J. Functional Programming **1** (1): 1–000, January 1993

Monadic Parser Combinators

Graham Hutton
University of Nottingham
`http://www.cs.nott.ac.uk/Department/Staff/gmh/`

Erik Meijer
University of Utrecht
`http://www.cs.ruu.nl/people/erik/`

Abstract

In functional programming, a popular approach to building recursive descent *parsers* is to model parsers as functions, and to define higher-order functions (or *combinators*) that implement grammar constructions such as sequencing, choice, and repetition. Such parsers form an instance of a *monad*, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems. The purpose of this article is to provide a step-by-step tutorial on the monadic approach to building functional parsers, and to explain some of the benefits that result from exploiting monads. No prior knowledge of parser combinators or of monads is assumed. Indeed, this article can also be viewed as a first introduction to the use of monads in programming.

Contents

1 Introduction 3
2 Combinator parsers 4
- 2.1 The type of parsers 4
- 2.2 Primitive parsers 4
- 2.3 Parser combinators 5

3 Parsers and monads 8
- 3.1 The parser monad 8
- 3.2 Monad comprehension syntax 10

4 Combinators for repetition 12
- 4.1 Simple repetition 13
- 4.2 Repetition with separators 14
- 4.3 Repetition with meaningful separators 15

5 Efficiency of parsers 18
- 5.1 Left factoring 19
- 5.2 Improving lazyness 19
- 5.3 Limiting the number of results 20

6 Handling lexical issues 22
- 6.1 White-space, comments, and keywords 22
- 6.2 A parser for λ-expressions 24

7 Factorising the parser monad 24
- 7.1 The exception monad 25
- 7.2 The non-determinism monad 26
- 7.3 The state-transformer monad 27
- 7.4 The parameterised state-transformer monad 28
- 7.5 The parser monad revisited 29

8 Handling the offside rule 30
- 8.1 The offside rule 30
- 8.2 Modifying the type of parsers 31
- 8.3 The parameterised state-reader monad 32
- 8.4 The new parser combinators 33

9 Acknowledgements 36
10 Appendix: a parser for data definitions 36
References 37

1 Introduction

In functional programming, a popular approach to building recursive descent *parsers* is to model parsers as functions, and to define higher-order functions (or *combinators*) that implement grammar constructions such as sequencing, choice, and repetition. The basic idea dates back to at least Burge's book on recursive programming techniques (Burge, 1975), and has been popularised in functional programming by Wadler (1985), Hutton (1992), Fokker (1995), and others. Combinators provide a quick and easy method of building functional parsers. Moreover, the method has the advantage over functional parser generators such as Ratatosk (Mogensen, 1993) and Happy (Gill & Marlow, 1995) that one has the full power of a functional language available to define new combinators for special applications (Landin, 1966).

It was realised early on (Wadler, 1990) that parsers form an instance of a *monad*, an algebraic structure from mathematics that has proved useful for addressing a number of computational problems (Moggi, 1989; Wadler, 1990; Wadler, 1992a; Wadler, 1992b). As well as being interesting from a mathematical point of view, recognising the monadic nature of parsers also brings practical benefits. For example, using a monadic sequencing combinator for parsers avoids the messy manipulation of nested tuples of results present in earlier work. Moreover, using *monad comprehension* notation makes parsers more compact and easier to read.

Taking the monadic approach further, the monad of parsers can be expressed in a modular way in terms of two simpler monads. The immediate benefit is that the basic parser combinators no longer need to be defined explicitly. Rather, they arise automatically as a special case of lifting monad operations from a base monad m to a certain other monad parameterised over m. This also means that, if we change the nature of parsers by modifying the base monad (for example, limiting parsers to producing at most one result), then new combinators for the modified monad of parsers also arise automatically via the lifting construction.

The purpose of this article is to provide a step-by-step tutorial on the monadic approach to building functional parsers, and to explain some of the benefits that result from exploiting monads. Much of the material is already known. Our contributions are the organisation of the material into a tutorial article; the introduction of new combinators for handling lexical issues without a separate lexer; and a new approach to implementing the offside rule, inspired by the use of monads.

Some prior exposure to functional programming would be helpful in reading this article, but special features of Gofer (Jones, 1995b) — our implementation language — are explained as they are used. Any other lazy functional language that supports (multi-parameter) constructor classes and the use of monad comprehension notation would do equally well. No prior knowledge of parser combinators or monads is assumed. Indeed, this article can also be viewed as a first introduction to the use of monads in programming. A library of monadic parser combinators taken from this article is available from the authors, via the World-Wide-Web.

2 Combinator parsers

We begin by reviewing the basic ideas of combinator parsing (Wadler, 1985; Hutton, 1992; Fokker, 1995). In particular, we define a type for parsers, three primitive parsers, and two primitive combinators for building larger parsers.

2.1 The type of parsers

Let us start by thinking of a parser as a function that takes a string of characters as input and yields some kind of tree as result, with the intention that the tree makes explicit the grammatical structure of the string:

```
type Parser = String -> Tree
```

In general, however, a parser might not consume all of its input string, so rather than the result of a parser being just a tree, we also return the unconsumed suffix of the input string. Thus we modify our type of parsers as follows:

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

Similarly, a parser might fail on its input string. Rather than just reporting a run-time error if this happens, we choose to have parsers return a list of pairs rather than a single pair, with the convention that the empty list denotes failure of a parser, and a singleton list denotes success:

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

Having an explicit representation of failure and returning the unconsumed part of the input string makes it possible to define combinators for building up parsers piecewise from smaller parsers. Returning a list of results opens up the possibility of returning more than one result if the input string can be parsed in more than one way, which may be the case if the underlying grammar is ambiguous.

Finally, different parsers will likely return different kinds of trees, so it is useful to abstract on the specific type `Tree` of trees, and make the type of result values into a parameter of the `Parser` type:

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

This is the type of parsers we will use in the remainder of this article. One could go further (as in (Hutton, 1992), for example) and abstract upon the type `String` of tokens, but we do not have need for this generalisation here.

2.2 Primitive parsers

The three primitive parsers defined in this section are the building blocks of combinator parsing. The first parser is `result v`, which succeeds without consuming any of the input string, and returns the single result `v`:

```
result   :: a -> Parser a
result v = \inp -> [(v,inp)]
```

An expression of the form `\x -> e` is called a λ-abstraction, and denotes the function that takes an argument `x` and returns the value of the expression `e`. Thus `result v` is the function that takes an input string `inp` and returns the singleton list `[(v,inp)]`. This function could equally well be defined by `result v inp = [(v,inp)]`, but we prefer the above definition (in which the argument `inp` is shunted to the body of the definition) because it corresponds more closely to the type `result :: a -> Parser a`, which asserts that `result` is a function that takes a single argument and returns a parser.

Dually, the parser `zero` always fails, regardless of the input string:

```
zero :: Parser a
zero  = \inp -> []
```

Our final primitive is `item`, which successfully consumes the first character if the input string is non-empty, and fails otherwise:

```
item :: Parser Char
item  = \inp -> case inp of
                   []     -> []
                   (x:xs) -> [(x,xs)]
```

2.3 Parser combinators

The primitive parsers defined above are not very useful in themselves. In this section we consider how they can be glued together to form more useful parsers. We take our lead from the BNF notation for specifying grammars, in which larger grammars are built up piecewise from smaller grammars using a *sequencing* operator — denoted by juxtaposition — and a *choice* operator — denoted by a vertical bar |. We define corresponding operators for combining parsers, such that the structure of our parsers closely follows the structure of the underlying grammars.

In earlier (non-monadic) accounts of combinator parsing (Wadler, 1985; Hutton, 1992; Fokker, 1995), sequencing of parsers was usually captured by a combinator

```
seq      :: Parser a -> Parser b -> Parser (a,b)
p `seq` q = \inp -> [((v,w),inp'') | (v,inp')  <- p inp
                                   , (w,inp'') <- q inp']
```

that applies one parser after another, with the results from the two parsers being combined as pairs. The infix notation ``p `seq` q`` is syntactic sugar for `seq p q`; any function of two arguments can used as an infix operator in this way, by enclosing its name in backquotes. At first sight, the `seq` combinator might seem a natural composition primitive. In practice, however, using `seq` leads to parsers with nested tuples as results, which are messy to manipulate.

The problem of nested tuples can be avoided by adopting a *monadic* sequencing combinator (commonly known as `bind`) which integrates the sequencing of parsers with the processing of their result values:

```
bind      :: Parser a -> (a -> Parser b) -> Parser b
p `bind` f = \inp -> concat [f v inp' | (v,inp') <- p inp]
```

The definition for `bind` can be interpreted as follows. First of all, the parser `p` is applied to the input string, yielding a list of (value,string) pairs. Now since `f` is a function that takes a value and returns a parser, it can be applied to each value (and unconsumed input string) in turn. This results in a list of lists of (value,string) pairs, that can then be flattened to a single list using `concat`.

The `bind` combinator avoids the problem of nested tuples of results because the results of the first parser are made directly available for processing by the second, rather than being paired up with the other results to be processed later on. A typical parser built using `bind` has the following structure

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

and can be read operationally as follows: apply parser `p1` and call its result value `x1`; then apply parser `p2` and call its result value `x2`; ...; then apply the parser `pn` and call its result value `xn`; and finally, combine all the results into a single value by applying the function `f`. For example, the `seq` combinator can be defined by

```
p `seq` q = p `bind` \x ->
            q `bind` \y ->
            result (x,y)
```

(On the other hand, `bind` cannot be defined in terms of `seq`.)

Using the `bind` combinator, we are now able to define some simple but useful parsers. Recall that the `item` parser consumes a single character unconditionally. In practice, we are normally only interested in consuming certain specific characters. For this reason, we use `item` to define a combinator `sat` that takes a predicate (a Boolean valued function), and yields a parser that consumes a single character if it satisfies the predicate, and fails otherwise:

```
sat   :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero
```

Note that if `item` fails (that is, if the input string is empty), then so does `sat p`, since it can readily be observed that ``zero `bind` f = zero`` for all functions `f` of the appropriate type. Indeed, this equation is not specific to parsers: it holds for an arbitrary *monad with a zero* (Wadler, 1992a; Wadler, 1992b). Monads and their connection to parsers will be discussed in the next section.

Using `sat`, we can define parsers for specific characters, single digits, lower-case letters, and upper-case letters:

```
char   :: Char -> Parser Char
char x = sat (\y -> x == y)
```

```
digit :: Parser Char
digit  = sat (\x -> '0' <= x && x <= '9')

lower :: Parser Char
lower  = sat (\x -> 'a' <= x && x <= 'z')

upper :: Parser Char
upper  = sat (\x -> 'A' <= x && x <= 'Z')
```

For example, applying the parser `upper` to the input string `"Hello"` succeeds with the single successful result `[('H',"ello")]`, since the `upper` parser succeeds with `'H'` as the result value and `"ello"` as the unconsumed suffix of the input. On the other hand, applying the parser `lower` to the string `"Hello"` fails with `[]` as the result, since `'H'` is not a lower-case letter.

As another example of using `bind`, consider the parser that accepts two lower-case letters in sequence, returning a string of length two:

```
lower `bind` \x ->
lower `bind` \y ->
result [x,y]
```

Applying this parser to the string `"abcd"` succeeds with the result `[("ab","cd")]`. Applying the same parser to `"aBcd"` fails with the result `[]`, because even though the initial letter `'a'` can be consumed by the first `lower` parser, the following letter `'B'` cannot be consumed by the second `lower` parser.

Of course, the above parser for two letters in sequence can be generalised to a parser for arbitrary strings of lower-case letters. Since the length of the string to be parsed cannot be predicted in advance, such a parser will naturally be defined recursively, using a choice operator to decide between parsing a single letter and recursing, or parsing nothing further and terminating. A suitable choice combinator for parsers, `plus`, is defined as follows:

```
plus        :: Parser a -> Parser a -> Parser a
p `plus` q  = \inp -> (p inp ++ q inp)
```

That is, both argument parsers `p` and `q` are applied to the same input string, and their result lists are concatenated to form a single result list. Note that it is not required that `p` and `q` accept disjoint sets of strings: if both parsers succeed on the input string then more than one result value will be returned, reflecting the different ways that the input string can be parsed.

As examples of using `plus`, some of our earlier parsers can now be combined to give parsers for letters and alpha-numeric characters:

```
letter   :: Parser Char
letter    = lower `plus` upper

alphanum :: Parser Char
alphanum  = letter `plus` digit
```

More interestingly, a parser for words (strings of letters) is defined by

```
word :: Parser String
word  = neWord `plus` result ""
        where
           neWord = letter `bind` \x  ->
                    word   `bind` \xs ->
                    result (x:xs)
```

That is, `word` either parses a non-empty word (a single letter followed by a word, using a recursive call to `word`), in which case the two results are combined to form a string, or parses nothing and returns the empty string.

For example, applying `word` to the input `"Yes!"` gives the result `[("Yes","!"), ("Ye","s!"), ("Y","es!"), ("","Yes!")]`. The first result, `("Yes","!")`, is the expected result: the string of letters `"Yes"` has been consumed, and the unconsumed input is `"!"`. In the subsequent results a decreasing number of letters are consumed. This behaviour arises because the choice operator `plus` is *non-deterministic*: both alternatives can be explored, even if the first alternative is successful. Thus, at each application of `letter`, there is always the option to just finish parsing, even if there are still letters left to be consumed from the start of the input.

3 Parsers and monads

Later on we will define a number of useful parser combinators in terms of the primitive parsers and combinators just defined. But first we turn our attention to the monadic nature of combinator parsers.

3.1 The parser monad

So far, we have defined (among others) the following two operations on parsers:

```
result :: a -> Parser a
bind   :: Parser a -> (a -> Parser b) -> Parser b
```

Generalising from the specific case of `Parser` to some arbitrary type constructor `M` gives the notion of a monad: a *monad* is a type constructor `M` (a function from types to types), together with operations `result` and `bind` of the following types:

```
result :: a -> M a
bind   :: M a -> (a -> M b) -> M b
```

Thus, parsers form a monad for which `M` is the `Parser` type constructor, and `result` and `bind` are defined as previously. Technically, the two operations of a monad must also satisfy a few algebraic properties, but we do not concern ourselves with such properties here; see (Wadler, 1992a; Wadler, 1992b) for more details.

Readers familiar with the categorical definition of a monad may have expected two operations `map :: (a -> b) -> (M a -> M b)` and `join :: M (M a) -> M a` in place of the single operation `bind`. However, our definition is equivalent to the categorical one (Wadler, 1992a; Wadler, 1992b), and has the advantage that `bind` generally proves more convenient for monadic programming than `map` and `join`.

Parsers are not the only example of a monad. Indeed, we will see later on how the parser monad can be re-formulated in terms of two simpler monads. This raises the question of what to do about the naming of the monadic combinators `result` and `bind`. In functional languages based upon the Hindley-Milner typing system (for example, Miranda† and Standard ML) it is not possible to use the same names for the combinators of different monads. Rather, one would have to use different names, such as `resultM` and `bindM`, for the combinators of each monad `M`.

Gofer, however, extends the Hindley-Milner typing system with an overloading mechanism that permits the use of the same names for the combinators of different monads. Under this overloading mechanism, the appropriate monad for each use of a name is calculated automatically during type inference.

Overloading in Gofer is accomplished by the use of *classes* (Jones, 1995c). A class for monads can be declared in Gofer by:

```
class Monad m where
   result :: a -> m a
   bind   :: m a -> (a -> m b) -> m b
```

This declaration can be read as follows: a type constructor `m` is a member of the class `Monad` if it is equipped with `result` and `bind` operations of the specified types. The fact that `m` must be a type constructor (rather than just a type) is inferred from its use in the types for the operations.

Now the type constructor `Parser` can be made into an instance of the class `Monad` using the `result` and `bind` from the previous section:

```
instance Monad Parser where
   -- result :: a -> Parser a
   result v   = \inp -> [(v,inp)]

   -- bind   :: Parser a -> (a -> Parser b) -> Parser b
   p `bind` f = \inp -> concat [f v out | (v,out) <- p inp]
```

We pause briefly here to address a couple of technical points concerning Gofer. First of all, type synonyms such as `Parser` must be supplied with all their arguments. Hence the instance declaration above is not actually valid Gofer code, since `Parser` is used in the first line without an argument. The problem is easy to solve (redefine `Parser` using `data` rather than `type`, or as a *restricted* type synonym), but for simplicity we prefer in this article just to assume that type synonyms *can* be partially applied. The second point is that the syntax of Gofer does not currently allow the types of the defined functions in instance declarations to be explicitly specified. But for clarity, as above, we include such types in comments.

Let us turn now to the following operations on parsers:

† Miranda is a trademark of Research Software Ltd.

```
zero :: Parser a
plus :: Parser a -> Parser a -> Parser a
```

Generalising once again from the specific case of the `Parser` type constructor, we arrive at the notion of a *monad with a zero and a plus*, which can be encapsulated using the Gofer class system in the following manner:

```
class Monad m => Monad0Plus m where
   zero :: m a
   (++) :: m a -> m a -> m a
```

That is, a type constructor `m` is a member of the class `Monad0Plus` if it is a member of the class `Monad` (that is, it is equipped with a `result` and `bind`), and if it is also equipped with `zero` and `(++)` operators of the specified types. Of course, the two extra operations must also satisfy some algebraic properties; these are discussed in (Wadler, 1992a; Wadler, 1992b). Note also that `(++)` is used above rather than `plus`, following the example of lists: we will see later on that lists form a monad for which the plus operation is just the familiar append operation `(++)`.

Now since `Parser` is already a monad, it can be made into a monad with a zero and a plus using the following definitions:

```
instance Monad0Plus Parser where
   -- zero :: Parser a
   zero     = \inp -> []

   -- (++) :: Parser a -> Parser a -> Parser a
   p ++ q   = \inp -> (p inp ++ q inp)
```

3.2 Monad comprehension syntax

So far we have seen one advantage of recognising the monadic nature of parsers: the monadic sequencing combinator `bind` handles result values better than the conventional sequencing combinator `seq`. In this section we consider another advantage of the monadic approach, namely that *monad comprehension* syntax can be used to make parsers more compact and easier to read.

As mentioned earlier, many parsers will have a structure as a sequence of `bind`s followed by single call to `result`:

```
p1 `bind` \x1 ->
p2 `bind` \x2 ->
...
pn `bind` \xn ->
result (f x1 x2 ... xn)
```

Gofer provides a special notation for defining parsers of this shape, allowing them to be expressed in the following, more appealing form:

```
[ f x1 x2 ... xn | x1 <- p1
```

```
                    , x2 <- p2
                    , ...
                    , xn <- pn ]
```

In fact, this notation is not specific to parsers, but can be used with any monad (Jones, 1995c). The reader might notice the similarity to the list comprehension notation supported by many functional languages. It was Wadler (1990) who first observed that the comprehension notation is not particular to lists, but makes sense for an arbitrary monad. Indeed, the algebraic properties required of the monad operations turn out to be precisely those required for the notation to make sense. To our knowledge, Gofer is the first language to implement Wadler's *monad comprehension* notation. Using this notation can make parsers much easier to read, and we will use the notation in the remainder of this article.

As our first example of using comprehension notation, we define a parser for recognising specific strings, with the string itself returned as the result:

```
string        :: String -> Parser String
string ""      = [""]
string (x:xs) = [x:xs | _ <- char x, _ <- string xs]
```

That is, if the string to be parsed is empty we just return the empty string as the result; `[""]` is just monad comprehension syntax for `result ""`. Otherwise, we parse the first character of the string using `char`, and then parse the remaining characters using a recursive call to `string`. Without the aid of comprehension notation, the above definition would read as follows:

```
string        :: String -> Parser String
string ""      = result ""
string (x:xs) = char x    `bind` \_ ->
                string xs `bind` \_ ->
                result (x:xs)
```

Note that the parser `string xs` fails if only a suffix of the given string `xs` is recognised in the input. For example, applying the parser `string "hello"` to the input `"hello there"` gives the successful result `[("hello"," there")]`. On the other hand, applying the same parser to `"helicopter"` fails with the result `[]`, even though the suffix `"hel"` of the input can be recognised.

In list comprehension notation, we are not just restricted to *generators* that bind variables to values, but can also use Boolean-valued *guards* that restrict the values of the bound variables. For example, a function `negs` that selects all the negative numbers from a list of integers can be expressed as follows:

```
negs    :: [Int] -> [Int]
negs xs = [x | x <- xs, x < 0]
```

In this case, the expression `x < 0` is a guard that restricts the variable `x` (bound by the generator `x <- xs`) to only take on values less than zero.

Wadler (1990) observed that the use of guards makes sense for an arbitrary monad with a zero. The monad comprehension notation in Gofer supports this use of guards. For example, the `sat` combinator

```
sat  :: (Char -> Bool) -> Parser Char
sat p = item `bind` \x ->
        if p x then result x else zero
```

can be defined more succinctly using a comprehension with a guard:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = [x | x <- item, p x]
```

We conclude this section by noting that there is another notation that can be used to make monadic programs easier to read: the so-called "`do`" notation (Jones, 1994; Jones & Launchbury, 1994). For example, using this notation the combinators `string` and `sat` can be defined as follows:

```
string        :: String -> Parser String
string ""      = do { result "" }
string (x:xs) = do { char x ; string xs ; result (x:xs) }

sat           :: (Char -> Bool) -> Parser Char
sat p          = do { x <- item ; if (p x) ; result x }
```

The `do` notation has a couple of advantages over monad comprehension notation: we are not restricted to monad expressions that end with a use of `result`; and generators of the form `_ <- e` that do not bind variables can be abbreviated by `e`. The `do` notation is supported by Gofer, but monad expressions involving parsers typically end with a use of `result` (to compute the result value from the parser), so the extra generality is not really necessary in this case. For this reason, and for simplicity, in this article we only use the comprehension notation. It would be an easy task, however, to translate our definitions into the `do` notation.

4 Combinators for repetition

Parser generators such as Lex and Yacc (Aho *et al.*, 1986) for producing parsers written in C, and Ratatosk (Mogensen, 1993) and Happy (Gill & Marlow, 1995) for producing parsers written in Haskell, typically offer a fixed set of combinators for describing grammars. In contrast, with the method of building parsers as presented in this article the set of combinators is completely extensible: parsers are first-class values, and we have the full power of a functional language at our disposal to define special combinators for special applications.

In this section we define combinators for a number of common patterns of *repetition*. These combinators are not specific to parsers, but can be used with an arbitrary monad with a zero and plus. For clarity, however, we specialise the types of the combinators to the case of parsers.

In subsequent sections we will introduce combinators for other purposes, including handling lexical issues and Gofer's offside rule.

4.1 Simple repetition

Earlier we defined a parser `word` for consuming zero or more letters from the input string. Using monad comprehension notation, the definition is:

```
word :: Parser String
word  = [x:xs | x <- letter, xs <- word] ++ [""]
```

We can easily imagine a number of other parsers that exhibit a similar structure to `word`. For example, parsers for strings of digits or strings of spaces could be defined in precisely the same way, the only difference being be that the component parser `letter` would be replaced by either `digit` or `char ' '`. To avoid defining a number of different parsers with a similar structure, we abstract on the pattern of recursion in `word` and define a general combinator, `many`, that parses sequences of items.

The combinator `many` applies a parser `p` zero or more times to an input string. The results from each application of `p` are returned in a list:

```
many  :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]
```

Different parsers can be made by supplying different arguments parsers `p`. For example, `word` can be defined just as `many letter`, and the other parsers mentioned above by `many digit` and `many (char ' ')`.

Just as the original `word` parser returns many results in general (decreasing in the number of letters consumed from the input), so does `many p`. Of course, in most cases we will only be interested in the first parse from `many p`, in which `p` is successfully applied as many times as possible. We will return to this point in the next section, when we address the efficiency of parsers.

As another application of `many`, we can define a parser for identifiers. For simplicity, we regard an identifier as a lower-case letter followed by zero or more alphanumeric characters. It would be easy to extend the definition to handle extra characters, such as underlines or backquotes.

```
ident :: Parser String
ident  = [x:xs | x <- lower, xs <- many alphanum]
```

Sometimes we will only be interested in non-empty sequences of items. For this reason we define a special combinator, `many1`, in terms of `many`:

```
many1  :: Parser a -> Parser [a]
many1 p = [x:xs | x <- p, xs <- many p]
```

For example, applying `many1 (char 'a')` to the input `"aaab"` gives the result `[("aaa","b"), ("aa","ab"), ("a","aab")]`, which is the same as for `many (char 'a')`, except that the final pair `("", "aaab")` is no longer present. Note also that `many1 p` may fail, whereas `many p` can always succeed.

Using `many1` we can define a parser for natural numbers:

```
nat :: Parser Int
nat  = [eval xs | xs <- many1 digit]
```

```
         where
            eval xs  = foldl1 op [ord x - ord '0' | x <- xs]
            m `op` n = 10*m + n
```

In turn, `nat` can be used to define a parser for integers:

```
int :: Parser Int
int  = [-n | _ <- char '-', n <- nat] ++ nat
```

A more sophisticated way to define `int` is as follows. First try and parse the negation character '-'. If this is successful then return the negation function as the result of the parse; otherwise return the identity function. The final step is then to parse a natural number, and use the function returned by attempting to parse the '-' character to modify the resulting number:

```
int :: Parser Int
int  = [f n | f <- op, n <- nat]
       where
          op = [negate | _ <- char '-'] ++ [id]
```

4.2 Repetition with separators

The `many` combinators parse sequences of items. Now we consider a slightly more general pattern of repetition, in which separators between the items are involved. Consider the problem of parsing a non-empty list of integers, such as `[1,-42,17]`. Such a parser can be defined in terms of the `many` combinator as follows:

```
ints :: Parser [Int]
ints  = [n:ns | _  <- char '['
              , n  <- int
              , ns <- many [x | _ <- char ',', x <- int]
              , _  <- char ']']
```

As was the case in the previous section for the `word` parser, we can imagine a number of other parsers with a similar structure to `ints`, so it is useful to abstract on the pattern of repetition and define a general purpose combinator, which we call `sepby1`. The combinator `sepby1` is like `many1` in that it recognises non-empty sequences of a given parser `p`, but different in that the instances of `p` are separated by a parser `sep` whose result values are ignored:

```
sepby1          :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = [x:xs | x  <- p
                       , xs <- many [y | _ <- sep, y <- p]]
```

Note that the fact that the results of the `sep` parser are ignored is reflected in the type of the `sepby1` combinator: the `sep` parser gives results of type `b`, but this type does not occur in the type `[a]` of the results of the combinator.

Now `ints` can be defined in a more compact form:

```
ints = [ns | _  <- char '['
           , ns <- int `sepby1` char ','
           , _  <- char ']']
```

In fact we can go a little further. The bracketing of parsers by other parsers whose results are ignored — in the case above, the bracketing parsers are `char '['` and `char ']'` — is common enough to also merit its own combinator:

```
bracket :: Parser a -> Parser b -> Parser c -> Parser b
bracket open p close = [x | _ <- open, x <- p, _ <- close]
```

Now `ints` can be defined just as

```
ints = bracket (char '[')
               (int `sepby1` char ',')
               (char ']')
```

Finally, while `many1` was defined in terms of `many`, the combinator `sepby` (for possibly-empty sequences) is naturally defined in terms of `sepby1`:

```
sepby          :: Parser a -> Parser b -> Parser [a]
p `sepby` sep  = (p `sepby1` sep) ++ [[]]
```

4.3 Repetition with meaningful separators

The `sepby` combinators handle the case of parsing sequences of items separated by text that can be ignored. In this final section on repetition, we address the more general case in which the separators themselves carry meaning. The combinators defined in this section are due to Fokker (1995).

Consider the problem of parsing simple arithmetic expressions such as `1+2-(3+4)`, built up from natural numbers using addition, subtraction, and parentheses. The two arithmetic operators are assumed to associate to the left (thus, for example, `1-2-3` should be parsed as `(1-2)-3`), and have the same precedence. The standard BNF grammar for such expressions is written as follows:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{expr}\ \mathit{addop}\ \mathit{factor} \mid \mathit{factor} \\
\mathit{addop} & ::= & \texttt{+} \mid \texttt{-} \\
\mathit{factor} & ::= & \mathit{nat} \mid \texttt{(}\ \mathit{expr}\ \texttt{)}
\end{array}
$$

This grammar can be translated directly into a combinator parser:

```
expr   :: Parser Int
addop  :: Parser (Int -> Int -> Int)
factor :: Parser Int

expr   = [f x y | x <- expr, f <- addop, y <- factor] ++ factor

addop  = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

In fact, rather than just returning some kind of parse tree, the `expr` parser above actually evaluates arithmetic expressions to their integer value: the `addop` parser returns a function as its result value, which is used to combine the result values produced by parsing the arguments to the operator.

Of course, however, there is a problem with the `expr` parser as defined above. The fact that the operators associate to the left is taken account of by `expr` being *left-recursive* (the first thing it does is make a recursive call to itself). Thus `expr` never makes any progress, and hence does not terminate.

As is well-known, this kind of non-termination for parsers can be solved by replacing left-recursion by iteration. Looking at the *expr* grammar, we see that an expression is a sequence of *factor*s, separated by *addop*s. Thus the parser for expressions can be re-defined using `many` as follows:

```
expr = [... | x   <- factor
            , fys <- many [(f,y) | f <- addop, y <- factor]]
```

This takes care of the non-termination, but it still remains to fill in the "`...`" part of the new definition, which computes the value of an expression.

Suppose now that the input string is `"1-2+3-4"`. Then after parsing using `expr`, the variable `x` will be `1` and `fys` will be the list `[((-),2), ((+),3), ((-),4)]`. These can be reduced to a single value `1-2+3-4 = ((1-2)+3)-4 = -2` by folding: the built-in function `foldl` is such that, for example, `foldl g a [b,c,d,e] = ((a `g` b) `g` c) `g` d) `g` e`. In the present case, we need to take `g` as the function `\x (f,y) -> f x y`, and `a` as the integer `x`:

```
expr = [foldl (\x (f,y) -> f x y) x fys
           | x   <- factor
           , fys <- many [(f,y) | f <- addop, y <- factor]]
```

Now, for example, applying `expr` to the input string `"1+2-(3+4)"` gives the result `[(-4,""), (3,"-(3+4)", (1,"+2-(3+4)")]`, as expected.

Playing the generalisation game once again, we can abstract on the pattern of repetition in `expr` and define a new combinator. The combinator, `chainl1`, parses non-empty sequences of items separated by operators that associate to the left:

```
chainl1       :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = [foldl (\x (f,y) -> f x y) x fys
                    | x   <- p
                    , fys <- many [(f,y) | f <- op, y <- p]]
```

Thus our parser for expressions can now be written as follows:

```
expr   = factor `chainl1` addop

addop  = [(+) | _ <- char '+'] ++ [(-) | _ <- char '-']

factor = nat ++ bracket (char '(') expr (char ')')
```

Most operator parsers will have a similar structure to `addop` above, so it is useful to abstract a combinator for building such parsers:

```
ops    :: [(Parser a, b)] -> Parser b
ops xs = foldr1 (++) [[op | _ <- p] | (p,op) <- xs]
```

The built-in function `foldr1` is such that, for example, `foldr1 g [a,b,c,d] = a 'g' (b 'g' (c 'g' d))`. It is defined for any non-empty list. In the above case then, `foldr1` places the choice operator `(++)` between each parser in the list. Using `ops`, our `addop` parser can now be defined by

```
addop = ops [(char '+', (+)), (char '-', (-))]
```

A possible inefficiency in the definition of the `chainl1` combinator is the construction of the intermediate list `fys`. This can be avoided by giving a direct recursive definition of `chainl1` that does not make use of `foldl` and `many`, using an accumulating parameter to construct the final result:

```
chainl1         :: Parser a -> Parser (a -> a -> a) -> Parser a
p 'chainl1' op = p 'bind' rest
                 where
                    rest x = (op 'bind' \f ->
                              p  'bind' \y ->
                              rest (f x y)) ++ [x]
```

This definition has a natural operational reading. The parser `p 'chainl1' op` first parses a single `p`, whose result value becomes the initial accumulator for the `rest` function. Then it attempts to parse an operator and a single `p`. If successful, the accumulator and the result from `p` are combined using the function `f` returned from parsing the operator, and the resulting value becomes the new accumulator when parsing the remainder of the sequence (using a recursive call to `rest`). Otherwise, the sequence is finished, and the accumulator is returned.

As another interesting application of `chainl1`, we can redefine our earlier parser `nat` for natural numbers such that it does not construct an intermediate list of digits. In this case, the `op` parser does not do any parsing, but returns the function that combines a natural and a digit:

```
nat :: Parser Int
nat  = [ord x - ord '0' | x <- digit] 'chainl1' [op]
       where
          m 'op' n = 10*m + n
```

Naturally, we can also define a combinator `chainr1` that parses non-empty sequences of items separated by operators that associate to the *right*, rather than to the left. For simplicity, we only give the direct recursive definition:

```
chainr1         :: Parser a -> Parser (a -> a -> a) -> Parser a
p 'chainr1' op =
   p 'bind' \x ->
      [f x y | f <- op, y <- p 'chainr1' op] ++ [x]
```

That is, `p 'chainr1' op` first parses a single `p`. Then it attempts to parse an operator and the rest of the sequence (using a recursive call to `chainr1`). If successful, the pair of results from the first p and the rest of the sequence are combined using the function f returned from parsing the operator. Otherwise, the sequence is finished, and the result from p is returned.

As an example of using chainr1, we extend our parser for arithmetic expressions to handle exponentiation; this operator has higher precedence than the previous two operators, and associates to the right:

```
expr   = term   `chainl1` addop

term   = factor `chainr1` expop

factor = nat ++ bracket (char '(') expr (char ')')

addop  = ops [(char '+', (+)), (char '-', (-))]

expop  = ops [(char '^', (^))]
```

For completeness, we also define combinators chainl and chainr that have the same behaviour as chainl1 and chainr1, except that they can also consume no input, in which case a given value v is returned as the result:

```
chainl :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op v = (p `chainl1` op) ++ [v]

chainr :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainr p op v = (p `chainr1` op) ++ [v]
```

In summary then, chainl and chainr provide a simple way to build parsers for expression-like grammars. Using these combinators avoids the need for transformations to remove left-recursion in the grammar, that would otherwise result in non-termination of the parser. They also avoid the need for left-factorisation of the grammar, that would otherwise result in unnecessary backtracking; we will return to this point in the next section.

5 Efficiency of parsers

Using combinators is a simple and flexible method of building parsers. However, the power of the combinators — in particular, their ability to backtrack and return multiple results — can lead to parsers with unexpected space and time performance if one does not take care. In this section we outline some simple techniques that can be used to improve the efficiency of parsers. Readers interested in further techniques are referred to Röjemo's thesis (1995), which contains a chapter on the use of heap profiling tools in the optimisation of parser combinators.

5.1 Left factoring

Consider the simple problem of parsing and evaluating two natural numbers separated by the addition symbol '`+`', or by the subtraction symbol '`-`'. This specification can be translated directly into the following parser:

```
eval :: Parser Int
eval  = add ++ sub
        where
           add = [x+y | x <- nat, _ <- char '+', y <- nat]
           sub = [x-y | x <- nat, _ <- char '-', y <- nat]
```

This parser gives the correct results, but is inefficient. For example, when parsing the string `"123-456"` the number `123` will first be parsed by the `add` parser, that will then fail because there is no '`+`' symbol following the number. The correct parse will only be found by backtracking in the input string, and parsing the number `123` *again*, this time from within the `sub` parser.

Of course, the way to avoid the possibility of backtracking and repeated parsing is to *left factorise* the `eval` parser. That is, the initial use of `nat` in the component parsers `add` and `sub` should be factorised out:

```
eval = [v | x <- nat, v <- add x ++ sub x]
       where
          add x = [x+y | _ <- char '+', y <- nat]
          sub x = [x+y | _ <- char '-', y <- nat]
```

This new version of `eval` gives the same results as the original version, but requires no backtracking. Using the new `eval`, the string `"123-456"` can now be parsed in linear time. In fact we can go a little further, and right factorise the remaining use of `nat` in both `add` and `sub`. This does not improve the efficiency of `eval`, but arguably gives a cleaner parser:

```
eval = [f x y | x <- nat
              , f <- ops [(char '+', (+)), (char '-', (-))]
              , y <- nat]
```

In practice, most cases where left factorisation of a parser is necessary to improve efficiency will concern parsers for some kind of expression. In such cases, manually factorising the parser will not be required, since expression-like parsers can be built using the `chain` combinators from the previous section, which already encapsulate the necessary left factorisation.

The motto of this section is the following: backtracking is a powerful tool, but it should not be used as a substitute for care in designing parsers.

5.2 Improving lazyness

Recall the definition of the repetition combinator `many`:

```
many   :: Parser a -> Parser [a]
many p = [x:xs | x <- p, xs <- many p] ++ [[]]
```

For example, applying `many (char 'a')` to the input `"aaab"` gives the result `[("aaa","b"), ("aa","ab"), ("a","aab"),("","aaab")]`. Since Gofer is lazy, we would expect the `a`'s in the first result `"aaa"` to become available one at a time, as they are consumed from the input. This is not in fact what happens. In practice no part of the result `"aaa"` will be produced until all the `a`'s have been consumed. In other words, `many` is not as lazy as we would expect.

But does this really matter? Yes, because it is common in functional programming to rely on lazyness to avoid the creation of large intermediate structures (Hughes, 1989). As noted by Wadler (1985; 1992b), what is needed to solve the problem with `many` is a means to make explicit that the parser `many p` always succeeds. (Even if `p` itself always fails, `many p` will still succeed, with the empty list as the result value.) This is the purpose of the `force` combinator:

```
force  :: Parser a -> Parser a
force p = \inp -> let x = p inp in
                  (fst (head x), snd (head x)) : tail x
```

Given a parser `p` that always succeeds, the parser `force p` has the same behaviour as `p`, except that before any parsing of the input string is attempted the result of the parser is immediately forced to take on the form $(\perp, \perp) : \perp$, where $\perp$ represents a presently undefined value.

Using `force`, the `many` combinator can be re-defined as follows:

```
many  :: Parser a -> Parser [a]
many p = force ([x:xs | x <- p, xs <- many p] ++ [[]])
```

The use of `force` ensures that `many p` and all of its recursive calls return at least one result. The new definition of `many` now has the expected behaviour under lazy evaluation. For example, applying `many (char 'a')` to the partially-defined string $'\texttt{a}' : \perp$ gives the partially-defined result $('\texttt{a}' : \perp, \perp) : \perp$. In contrast, with the old version of `many`, the result for this example is the completely undefined value $\perp$.

Some readers might wonder why `force` is defined using the following selection functions, rather than by pattern matching?

```
fst :: (a,b) -> a      head :: [a] -> a
snd :: (a,b) -> b      tail :: [a] -> [a]
```

The answer is that, depending on the semantics of patterns in the particular implementation language, a definition of `force` using patterns might not have the expected behaviour under lazy evaluation.

5.3 Limiting the number of results

Consider the simple problem of parsing a natural number, or if no such number is present just returning the number 0 as the default result. A first approximation to such a parser might be as follows:

```
number :: Parser Int
number  = nat ++ [0]
```

However, this does not quite have the required behaviour. For example, applying `number` to the input `"hello"` gives the correct result `[(0,"hello")]`. On the other hand, applying `number` to `"123"` gives the result `[(123,""), (0,"123")]`, whereas we only really want the single result `[(123,"")]`.

One solution to the above problem is to make use of *deterministic* parser combinators (see section 7.5) — all parsers built using such combinators are restricted by construction to producing at most one result. A more general solution, however, is to retain the flexibility of the non-deterministic combinators, but to provide a means to make explicit that we are only interested in the first result produced by certain parsers, such as `number`. This is the purpose of the `first` combinator:

```
first  :: Parser a -> Parser a
first p = \inp -> case p inp of
                     []     -> []
                     (x:xs) -> [x]
```

Given a parser `p`, the parser `first p` has the same behaviour as `p`, except that only the first result (if any) is returned. Using `first` we can define a deterministic version `(+++)` of the standard choice combinator `(++)` for parsers:

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

Replacing `(++)` by `(+++)` in `number` gives the desired behaviour.

As well as being used to ensure the correct behaviour of parsers, using `(+++)` can also improve their efficiency. As an example, consider a parser that accepts either of the strings `"yellow"` or `"orange"`:

```
colour :: Parser String
colour  = p1 ++ p2
          where
             p1 = string "yellow"
             p2 = string "orange"
```

Recall now the behaviour of the choice combinator `(++)`: it takes a string, applies both argument parsers to this string, and concatenates the resulting lists. Thus in the `colour` example, if `p1` is successfully applied then `p2` will still be applied to the same string, even though it is guaranteed to fail. This inefficiency can be avoided using `(+++)`, which ensures that if `p1` succeeds then `p2` is never applied:

```
colour = p1 +++ p2
         where
            p1 = string "yellow"
            p2 = string "orange"
```

More generally, if we know that a parser of the form `p ++ q` is deterministic (only ever returns at most one result value), then `p +++ q` has the same behaviour, but is more efficient: if `p` succeeds then `q` is never applied. In the remainder of this article it will mostly be the `(+++)` choice combinator that is used. For reasons of efficiency, in the combinator libraries that accompany this article, the repetition combinators from the previous section are defined using `(+++)` rather than `(++)`.

We conclude this section by asking why `first` is defined by pattern matching, rather than by using the selection function `take :: Int -> [a] -> [a]` (where, for example, `take 3 "parsing" = "par"`):

```
first p = \inp -> take 1 (p inp)
```

The answer concerns the behaviour under lazy evaluation. To see the problem, let us unfold the use of `take` in the above definition:

```
first p = \inp -> case p inp of
                     []     -> []
                     (x:xs) -> x : take 0 xs
```

When the sub-expression `take 0 xs` is evaluated, it will yield `[]`. However, under lazy evaluation this computation will be suspended until its value is required. The effect is that the list `xs` may be retained in memory for some time, when in fact it can safely be discarded immediately. This is an example of a *space leak*. The definition of `first` using pattern matching does not suffer from this problem.

6 Handling lexical issues

Traditionally, a string to be parsed is not supplied directly to a parser, but is first passed through a lexical analysis phase (or *lexer*) that breaks the string into a sequence of tokens (Aho *et al.*, 1986). Lexical analysis is a convenient place to remove white-space (spaces, newlines, and tabs) and comments from the input string, and to distinguish between identifiers and keywords.

Since lexers are just simple parsers, they can be built using parser combinators, as discussed by Hutton (1992). However, as we shall see in this section, the need for a separate lexer can often be avoided (even for substantial grammars such as that for Gofer), with lexical issues being handled within the main parser by using some special purpose combinators.

6.1 White-space, comments, and keywords

We begin by defining a parser that consumes white-space from the beginning of a string, with a dummy value `()` returned as result:

```
spaces :: Parser ()
spaces  = [() | _ <- many1 (sat isSpace)]
          where
             isSpace x =
                (x == ' ') || (x == '\n') || (x == '\t')
```

Similarly, a single-line Gofer comment can be consumed as follows:

```
comment :: Parser ()
comment  = [() | _ <- string "--"
               , _ <- many (sat (\x -> x /= '\n'))]
```

We leave it as an exercise for the reader to define a parser for consuming multi-line Gofer comments `{- ... -}`, which can be nested.

After consuming white-space, there may still be a comment left to consume from the input string. Dually, after a comment there may still be white-space. Thus we are motivated to defined a special parser that repeatedly consumes white-space and comments until no more remain:

```
junk :: Parser ()
junk  = [() | _ <- many (spaces +++ comment)]
```

Note that while `spaces` and `comment` can fail, the `junk` parser always succeeds. We define two combinators in terms of `junk`: `parse` removes junk *before* applying a given parser, and `token` removes junk *after* applying a parser:

```
parse  :: Parser a -> Parser a
parse p = [v | _ <- junk, v <- p]

token  :: Parser a -> Parser a
token p = [v | v <- p, _ <- junk]
```

With the aid of these two combinators, parsers can be modified to ignore white-space and comments. Firstly, `parse` is applied once to the parser as a whole, ensuring that input to the parser begins at a significant character. And secondly, `token` is applied once to all sub-parsers that consume complete tokens, thus ensuring that the input always remains at a significant character.

Examples of parsers for complete tokens are `nat` and `int` (for natural numbers and integers), parsers of the form `string xs` (for symbols and keywords), and `ident` (for identifiers). It is useful to define special versions of these parsers — and more generally, special versions of any user-defined parsers for complete tokens — that encapsulate the necessary application of `token`:

```
natural      :: Parser Int
natural       = token nat

integer      :: Parser Int
integer       = token int

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

identifier   :: [String] -> Parser String
identifier ks = token [x | x <- ident, not (elem x ks)]
```

Note that `identifier` takes a list of keywords as an argument, where a keyword is a string that is not permitted as an identifier. For example, in Gofer the strings "`data`" and "`where`" (among others) are keywords. Without the keyword check, parsers defined in terms of `identifier` could produce unexpected results, or involve unnecessary backtracking to construct the correct parse of the input string.

6.2 A parser for λ-expressions

To illustrate the use of the new combinators given above, let us define a parser for simple λ-expressions extended with a "`let`" construct for local definitions. Parsed expressions will be represented in Gofer as follows:

```
data Expr = App Expr Expr          -- application
          | Lam String Expr        -- lambda abstraction
          | Let String Expr Expr   -- local definition
          | Var String             -- variable
```

Now a parser `expr :: Parser Expr` can be defined by:

```
expr     = atom `chainl1` [App]

atom     = lam +++ local +++ var +++ paren

lam      = [Lam x e | _ <- symbol "\\"
                    , x <- variable
                    , _ <- symbol "->"
                    , e <- expr]

local    = [Let x e e' | _  <- symbol "let"
                       , x  <- variable
                       , _  <- symbol "="
                       , e  <- expr
                       , _  <- symbol "in"
                       , e' <- expr]

var      = [Var x | x <- variable]

paren    = bracket (symbol "(") expr (symbol ")")

variable = identifier ["let","in"]
```

Note how the `expr` parser handles white-space and comments by using the `symbol` parser in place of `string` and `char`. Similarly, the keywords "`let`" and "`in`" are handled by using `identifier` to define the parser for variables. Finally, note how applications `(f e1 e2 ... en)` are parsed in the form `(((f e1) e2) ... )` by using the `chainl1` combinator.

7 Factorising the parser monad

Up to this point in the article, combinator parsers have been our only example of the notion of a monad. In this section we define a number of other monads related to the parser monad, leading up to a modular reformulation of the parser monad in terms of two simpler monads (Jones, 1995a). The immediate benefit is that, as we shall see, the basic parser combinators no longer need to be defined explicitly. Rather, they arise automatically as a special case of lifting monad operations from a base monad m to a certain other monad parameterised over m. This also means that, if we change the nature of parsers by modifying the base monad (for example, limiting parsers to producing at most one result), new combinators for the modified monad of parsers are also defined automatically.

7.1 The exception monad

Before starting to define other monads, it is useful to first focus briefly on the intuition behind the use of monads in functional programming (Wadler, 1992a).

The basic idea behind monads is to distinguish the *values* that a computation can produce from the *computation* itself. More specifically, given a monad `m` and a type `a`, we can think of `m a` as the type of computations that yield results of type `a`, with the nature of the computation captured by the type constructor `m`. The combinators `result` and `bind` (with `zero` and `(++)` if appropriate) provide a means to structure the building of such computations:

```
result :: m a
bind   :: m a -> (a -> m b) -> m b
zero   :: m a
(++)   :: m a -> m a -> m a
```

From a computational point of view, `result` converts values into computations that yield those values; `bind` chains two computations together in sequence, with results of the first computation being made available for use in the second; `zero` is the trivial computation that does nothing; and finally, `(++)` is some kind of choice operation for computations.

Consider, for example, the type constructor `Maybe`:

```
data Maybe a = Just a | Nothing
```

We can think of a value of type `Maybe a` as a computation that either succeeds with a value of type `a`, or fails, producing no value. Thus, the type constructor `Maybe` captures computations that have the possibility to fail.

Defining the monad combinators for a given type constructor is usually just a matter of making the "obvious definitions" suggested by the types of the combinators. For example, the type constructor `Maybe` can be made into a monad with a zero and plus using the following definitions:

```
instance Monad Maybe where
   -- result        :: a -> Maybe a
   result x          = Just x

   -- bind          :: Maybe a -> (a -> Maybe b) -> Maybe b
   (Just x) `bind` f = f x
   Nothing  `bind` f = Nothing
```

```
instance Monad0Plus Maybe where
   -- zero          :: Maybe a
   zero              = Nothing

   -- (++)          :: Maybe a -> Maybe a -> Maybe a
   Just x  ++ y      = Just x
   Nothing ++ y      = y
```

That is, `result` converts a value into a computation that succeeds with this value; `bind` is a sequencing operator, with a successful result from the first computation being available for use in the second computation; `zero` is the computation that fails; and finally, `(++)` is a (deterministic) choice operator that returns the first computation if it succeeds, and the second otherwise.

Since failure can be viewed as a simple kind of exception, `Maybe` is sometimes called the *exception* monad in the literature (Spivey, 1990).

7.2 The non-determinism monad

A natural generalisation of `Maybe` is the list type constructor `[]`. While a value of type `Maybe a` can be thought of as a computation that either succeeds with a single result of type a or fails, a value of type `[a]` can be thought of as a computation that has the possibility to succeed with any number of results of type `a`, including zero (which represents failure). Thus the list type constructor `[]` can be used to capture *non-deterministic* computations.

Now `[]` can be made into a monad with a zero and plus:

```
instance Monad [] where
   -- result      :: a -> [a]
   result x        = [x]

   -- bind        :: [a] -> (a -> [b]) -> [b]
   []     `bind` f = []
   (x:xs) `bind` f = f x ++ (xs `bind` f)

instance Monad0Plus [] where
   -- zero        :: [a]
   zero            = []

   -- (++)        :: [a] -> [a] -> [a]
   []     ++ ys    = ys
   (x:xs) ++ ys    = x : (xs ++ ys)
```

That is, `result` converts a value into a computation that succeeds with this single value; `bind` is a sequencing operator for non-deterministic computations; `zero` always fails; and finally, `(++)` is a (non-deterministic) choice operator that appends the results of the two argument computations.

7.3 The state-transformer monad

Consider the (binary) type constructor `State`:

```
type State s a = s -> (a,s)
```

Values of type `State s a` can be interpreted as follows: they are computations that take an initial state of type `s`, and yield a value of type `a` together with a new state of type `s`. Thus, the type constructor `State s` obtained by applying `State` to a single type `s` captures computations that involve state of type `s`. We will refer to values of type `State s a` as *stateful computations*.

Now `State s` can be made into a monad:

```
instance Monad (State s) where
   -- result  :: a -> State s a
   result v    = \s -> (v,s)

   -- bind    :: State s a -> (a -> State s b) -> State s b
   st `bind` f = \s -> let (v,s') = st s in f v s'
```

That is, `result` converts a value into a stateful computation that returns that value without modifying the internal state, and `bind` composes two stateful computations in sequence, with the result value from the first being supplied as input to the second. Thinking pictorially in terms of boxes and wires is a useful aid to becoming familiar with these two operations (Jones & Launchbury, 1994).

The *state-transformer* monad `State s` does not have a zero and a plus. However, as we shall see in the next section, the *parameterised* state-transformer monad over a given based monad `m` *does* have a zero and a plus, provided that `m` does.

To allow us to access and modify the internal state, a few extra operations on the monad `State s` are introduced. The first operation, `update`, modifies the state by applying a given function, and returns the old state as the result value of the computation. The remaining two operations are defined in terms of `update`: `set` replaces the state with a new state, and returns the old state as the result; `fetch` returns the state without modifying it.

```
update  :: (s -> s) -> State s s
set     :: s -> State s s
fetch   :: State s s

update f = \s -> (s, f s)
set s    = update (\_ -> s)
fetch    = update id
```

In fact `State s` is not the only monad for which it makes sense to define these operations. For this reason we encapsulate the extra operations in a class, so that the same names can be used for the operations of different monads:

```
class Monad m => StateMonad m s where
   update :: (s -> s) -> m s
```

```
  set    :: s -> m s
  fetch  :: m s

  set s   = update (\_ -> s)
  fetch   = update id
```

This declaration can be read as follows: a type constructor `m` and a type `s` are together a member of the class `StateMonad` if `m` is a member of the class `Monad`, and if `m` is also equipped with `update`, `set`, and `fetch` operations of the specified types. Moreover, the fact that `set` and `fetch` can be defined in terms of `update` is also reflected in the declaration, by means of default definitions.

Now because `State s` is already a monad, it can be made into a state monad using the `update` operation as defined earlier:

```
instance StateMonad (State s) s where
   -- update :: (s -> s) -> State s s
   update f   = \s -> (s, f s)
```

7.4 The parameterised state-transformer monad

Recall now our type of combinator parsers:

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

We see now that parsers combine two kinds of computation: non-deterministic computations (the result of a parser is a list), and stateful computations (the state is the string being parsed). Abstracting from the specific case of returning a list of results, the `Parser` type gives rise to a generalised version of the `State` type constructor that applies a given type constructor `m` to the result of the computation:

```
type StateM m s a = s -> m (a,s)
```

Now `StateM m s` can be made into a monad with a zero and a plus, by inheriting the monad operations from the base monad `m`:

```
instance Monad m => Monad (StateM m s) where
   -- result   :: a -> StateM m s a
   result v     = \s -> result (v,s)

   -- bind     :: StateM m s a ->
   --                (a -> StateM m s b) -> StateM m s b
   stm `bind` f = \s -> stm s `bind` \(v,s') -> f v s'

instance Monad0Plus m => Monad0Plus (StateM m s) where
   -- zero     :: StateM m s a
   zero         = \s -> zero

   -- (++)     :: StateM m s a -> StateM m s a -> StateM m s a
   stm ++ stm'  = \s -> stm s ++ stm' s
```

That is, `result` converts a value into a computation that returns this value without modifying the internal state; `bind` chains two computations together; `zero` is the computation that fails regardless of the input state; and finally, `(++)` is a choice operation that passes the same input state through to both of the argument computations, and combines their results.

In the previous section we defined the extra operations `update`, `set` and `fetch` for the monad `State s`. Of course, these operations can also be defined for the *parameterised state-transformer monad* `StateM m s`. As previously, we only need to define `update`, the remaining two operations being defined automatically via default definitions:

```
instance Monad m => StateMonad (StateM m s) s where
   -- update :: Monad m => (s -> s) -> StateM m s s
   update f   = \s -> result (s, f s)
```

7.5 The parser monad revisited

Recall once again our type of combinator parsers:

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

This type can now be re-expressed using the parameterised state-transformer monad `StateM m s` by taking `[]` for `m`, and `String` for `s`:

```
type Parser a = StateM [] String a
```

But why view the `Parser` type in this way? The answer is that all the basic parser combinators no longer need to be defined explicitly (except one, the parser `item` for single characters), but rather arise as an instance of the general case of extending monad operations from a type constructor `m` to the type constructor `StateM m s`. More specifically, since `[]` forms a monad with a zero and a plus, so does `State [] String`, and hence Gofer automatically provides the following combinators:

```
result :: a -> Parser a
bind   :: Parser a -> (a -> Parser b) -> Parser b
zero   :: Parser a
(++)   :: Parser a -> Parser a -> Parser a
```

Moreover, defining the parser monad in this modular way in terms of `StateM` means that, if we change the type of parsers, then new combinators for the modified type are also defined automatically. For example, consider replacing

```
type Parser a = StateM [] String a
```

by a new definition in which the list type constructor `[]` (which captures non-deterministic computations that can return many results) is replaced by the `Maybe` type constructor (which captures deterministic computations that either fail, returning no result, or succeed with a single result):

```
data Maybe a  = Just a | Nothing

type Parser a = StateM Maybe String a
```

Since `Maybe` forms a monad with a zero and a plus, so does the re-defined `Parser` type constructor, and hence Gofer automatically provides `result`, `bind`, `zero`, and `(++)` combinators for deterministic parsers. In earlier approaches that do not exploit the monadic nature of parsers (Wadler, 1985; Hutton, 1992; Fokker, 1995), the basic combinators would have to be re-defined by hand.

The only basic parsing primitive that does not arise from the monadic structure of the `Parser` type is the parser `item` for consuming single characters:

```
item :: Parser Char
item  = \inp -> case inp of
                  []     -> []
                  (x:xs) -> [(x,xs)]
```

However, `item` can now be re-defined in monadic style. We first fetch the current state (the input string); if the string is empty then the `item` parser fails, otherwise the first character is consumed (by applying the `tail` function to the state), and returned as the result value of the parser:

```
item  = [x | (x:_) <- update tail]
```

The advantage of the monadic definition of `item` is that it does not depend upon the internal details of the `Parser` type. Thus, for example, it works equally well for both the non-deterministic and deterministic versions of `Parser`.

8 Handling the offside rule

Earlier (section 6) we showed that the need for a lexer to handle white-space, comments, and keywords can be avoided by using special combinators within the main parser. Another task usually performed by a lexer is handling the Gofer *offside rule*. This rule allows the grouping of definitions in a program to be indicated using indentation, and is usually implemented by the lexer inserting extra tokens (concerning indentation) into its output stream.

In this section we show that Gofer's offside rule can be handled in a simple and natural manner without a separate lexer, by once again using special combinators. Our approach was inspired by the monadic view of parsers, and is a development of an idea described earlier by Hutton (1992).

8.1 The offside rule

Consider the following simple Gofer program:

```
a = b + c
    where
       b = 10
```

```
         c = 15 - 5
d = a * 2
```

It is clear from the use of indentation that `a` and `d` are intended to be global definitions, with `b` and `c` local definitions to `a`. Indeed, the above program can be viewed as a shorthand for the following program, in which the grouping of definitions is made explicit using special brackets and separators:

```
{ a = b + c
      where
         { b = 10
         ; c = 15 - 5 }
; d = a * 2 }
```

How the grouping of Gofer definitions follows from their indentation is formally specified by the *offside rule*. The essence of the rule is as follows: consecutive definitions that begin in the same column c are deemed to be part of the same group. To make parsing easier, it is further required that the remainder of the text of each definition (excluding white-space and comments, of course) in a group must occur in a column strictly greater than c. In terms of the offside rule then, definitions `a` and `d` in the example program above are formally grouped together (and similarly for `b` and `c`) because they start in the same column as one another.

8.2 Modifying the type of parsers

To implement the offside rule, we will have to maintain some extra information during parsing. First of all, since column numbers play a crucial role in the offside rule, parsers will need to know the column number of the first character in their input string. In fact, it turns out that parsers will also require the current line number. Thus our present type of combinator parsers,

```
type Parser a = StateM [] String a
```

is revised to the following type, in which the internal state of a parser now contains a (line,column) position in addition to a string:

```
type Parser a = StateM [] Pstring a

type Pstring  = (Pos,String)

type Pos      = (Int,Int)
```

In addition, parsers will need to know the starting position of the current definition being parsed — if the offside rule is not in effect, this *definition position* can be set with a negative column number. Thus our type of parsers is revised once more, to take the current definition position as an extra argument:

```
type Parser a = Pos -> StateM [] Pstring a
```

Another option would have been to maintain the definition position in the parser state, along with the current position and the string to be parsed. However, definition positions can be nested, and supplying the position as an extra argument to parsers — as opposed to within the parser state — is more natural from the point of view of implementing nesting of positions.

Is the revised `Parser` type still a monad? Abstracting from the details, the body of the `Parser` type definition is of the form `s -> m a` (in our case `s` is `Pos`, `m` is the monad `StateM [] Pstring`, and `a` is the parameter type `a`.) We recognise this as being similar to the type `s -> m (a,s)` of parameterised state-transformers, the difference being that the type `s` of states no longer occurs in the type of the result: in other words, the state can be read, but not modified. Thus we can think of `s -> m a` as the type of *parameterised state-readers*. The monadic nature of this type is the topic of the next section.

8.3 The parameterised state-reader monad

Consider the type constructor `ReaderM`, defined as follows:

```
type ReaderM m s a = s -> m a
```

In a similar way to `StateM m s`, `ReaderM m s` can be made into a monad with a zero and a plus, by inheriting the monad operations from the base monad `m`:

```
instance Monad m => Monad (ReaderM m s) where
   -- result   :: a -> ReaderM m s a
   result v     = \s -> result v

   -- bind     :: ReaderM m s a ->
   --                (a -> ReaderM m s b) -> ReaderM m s b
   srm `bind` f = \s -> srm s `bind` \v -> f v s

instance Monad0Plus m => Monad0Plus (ReaderM m s) where
   -- zero     :: ReaderM m s a
   zero         = \s -> zero

   -- (++)     :: ReaderM m s a ->
   --                ReaderM m s a -> ReaderM m s a
   srm ++ srm'  = \s -> srm s ++ srm' s
```

That is, `result` converts a value into a computation that returns this value without consulting the state; `bind` chains two computations together, with the same state being passed to both computations (contrast with the `bind` operation for `StateM`, in which the second computation receives the new state produced by the first computation); `zero` is the computation that fails; and finally, `(++)` is a choice operation that passes the same state to both of the argument computations.

To allow us to access and set the state, a couple of extra operations on the *parameterised state-reader* monad `ReaderM m s` are introduced. As for `StateM`, we encapsulate the extra operations in a class. The operation **env** returns the state as the result of the computation, while **setenv** replaces the current state for a given computation with a new state:

```
class Monad m => ReaderMonad m s where
   env    :: m s
   setenv :: s -> m a -> m a

instance Monad m => ReaderMonad (ReaderM m s) s where
   -- env      :: Monad m => ReaderM m s s
   env          = \s -> result s

   -- setenv   :: Monad m => s ->
   --                 ReaderM m s a -> ReaderM m s a
   setenv s srm = \_ -> srm s
```

The name **env** comes from the fact that one can think of the state supplied to a state-reader as being a kind of *env*ironment. Indeed, in the literature state-reader monads are sometimes called *environment* monads.

8.4 The new parser combinators

Using the `ReaderM` type constructor, our revised type of parsers

```
type Parser a = Pos -> StateM [] Pstring
```

can now be expressed as follows:

```
type Parser a = ReaderM (StateM [] Pstring) Pos a
```

Now since `[]` forms a monad with a zero and a plus, so does `StateM [] Pstring`, and hence so does `ReaderM (StateM [] Pstring) Pos`. Thus Gofer automatically provides **result**, **bind**, **zero**, and **(++)** operations for parsers that can handle the offside rule. Since the type of parsers is now defined in terms of `ReaderM` at the top level, the extra operations **env** and **setenv** are also provided for parsers. Moreover, the extra operation `update` (and the derived operations `set` and `fetch`) from the underlying state monad can be lifted to the new type of parsers — or more generally, to any parameterised state-reader monad — by ignoring the environment:

```
instance StateMonad m a => StateMonad (ReaderM m s) a where
   -- update :: StateMonad m a => (a -> a) -> ReaderM m s a
   update f   = \_ -> update f
```

Now that the internal state of parsers has been modified (from `String` to `Pstring`), the parser `item` for consuming single characters from the input must also be modified. The new definition for `item` is similar to the old,

```
item :: Parser Char
item  = [x | (x:_) <- update tail]
```

except that the `item` parser now fails if the position of the character to be consumed is not *onside* with respect to current definition position:

```
item :: Parser Char
item  = [x | (pos,x:_) <- update newstate
           , defpos    <- env
           , onside pos defpos]
```

A position is onside if its column number is strictly greater than the current definition column. However, the first character of a new definition begins in the same column as the definition column, so this is handled as a special case:

```
onside              :: Pos -> Pos -> Bool
onside (l,c) (dl,dc) = (c > dc) || (l == dl)
```

The remaining auxiliary function, `newstate`, consumes the first character from the input string, and updates the current position accordingly (for example, if a newline character was consumed, the current line number is incremented, and the current column number is set back to zero):

```
newstate :: Pstring -> Pstring
newstate ((l,c),x:xs)
   = (newpos,xs)
     where
        newpos = case x of
                    '\n' -> (l+1,0)
                    '\t' -> (l,((c `div` 8)+1)*8)
                    _    -> (l,c+1)
```

One aspect of the offside rule still remains to be addressed: for the purposes of this rule, white-space and comments are not significant, and should always be successfully consumed even if they contain characters that are not onside. This can be handled by temporarily setting the definition position to $(0, -1)$ within the `junk` parser for white-space and comments:

```
junk :: Parser ()
junk  = [() | _ <- setenv (0,-1) (many (spaces +++ comment))]
```

All that remains now is to define a combinator that parses a sequence of definitions subject to the Gofer offside rule:

```
many1_offside  :: Parser a -> Parser [a]
many1_offside p = [vs | (pos,_) <- fetch
                      , vs      <- setenv pos (many1 (off p))]
```

That is, `many1_offside p` behaves just as `many1 (off p)`, except that within this parser the definition position is set to the current position. (There is no need to skip white-space and comments before setting the position, since this will already have been effected by proper use of the lexical combinators `token` and `parse`.) The auxiliary combinator `off` takes care of setting the definition position locally for each new definition in the sequence, where a new definition begins if the column position equals the definition column position:

```
off   :: Parser a -> Parser a
off p = [v | (dl,dc)    <- env
           , ((l,c),_) <- fetch
           , c == dc
           , v          <- setenv (l,dc) p]
```

For completeness, we also define a combinator `many_offside` that has the same behaviour as the combinator `many1_offside`, except that it can also parse an empty sequence of definitions:

```
many_offside :: Parser a -> Parser [a]
many_offside p = many1_offside p +++ [[]]
```

To illustrate the use of the new combinators defined above, let us modify our parser for λ-expressions (section 6.2) so that the "`let`" construct permits non-empty sequences of local definitions subject to the offside rule. The datatype `Expr` of expressions is first modified so that the `Let` constructor has type `[(String,Expr)] -> Expr` instead of `String -> Expr -> Expr`:

```
data Expr = ...
          | Let [(String,Expr)] Expr
          | ...
```

The only part of the parser that needs to be modified is the parser `local` for local definitions, which now accepts sequences:

```
local = [Let ds e | _  <- symbol "let"
                  , ds <- many1_offside defn
                  , _  <- symbol "in"
                  , e  <- expr]

defn  = [(x,e) | x <- identifier
               , _ <- symbol "="
               , e <- expr]
```

We conclude this section by noting that the use of the offside rule when laying out sequences of Gofer definitions is not mandatory. As shown in our initial example, one also has the option to include explicit layout information in the form of parentheses "{" and "}" around the sequence, with definitions separated by semi-colons "`;`". We leave it as an exercise to the reader to use `many_offside` to define a combinator that implements this convention.

In summary then, to permit combinator parsers to handle the Gofer offside rule, we changed the type of parsers to include some positional information, modified the `item` and `junk` combinators accordingly, and defined two new combinators: `many1_offside` and `many_offside`. All other necessary redefining of combinators is done automatically by the Gofer type system.

9 Acknowledgements

The first author was employed by the University of Utrecht during part of the writing of this article, for which funding is gratefully acknowledged.

Special thanks are due to Luc Duponcheel for many improvements to the implementation of the combinator libraries in Gofer (particularly concerning the use of type classes and restricted type synonyms), and to Mark P. Jones for detailed comments on the final draft of this article.

10 Appendix: a parser for data definitions

To illustrate the monadic parser combinators developed in this article in a real-life setting, we consider the problem of parsing a sequence of Gofer datatype definitions. An example of such a sequence is as follows:

```
data List a = Nil | Cons a (List a)

data Tree a b = Leaf a
              | Node (Tree a b, b, Tree a b)
```

Within the parser, datatypes will be represented as follows:

```
type Data = (String,             -- type name
            [String],            -- parameters
            [(String,[Type])])   -- constructors and arguments
```

The representation `Type` for types will be treated shortly. A parser `datadecls :: Parser [Data]` for a sequence of datatypes can now be defined by

```
datadecls   = many_offside datadecl

datadecl    = [(x,xs,b) | _  <- symbol "data"
                        , x  <- constructor
                        , xs <- many variable
                        , _  <- symbol "="
                        , b  <- condecl `sepby1` symbol "|"]

constructor = token [(x:xs) | x  <- upper
                            , xs <- many alphanum]

variable    = identifier ["data"]

condecl     = [(x,ts) | x  <- constructor
                      , ts <- many type2]
```

There are a couple of points worth noting about this parser. Firstly, all lexical issues (white-space and comments, the offside rule, and keywords) are handled by combinators. And secondly, since `constructor` is a parser for a complete token, the `token` combinator is applied within its definition.

Within the parser, types will be represented as follows:

```
data Type   = Arrow Type Type    -- function
            | Apply Type Type    -- application
            | Var String         -- variable
            | Con String         -- constructor
            | Tuple [Type]       -- tuple
            | List Type          -- list
```

A parser `type0 :: Parser Type` for types can now be defined by

```
type0       = type1 `chainr1` [Arrow | _ <- symbol "->"]
type1       = type2 `chainl1` [Apply]
type2       = var +++ con +++ list +++ tuple

var         = [Var x | x <- variable]

con         = [Con x | x <- constructor]

list        = bracket (symbol "[") type0 (symbol "]")

tuple       = [f ts | ts <- bracket
                              (symbol "(")
                              (type0 `sepby` symbol ",")
                              (symbol ")")]
              where f [t] = t
                    f ts  = Tuple ts
```

Note how `chainr1` and `chainl1` are used to handle parsing of function-types and application. Note also that (as in Gofer) building a singleton tuple `(t)` of a type `t` is not possible, since `(t)` is treated as a parenthesised expression.

References

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers — principles, techniques and tools.* Addison-Wesley.

Burge, W.H. (1975). *Recursive programming techniques.* Addison-Wesley.

Fokker, Jeroen. 1995 (May). Functional parsers. *Proc. lecture notes of the Baastad Spring school on functional programming.*

Gill, Andy, & Marlow, Simon. 1995 (Jan.). *Happy: the parser generator for Haskell.* University of Glasgow.

Hughes, John. (1989). Why functional programming matters. *The computer journal,* **32**(2), 98–107.

Hutton, Graham. (1992). Higher-order functions for parsing. *Journal of functional programming,* **2**(3), 323–343.

Jones, Mark P. (1994). *Gofer 2.30a release notes.* Unpublished manuscript.

Jones, Mark P. (1995a). Functional programming beyond the Hindley/Milner type system. *Proc. lecture notes of the Baastad spring school on functional programming.*

Jones, Mark P. (1995b). *The Gofer distribution.* Available from the University of Nottingham: `http://www.cs.nott.ac.uk/Department/Staff/mpj/`.

Jones, Mark P. (1995c). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Jones, Simon Peyton, & Launchbury, John. (1994). *State in Haskell.* University of Glasgow.

Landin, Peter. (1966). The next 700 programming languages. *Communications of the ACM*, **9**(3).

Mogensen, Torben. (1993). *Ratatosk: a parser generator and scanner generator for Gofer.* University of Copenhagen (DIKU).

Moggi, Eugenio. (1989). Computation lambda-calculus and monads. *Proc. IEEE symposium on logic in computer science.* A extended version of the paper is available as a technical report from the University of Edinburgh.

Röjemo, Niklas. (1995). *Garbage collection and memory efficiency in lazy functional languages.* Ph.D. thesis, Chalmers University of Technology.

Spivey, Mike. (1990). A functional theory of exceptions. *Science of computer programming*, **14**, 25–42.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Proc. conference on functional programming and computer architecture.* Springer–Verlag.

Wadler, Philip. (1990). Comprehending monads. *Proc. ACM conference on Lisp and functional programming.*

Wadler, Philip. (1992a). The essence of functional programming. *Proc. principles of programming languages.*

Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), *Proc. Marktoberdorf Summer school on program design calculi.* Springer–Verlag.