successful. For example,

```
ghci> apply (string "hell") "hello"
[((),"o")]
```

The parser `digit` looks for a digit character and returns the corresponding integer if successful. The parser `lower` looks for a lowercase letter, returning such a letter if found.

11.3 Choice and repetition

In order to define more sophisticated parsers we need operations for choosing between alternative parsers and for repeating parsers. One such alternation operator is `(<|>)`, defined by

```
(<|>) :: Parser a -> Parser a -> Parser a
p <|> q = Parser f
          where f s = let ps = apply p s in
                      if null ps then apply q s
                      else ps
```

Thus `p <|> q` returns the same parses as `p` unless `p` fails, in which case the parses of `q` are returned. If both `p` and `q` are deterministic, then so is `p <|> q`. For another choice of `<|>` see the exercises. We claim that `<|>` is associative with `fail` as its identity element, but again we relegate the proof as an exercise.

Here is a parser for recognising a string of lowercase letters:

```
lowers :: Parser String
lowers = do {c <- lower; cs <- lowers; return (c:cs)}
         <|> return ""
```

To see how this parser works, suppose the input is the string 'Upper'. In this case the parser on the left of `<|>` fails because 'U' is not a lowercase letter. However, the parser on the right succeeds, so

```
ghci> apply lowers "Upper"
[("","Upper")]
```

With input string 'isUpper', the left-hand parser succeeds, so

```
ghci> apply lowers "isUpper"
[("is","Upper")]
```

Use of the choice operator <|> requires care. For example, consider a very simple form of arithmetic expression that consists of either a single digit or a digit followed by a plus sign followed by another digit. Here is a possible parser:

```
wrong :: Parser Int
wrong = digit <|> addition

addition :: Parser Int
addition = do {m <- digit; char '+'; n <- digit;
               return (m+n)}
```

We have

```
ghci> apply wrong "1+2"
[(1,"+2")]
```

The parser `digit` succeeds, so `addition` is not invoked. But what we really wanted was to return `[(3,"")]`, absorbing as much of the input as possible. One way to correct `wrong` is to rewrite it in the form

```
better = addition <|> digit
```

Then on `1+2` the parser `addition` succeeds, returning the result we want. What is wrong with `better` is that it is inefficient: applied to the input `1` it parses the digit but fails to find a subsequent plus sign, so parser `addition` fails. As a result `digit` is invoked and the input is parsed again from scratch. Not really a problem with a single digit, but the repetition of effort could be costly if we were parsing for a numeral that could contain many digits.

The best solution is to *factor* the parser for digits out of the two component parsers:

```
best   = digit >>= rest
rest m = do {char '+'; n <- digit; return (m+n)}
         <|> return m
```

The argument to `rest` is just an accumulating parameter. We saw essentially the same solution in the chapter on pretty-printing. Factoring parsers to bring out common prefixes is a Good Idea to improve efficiency.

Generalising from the definition of `lowers`, we can define a parser combinator that repeats a parser zero or more times:

```
many :: Parser a -> Parser [a]
many p = do {x <- p; xs <- many p; return (x:xs)}
         <|> none
```

```
none = return []
```

The value `none` is different from `fail` (why?). We can now define

```
lowers = many lower
```

In many applications, so-called *white space* (sequences of space, newline and tab characters) can appear between *tokens* (identifiers, numbers, opening and closing parentheses, and so on) just to make the text easier to read. The parser `space` recognises white space:

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

The function `isSpace` is defined in the library `Data.Char`. The function

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

ignores white space before recognising a given string. More generally we can define

```
token :: Parser a -> Parser a
token p =  space >> p
```

for ignoring white space before invoking a parser. Note that

```
token p <|> token q = token (p <|> q)
```

but the right-hand parser is more efficient as it does not look for white space twice if the first parser fails.

Sometimes we want to repeat a parser one or more times rather than zero or more times. This can be done by a combinator which we will call `some` (it is also called `many1` in some parser libraries):

```
some :: Parser a -> Parser [a]
some p = do {x <- p; xs <- many p; return (x:xs)}
```

This definition repeats that of the first parser in the definition of `many`, a fact we can take into account by redefining `many` in terms of `some`:

```
many :: Parser a -> Parser [a]
many p = optional (some p)

optional :: Parser [a] -> Parser [a]
optional p = p <|> none
```

The parsers `many` and `some` are now mutually recursive.

Here is a parser for natural numbers, one that allows white space before the number:

```
natural :: Parser Int
natural = token nat
nat = do {ds <- some digit;
          return (foldl1 shiftl ds)}
      where shiftl m n = 10*m+n
```

The subsidiary parser `nat` does not allow white space before the number.

Consider now how to define a parser for an *integer* numeral, which by definition is a nonempty string of digits possibly prefixed by a minus sign. You might think that the parser

```
int :: Parser Int
int = do {symbol "-"; n <- natural; return (-n)}
      <|> natural
```

does the job, but it is inefficient (see Exercise H) and may or may not be what we want. For example,

```
ghci> apply int "  -34"
[(-34,"")]
ghci> apply int "  - 34"
[(-34,"")]
```

Whereas we are quite happy with white space before a numeral, we may not want any white space to appear between the minus sign and the ensuing digits. If that is the case, then the above parser will not do. It is easy to modify the given definition of `int` to give what we want:

```
int :: Parser Int
int = do {symbol "-"; n <- nat; return (-n)}
      <|> natural
```

This parser is still inefficient, and a better alternative is to define

```
int :: Parser Int
int = do {space; f <- minus; n <- nat; return (f n)}
 where
 minus = (char '-' >> return negate) <|> return id
```

The parser `minus` returns a function, either `negate` if the first symbol is a minus sign, or the identity function otherwise.

Next, let us parse a list of integers, separated by commas and enclosed in square brackets. White space is allowed before and after commas and brackets though not of course between the digits of the integers. Here is a very short definition:

```
ints :: Parser [Int]
ints = bracket (manywith (symbol ",") int)
```

The subsidiary parser `bracket` deals with the brackets:

```
bracket :: Parser a -> Parser a
bracket p = do {symbol "[";
                x <- p;
                symbol "]";
                return x}
```

The function `manywith sep p` acts a bit like `many p` but differs in that the instances of `p` are separated by instances of `sep` whose results are ignored. The definition is

```
manywith :: Parser b -> Parser a -> Parser [a]
manywith q p = optional (somewith q p)

somewith :: Parser b -> Parser a -> Parser [a]
somewith q p = do {x <- p;
                   xs <- many (q >> p);
                   return (x:xs)}
```

For example,

```
ghci> apply ints "[2, -3, 4]"
[([2,-3,4],"")]
ghci> apply ints "[2, -3, +4]"
[]
ghci> apply ints "[]"
[([],"")]
```

Integers cannot be preceded by a plus sign, so parsing the second expression fails.

11.4 Grammars and expressions

The combinators described so far are sufficiently powerful for translating a structural description of what is required directly into a functional parser. Such a struc-

tural description is provided by a *grammar*. We will illustrate some typical grammars by looking at parsers for various kinds of arithmetic expression.

Let us start by building a parser for the type `Expr`, defined by

```
data Expr = Con Int | Bin Op Expr Expr
data Op   = Plus | Minus
```

Here is a grammar for fully parenthesised expressions, expressed in what is known as *Backus-Naur form*, or BNF for short:

```
expr  ::= nat | '(' expr op expr ')'
op    ::= '+' | '-'
nat   ::= {digit}+
digit ::= '0' | '1' | ... | '9'
```

This grammar defines four *syntactic categories*. Symbols enclosed in quotes are called *terminal* symbols and describe themselves; these are symbols that actually occur in the text. There are ten possible characters for a digit, and a `nat` is defined as a sequence of one or more digits. The meta-symbol `{-}+` describes a non-zero repetition of a syntactic category. Note that we do not allow an optional minus sign before a sequence of digits, so constants are natural numbers, not arbitrary integers. The grammar states that an expression is either a natural number or else a compound expression consisting of an opening parenthesis, followed by an expression, followed by either a plus or minus sign, followed by another expression, and finally followed by a closing parenthesis. It is implicitly understood in the description that white space is ignored between terminal symbols except between the digits of a number. The grammar translates directly into a parser for expressions:

```
expr :: Parser Expr
expr = token (constant <|> paren binary)
constant = do {n <- nat; return (Con n)}
binary = do {e1 <- expr;
             p <- op;
             e2 <- expr;
             return (Bin p e1 e2)}
op = (symbol "+" >> return Plus) <|>
     (symbol "-" >> return Minus)
```

For readability we have made use of a subsidiary parser `binary`; the parser `paren` is left as an exercise.

Now suppose we want a parser that also works for expressions that are not fully parenthesised, things like `6-2-3` and `6-(2-3)` and `(6-2)-3`. In such a case, `(+)`

and (-) should associate to the left in expressions, as is normal with arithmetic. One way to express such a grammar in BNF is to write

```
expr ::= expr op term | term
term ::= nat | '(' expr ')'
```

This grammar says that an expression is a sequence of one or more terms separated by operators. A term is either a number or a parenthesised expression. In particular, 6-2-3 will be parsed as the expression 6-2 followed by a minus operator, followed by the term 3. In other words, the same as (6-2)-3, as required. This grammar also translates directly into a parser:

```
expr = token (binary <|> term)
binary = do {e1 <- expr;
             p  <- op;
             e2 <- term;
             return (Bin p e1 e2)}
term = token (constant <|> paren expr)
```

However, there is a fatal flaw with this parser: it falls into an infinite loop. After ignoring initial white space the first action of expr is to invoke the parser binary, whose first action is to invoke the parser expr again. Whoops!

Furthermore, it will not do to rewrite expr as

```
expr = token (term <|> binary)
```

because, for example,

```
Main*> apply expr "3+4"
[(Con 3,"+4")]
```

Only the first term is parsed. The problem is called the *left recursion* problem and is a difficulty with all recursive parsers, functional or otherwise.

One solution is to rewrite the grammar in the following equivalent form:

```
expr ::= term {op term}*
```

The meta-symbol {-}* indicates a syntactic category that can be repeated zero or more times. The new parser then takes the form

```
expr = token (term >>= rest)
rest e1 = do {p <- op;
              e2 <- term;
              rest (Bin p e1 e2)} <|> return e1
```

The parser `rest` corresponds to the category `{op term}*` and takes an argument (an accumulating parameter) whose value is the expression parsed so far.

Finally, let us design a parser for arithmetic expressions that may contain multiplication and division, changing the definition of `Op` to

```
data Op = Plus | Minus | Mul | Div
```

The usual rules apply in that multiplication and division take precedence over addition and subtraction, and operations of the same precedence associate to the left. Here is a grammar:

```
expr ::= term {addop term}*
term ::= factor {mulop factor}*
factor ::= nat | '(' expr ')'
addop ::= '+' | '-'
mulop ::= '*' | '/'
```

And here is the parser:

```
expr = token (term >>= rest)
rest e1 = do {p <- addop;
              e2 <- term;
              rest (Bin p e1 e2)}
          <|> return e1
term = token (factor >>= more)
more e1 = do {p <- mulop;
              e2 <- factor;
              more (Bin p e1 e2)}
          <|> return e1
factor = token (constant <|> paren expr)
```

The definitions of `addop` and `mulop` are left as exercises.

11.5 Showing expressions

Our final question is: how can we install `Expr` as a member of the type class `Show` so that the function `show` is the inverse of parsing? More precisely, we want to define `show` so that

```
parse expr (show e) = e
```

Recall that `parse p` extracts the first parse returned by `apply p`.

As a warm-up, here is the instance of `Show` when `expr` is the parser for fully parenthesised expressions involving addition and subtraction only:

```
instance Show Expr where
  show (Con n) = show n
  show (Bin op e1 e2) =
    = "(" ++ show e1 ++
      " " ++ showop op ++
      " " ++ show e2 ++ ")"
showop Plus = "+"
showop Minus = "-"
```

Clear enough, but there is a problem with efficiency. Because `(++)` has time complexity linear in the length of its left argument, the cost of evaluating `show` is, in the worst case, quadratic in the size of the expression.

The solution, yet again, is to use an accumulating parameter. Haskell provides a type synonym `ShowS`:

```
type ShowS = String -> String
```

and also the following subsidiary functions

```
showChar   :: Char -> ShowS
showString :: String -> ShowS
showParen  :: Bool -> ShowS -> ShowS
```

These functions are defined by

```
showChar      = (:)
showString    = (++)
showParen p x = if b then
                  showChar '(' . p . showChar ')'
                  else p
```

Now we can define `show` for expressions by

```
show e = shows e ""
 where
 shows (Con n) = showString (show n)
 shows (Bin op e1 e2)
  = showParen True (shows e1 . showSpace .
    showsop op . showSpace  . shows e2)
showsop Plus  = showChar '+'
showsop Minus = showChar '-'
```

```
showSpace     = showChar ' '
```

This version, which contains no explicit concatenation operations, takes linear time in the size of the expression.

Now suppose we want to display expressions that are not fully parenthesised. There is no need for parentheses around left-hand expressions, but we do need parentheses around right-hand expressions. That leads to

```
show = shows False e ""
 where
 shows b (Con n) = showString (show n)
 shows b (Bin op e1 e2)
  = showParen p (shows False e1 . showSpace .
    showsop op . showSpace  . shows True e2)
```

This definition takes no account of associativity; for example, `1+(2+3)` is not shown as `1+2+3`.

Finally, let's tackle expressions involving all four arithmetic operations. The difference here is that:

1. With expressions `e1 + e2` or `e1 - e2` we will never need parentheses around `e1` (just as above), nor will we need parentheses around `e2` if `e2` is a compound expression with a multiplication or division at the root.
2. On the other hand, with expressions `e1 * e2` or `e1 / e2` we will need parentheses around `e1` if `e1` is a compound expression with a plus or minus at the root, and we will always need parentheses around `e2`.

One way to codify these rules is to introduce precedence levels (for another way, see Exercise L). Define

```
prec :: Op -> Int
prec Mul   = 2
prec Div   = 2
prec Plus  = 1
prec Minus = 1
```

Consider now how to define a function `showsPrec` with type

```
showsPrec :: Int -> Expr -> ShowS
```

such that `showsPrec p e` shows the expression `e` assuming that the parent of `e` is a compound expression with an operator of precedence `p`. We will define `show` by

```
show e = showsPrec 0 e ""
```

so the enclosing *context* of `e` is an operator with fictitious precedence `0`. We can at once define

```
showsPrec p (Con n) = showString (show n)
```

because constants are never enclosed in parentheses. The interesting case is when we have a compound expression. We give the definition first and explain it afterwards:

```
showsPrec p (Bin op e1 e2)
 = showParen (p>q) (showsPrec q e1 . showSpace .
   showsop op . showSpace . showsPrec (q+1) e2)
   where q = prec op
```

We put parentheses around an expression if the parent operator has greater precedence than the current one. To display the expression `e1` it is therefore sufficient to pass the current precedence as the new parent precedence. But we need parentheses around `e2` if the root operator of `e2` has precedence less than *or equal to* `q`; so we have to increment `q` in the second call.

Admittedly, the above definition of `showsPrec` requires a little thought, but there is a payoff. The type class `Show` has a *second* method in it, namely `showsPrec`. Moreover, the default definition of `show` is just the one above. So to install expressions as a member of `Show` we merely have to give the definition of `showsPrec`.

11.6 Exercises

Exercise A

Consider the synonym

```
type Angle = Float
```

Suppose we want to define equality on angles to be equality modulo a multiple of 2π. Why can't we use `(==)` for this test? Now consider

```
newtype Angle = Angle Float
```

Install `Angle` as a member of `Eq`, thereby allowing `(==)` as an equality test between angles.

Exercise B

We could have defined

```
newtype Parser a = Parser (String -> Maybe (a,String))
```

Give the monad instance of this kind of parser.

Exercise C

Prove that `fail >> p = fail`.

Exercise D

Could we have defined `<|>` in the following way?

```
p <|> q = Parser (\s -> parse p s ++ parse q s)
```

When is the result a deterministic parser? Define a function

```
limit :: Parser a -> Parser a
```

such that `limit (p <|> q)` is a deterministic parser, even if `p` and `q` are not.

Exercise E

Parsers are not only instances of monads, they can also be made instances of a more restricted class, called `MonadPlus`, a class we could have introduced in the previous chapter. Basically, these are monads that support choice and failure. The Haskell definition is

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

As examples, both `[]` and `Maybe` can be made members of `MonadPlus`:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` y = y
  Just x  `mplus` y = Just x
```

Install `Parser` as an instance of `MonadPlus`.

Exercise F

Continuing from the previous exercise, the new methods `mzero` and `mplus` are expected to satisfy some equational laws, as is usually the case with the methods of a type class. But currently the precise set of rules that these methods should obey is not agreed on by the Haskell designers! Uncontroversial are the laws that `mplus` should be associative with identity element `mzero`. That's three equations. Another reasonable law is the *left-zero* law

```
mzero >>= f = mzero
```

The corresponding *right-zero* law, namely

```
p >> mzero = mzero
```

can also be imposed. Does the `MonadPlus` instance of the list monad satisfy these five laws? How about the `Maybe` monad?

Finally, the really contentious law is the following one:

```
(p `mplus` q) >>= f = (p >>= f) `mplus` (q >>= f)
```

This law is call the *left-distribution* law. Why can't `Maybe` be installed as a member of `MonadPlus` if the left-distribution is imposed?

Exercise G

Design a parser for recognising Haskell floating-point numbers. Bear in mind that `.314` is not a legitimate number (no digits before the decimal point) and that `3 . 14` is not legitimate either (because no spaces are allowed before or after the decimal point).

Exercise H

Why are the first and second definitions of `int` given in the text inefficient, compared to the third definition?

Exercise I

Is `"(3)"` a fully parenthesised expression? Is it a non-fully parenthesised expression? Haskell allows parenthesised constants:

```
ghci> (3)+4
7
```

Design a parser for fully parenthesised expressions that allows parentheses around constants.

Exercise J

Consider the grammar `expr ::= term {op term}*`. Define `pair` and `shunt` so that the following parser is legitimate:

```
expr = do {e1 <- term;
           pes <- many (pair op term);
           return (foldl shunt e1 pes)}
```

Exercise K

Define the parsers `addop` and `mulop`.

Exercise L

Consider again the showing of expressions with all four arithmetic operations. The rules for putting in parentheses come down to: we need parentheses around `e1` in `e1 op e2` if `op` is a multiplication operator, and the root of `e1` isn't. Dually we will need parentheses around `e2` if either `op` is a multiplication operator or the root of `e2` isn't. Defining

```
isMulOp Mul = True
isMulOp Div = True
isMulOp _   = False
```

construct an alternative definition of `show` involving a subsidiary function

```
showsF :: (Op -> Bool) -> Expr -> ShowS
```

11.7 Answers

Answer to Exercise A

Because `(==)` is the equality test on floating-point numbers, and different numbers cannot be equal.

```
instance Eq Angle where
 Angle x == Angle y = reduce x == reduce y
  where
  reduce x | x<0 = reduce (x + r)
           | x>r = reduce (x - r)
           | otherwise = x
          where r = 2*pi
```

Answer to Exercise B

```
instance Monad Parser where
  return x = Parser (\s -> Just (x,s))
  P >>= q  = Parser (\s -> case apply p s of
                             Nothing -> apply q s
                             Just (x,s') -> Just (x,s'))
```

Answer to Exercise C

```
    fail >> p
  = fail >>= const p
  = fail
```

The fact that `fail >>= p = fail` is immediate from the definition of `fail` and the definition of `p >>= q`.

Answer to Exercise D

Yes, but the result is only a deterministic parser when either `p` or `q` is `fail`. The function `limit` can be defined by

```
limit p = Parser (take 1 . apply p)
```

Answer to Exercise E

```
mzero = fail
mplus = (<|>)
```

Answer to Exercise F

Yes, both the list monad and the `Maybe` monad satisfy the five laws. For example, in the list monad

```
mzero >>= f = concat (map f []) = [] = mzero
xs >> mzero = concat (map (const []) xs) = [] = mzero
```

With `Maybe` the left-distribution law doesn't hold. We have

```
    (Just x `mplus` q) >>= (\x -> Nothing)
  = Just x >>= (\x -> Nothing)
  = Nothing
```

but

```
    (Just x >> \x -> Nothing) `mplus`
    (q >>= \x -> Nothing)
  = Nothing `mplus` (q >>= \x -> Nothing)
  = q >>= \x -> Nothing
```

The two resulting expressions are not equal (take `q = undefined`).

Answer to Exercise G

```
float :: Parser Float
float = do {ds <- some digit;
            char '.';
            fs <- some digit;
            return (foldl shiftl 0 ds +
                    foldr shiftr 0 fs)}
        where shiftl n d = 10*n + fromIntegral d
              shiftr f x = (fromIntegral f+x)/10
```

The parser `digit` returns an `Int`, which has to be converted to a number (in this case a `Float`).

Answer to Exercise H

White space is parsed twice. For example, calling the first version `int1` and the third `int3` we have

```
ghci> apply int3 $ replicate 100000 ' ' ++ "3"
[(3,"")]
(1.40 secs, 216871916 bytes)
ghci> apply int1 $ replicate 100000 ' ' ++ "3"
[(3,"")]
(2.68 secs, 427751932 bytes)
```

Answer to Exercise I

No, according to the first grammar for `expr`, only binary expressions can be parenthesised. Yes, according to the second grammar as arbitrary expressions can be parenthesised.

The revised grammar is

```
expr ::= term | '(' expr op expr ')'
term ::= nat | '(' expr ')'
```

The corresponding parser is

```
expr = token (term <|> paren binary)
 where
 term = token (constant <|> paren expr)
 binary = do {e1 <- expr;
              p <- op;
              e2 <- expr;
              return (Bin p e1 e2)}
```

Answer to Exercise J

```
pair :: Parser a -> Parser b -> Parser (a,b)
pair p q = do {x <- p; y <- q; return (x,y)}

shunt e1 (p,e2) = Bin p e1 e2
```

Answer to Exercise K

```
addop = (symbol "+" >> return Plus) <|>
        (symbol "-" >> return Minus)
mulop = (symbol "*" >> return Mul) <|>
        (symbol "/" >> return Div)
```

Answer to Exercise L

```
show e = showsF (const False) e ""
 where
 showsF f (Con n) = showString (show n)
 showsF f (Bin op e1 e2)
  = showParen (f op) (showsF f1 e1 . showSpace .
    showsop op . showSpace . showsF f2 e2)
    where f1 x = isMulOp op && not (isMulOp x)
          f2 x = isMulOp op || not (isMulOp x)
```

11.8 Chapter notes

The design of functional parsers in a monadic setting has long been a favourite application of functional programming. Our presentation follows that of 'Monadic parsing in Haskell' by Graham Hutton and Erik Meijer, which appears in *The Journal of Functional Programming* 8(4), 437–144, 1998.