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

Higher-Order Functions for Parsing†

Graham Hutton
Department of Computer Science, University of Utrecht,
PO Box 80.089, 3508 TB Utrecht, The Netherlands.

Abstract

In *combinator parsing*, the text of parsers resembles BNF notation. We present the basic method, and a number of extensions. We address the special problems presented by white–space, and parsers with separate lexical and syntactic phases. In particular, a combining form for handling the "offside rule" is given. Other extensions to the basic method include an "into" combining form with many useful applications, and a simple means by which combinator parsers can produce more informative error messages.

1 Introduction

Broadly speaking, a parser may be defined as a program which analyses text to determine its logical structure. For example, the parsing phase in a compiler takes a program text, and produces a parse tree which expounds the structure of the program. Many programs can be improved by having their input parsed. The form of input which is acceptable is usually defined by a context–free grammar, using BNF notation. Parsers themselves may be built by hand, but are most often generated automatically using tools like Lex and Yacc from Unix (Aho86).

Although there are many methods of building parsing, one in particular has gained widespread acceptance for use in lazy functional languages. In this method, parsers are modelled directly as functions; larger parsers are built piecewise from smaller parsers using higher order functions. For example, we define higher order functions for sequencing, alternation and repetition. In this way, the text of parsers closely resembles BNF notation. Parsers in this style are quick to build, and simple to understand and modify. In the sequel, we refer to the method as *combinator parsing*, after the higher order functions used to combine parsers.

Combinator parsing is considerably more powerful than the commonly used methods, being able to handle ambiguous grammars, and providing full backtracking if it is needed. In fact, we can do more than just parsing. Semantic actions can be added to parsers, allowing their results to be manipulated in any way we please. For example, in section 2.4 we convert a parser for arithmetic expressions to an evaluator simply by changing the semantic actions. More generally, we could imagine generating some form of abstract machine code as programs are parsed.

† Appears in the Journal of Functional Programming 2(3):323–343, July 1992.

Although the principles are widely known (due in most part to (Wadler85)), little has been written on combinator parsing itself. In this article, we present the basic method, and a number of extensions. The techniques may be used in any lazy functional language with a higher–order/polymorphic style type system. All our programming examples are given in Miranda†; features and standard functions are explained as they are used. A library of parsing functions taken from this paper is available by electronic mail from the author. Versions exist in both Miranda and Lazy ML.

2 Parsing Using Combinators

We begin by defining a *type* of parsers. A parser may be viewed as a function from a string of symbols to a result value. Since a parser might not consume the entire string, part of this result will be a suffix of the input string. Sometimes a parser may not be able to produce a result at all. For example, it may be expecting a letter, but find a digit. Rather than defining a special type for the success or failure of a parser, we choose to have parsers return a list of pairs as their result, with the empty list `[]` denoting failure, and a singleton list `[(v,xs)]` indicating success, with value `v` and unconsumed input `xs`. As we shall see in section 2.2, having parsers return a list of results proves very useful. Since we want to specify the type of any parser, regardless of the kind of symbols and results involved, these types are included as extra parameters. In Miranda, type variables are denoted by sequences of stars.

```
parser * ** == [*] -> [(**,[*])]
```

For example, a parser for arithmetic expressions might have type (`parser char expr`), indicating that it takes a string of characters, and produces an expression tree. Notice that `parser` is not a new type as such, but an abbreviation (or synonym); it's only purpose is to make types involving parsers easier to understand.

2.1 Primitive parsers

The primitive parsers are the building blocks of combinator parsing. The first of these corresponds to the ε symbol in BNF notation, denoting the empty string. The `succeed` parser always succeeds, without actually consuming any of the input string. Since the outcome of `succeed` does not depend upon its input, its result value must be pre–determined, so is included as an extra parameter:

```
succeed :: ** -> parser * **

succeed v inp = [(v,inp)]
```

This definition relies on partial application to work properly. The order of the arguments means that if `succeed` is supplied only one argument, the result is a parser (i.e. a function) which always succeeds with this value. For example, (`succeed 5`)

† Miranda is a trademark of Research Software Limited.

is a parser which always returns the value `5`. Furthermore, even though `succeed` plainly has two arguments, its type would suggest it has only one. There is no magic, the second argument is simply hidden inside the type of the result, as would be clear upon expansion of the type according to the `parser` abbreviation.

While `succeed` never fails, `fail` always does, regardless of the input string:

```
fail :: parser * **

fail inp = []
```

The next function allows us to make parsers that recognise single symbols. Rather than enumerating the acceptable symbols, we find it more convenient to provide the set implicitly, via a predicate which determines if an arbitrary symbol is a member. Successful parses return the consumed symbol as their result value.

```
satisfy  :: (* -> bool) -> parser * *

satisfy p []     = fail []
satisfy p (x:xs) = succeed x xs , p x
                 = fail xs      , otherwise
```

Notice how `succeed` and `fail` are used in this example. Although they are not strictly necessary, their presence makes the parser easier to read. Note also that the parser (`satisfy p`) returns failure if supplied with an empty input string.

Using `satisfy` we can define a parser for single symbols:

```
literal  :: * -> parser * *

literal x = satisfy (=x)
```

For example, applying the parser (`literal '3'`) to the string `"345"` gives the result `[('3',"45")]`. In the definition of `literal`, `(=x)` is a function which tests its argument for equality with `x`. It is an example of *operator sectioning*, a useful syntactic convention which allows us to partially apply infix operators.

2.2 Combinators

Now that we have the basic building blocks, we consider how they should be put together to form useful parsers. In BNF notation, larger grammars are built piecewise from smaller ones using | to denote alternation, and juxtaposition to indicate sequencing. So that our parsers resemble BNF notation, we define higher order functions which correspond directly to these operators. Since higher order functions like these combine parsers to form other parsers, they are often referred to as *combining forms* or *combinators*. We will use these terms from now on.

The `alt` combinator corresponds to alternation in BNF. The parser (`p1 $alt p2`) recognises anything that either `p1` or `p2` would. Normally we would interpret *either* in a sequential (or exclusive) manner, returning the result of the first parser to succeed, and failure if neither does. This approach is taken in (Fairbairn86).

In combinator parsing however, we use inclusive *either* — it is acceptable for both parsers to succeed, in which case we return both results. In general then, combinator parsers may return an arbitrary number of results. This explains our decision earlier to have parsers return a list of results.

With parsers returning a list, `alt` is implemented simply by appending (denoted by `++` in Miranda) the result of applying both parsers to the input string. In keeping with the BNF notation, we use the Miranda $ notation to convert `alt` to an infix operator. Just as for sectioning, the infix notation is merely a syntactic convenience: (`x $f y`) is equivalent to (`f x y`) in all contexts.

```
alt :: parser * ** -> parser * ** -> parser * **

(p1 $alt p2) inp = p1 inp ++ p2 inp
```

Knowing that the empty–list `[]` is the identity element for `++`, it is easy to verify from this definition that failure is the identity element for alternation: (`fail $alt p`) = (`p $alt fail`) = `p`. In practical terms this means that `alt` has the expected behaviour if only one of the argument parsers succeeds. Similarly, `alt` inherits associativity from `++`: (`p $alt q`) `$alt r` = `p $alt` (`q $alt r`). This means we do not not need to worry about bracketing repeated alternation correctly.

Allowing parsers to produce more than one result allows us to handle ambiguous grammars, with all possible parses being produced for an ambiguous string. The feature has proved particularly useful in natural language processing (Frost88). An example ambiguous string from (Frost88) is "Who discovered a moon that orbits Mars or Jupiter ?" Most often however, we are only interested in the single longest parse of a string (i.e. that which consumes the most symbols). For this reason, it is normal in combinator parsing to arrange for the parses to be returned in descending order of length. All that is required is a little care in the ordering of the argument parsers to `alt`. See for example the `many` combinator in the next section.

The `then` combinator corresponds to sequencing in BNF. The parser (`p1 $then p2`) recognises anything that `p1` and `p2` would if placed in succession. Since the first parser may succeed with many results, each with an input stream suffix, the second parser must be applied to each of these in turn. In this manner, two results are produced for each successful parse, one from each parser. They are combined (by pairing) to form a single result for the compound parser.

```
then :: parser * ** -> parser * *** -> parser * (**,***)

(p1 $then p2) inp = [((v1,v2),out2) | (v1,out1) <- p1 inp;
                                      (v2,out2) <- p2 out1]
```

For example, applying the parser (`literal 'a' $then literal 'b'`) to the input `"abcd"` gives the result `[(('a','b'),"cd")]`. The `then` combinator is an excellent example of *list comprehension* notation, analogous to set comprehension in mathematics (e.g. $\{x^2 \mid x \in \mathbb{N} \wedge x < 10\}$ defines the first ten squares), except that lists replace sets, and elements are drawn in a determined order. Much of the elegance of the `then` combinator would be lost if this notation were not available.

Unlike alternation, sequencing is not associative, due to the tupling of results from the component parsers. In Miranda, all infix operators made using the $ notation are assumed to associate to the right. Thus, when we write (`p $then q $then r`) it is interpreted as (`p $then` (`q $then r`)).

2.3 Manipulating values

Part of the result from a parser is a value. The `using` combinator allows us to manipulate these results, building a parse tree being the most common application. The parser (`p $using f`) has the same behaviour as the parser `p`, except that the function `f` is applied to each of its result values:

```
using :: parser * ** -> (** -> ***) -> parser * ***

(p $using f) inp = [(f v,out) | (v,out) <- p inp]
```

Although `using` has no counterpart in pure BNF notation, it does have much in common with the $\{\cdots\}$ operator in Yacc (Aho86). In fact, the `using` combinator does not restrict us to building parse trees. Arbitrary semantic actions can be used. For example, in section 2.4 we convert a parser for arithmetic expressions to an evaluator simply by changing the actions. There is a clear connection here with *attribute grammers*. A recent and relevant article on attribute grammars is (Johnsson87). A combinator parser may be viewed as the implementation in a lazy functional language of an attribute grammar in which every node has one *inherited* attribute (the input string), and two *synthesised* attributes (the result value of the parse and the unconsumed part of the input string.) In the remainder of this section we define some useful new parsers and combinators in terms of our primitives.

In BNF notation, repetition occurs often enough to merit its own abbreviation. When zero or more repetitions of a phrase p are admissible, we simply write p^*. Formally, this notation is defined by the equation $p^* = p\ p^* \mid \varepsilon$. The `many` combinator corresponds directly to this operator, and is defined in much the same way:

```
many :: parser * ** -> parser * [**]

many p = ((p $then many p) $using cons) $alt (succeed [])
```

The action `cons` is the uncurried version of the list constructor "`:`", and is defined by `cons (x,xs)` = `x:xs`. Since combinator parsers return all possible parses according to a grammar, if failure occurs on the nth application of (`many p`), n results will be returned, one for each of the 0 to n–1 successful applications. Following convention, the results are returned in descending order of length. For example, applying the parser `many (literal 'a')` to the string `"aaab"` gives the list

```
[("aaa","b"),("aa","ab"),("a","aab"),("","aaab")]
```

Not surprisingly, the next parser corresponds to the other common iterative form in BNF, defined by $p^+ = p\ p^*$. The parser (`some p`) has the same behaviour as (`many p`), except that it accepts one or more repetitions of `p`, rather of zero or more:

```
some :: parser * ** -> parser * [**]

some p = (p $then many p) $using cons
```

Note that (`some p`) may fail, whereas (`many p`) always succeeds. Using `some` we define parsers for number and words — non–empty sequences of digits and letters:

```
number :: parser char [char]
word   :: parser char [char]

number = some (satisfy digit)
         where digit x = '0' <= x <= '9'

word = some (satisfy letter)
       where letter x = ('a' <= x <= 'z') \/ ('A' <= x <= 'Z')
```

The next combinator is a generalisation of the `literal` primitive, allowing us build parsers which recognise strings of symbols, rather than just single symbols:

```
string :: [*] -> parser * [*]

string []     = succeed []
string (x:xs) = (literal x $then string xs) $using cons
```

For example, applying the parser (`string "begin"`) to the string `"begin end"` gives the output `[("begin"," end")]`. It is important to note that (`string xs`) fails if only a prefix of the sequence `xs` is available in the input string.

As well as being used the define other parsers, the `using` combinator is often used to prune unwanted components from a parse tree. Recall that two parsers composed in sequence produce a pair of results. Sometimes we are only interested in one component of the pair. For example, it is common to throw away reserved words such as "begin" and "where" during parsing. In such cases, two special versions of the `then` combinator are useful, which throw away either the left or right result values, as reflected by the position of the letter "`x`" in their names:

```
xthen :: parser * ** -> parser * *** -> parser * ***
thenx :: parser * ** -> parser * *** -> parser * **

p1 $xthen p2 = (p1 $then p2) $using snd
p1 $thenx p2 = (p1 $then p2) $using fst
```

The actions `fst` and `snd` are the standard projection functions on pairs, defined by `fst (x,y) = x` and `snd (x,y) = y`.

Sometimes we are not interested in the result from a parser at all, only that the parser succeeds. For example, if we find a reserved word during lexical analysis, it may be convenient to return some short representation rather than the string itself. The `return` combinator is useful in such cases. The parser (`p $return v`) has the same behaviour as `p`, except that it returns the value `v` if successful:

```
return :: parser * ** -> *** -> parser * ***

p $return v = p $using (const v)
              where const x y = x
```

2.4 Example

To conclude our introduction to combinator parsing, we will work through the derivation of a simple parser. Suppose we have a program which works with arithmetic expressions, defined in Miranda as follows:

```
expr ::= Num num | expr $Add expr | expr $Sub expr
                 | expr $Mul expr | expr $Div expr
```

We can imagine a function `showexpr` which converts terms of type `expr` to the normal arithmetic notation. For example,

```
showexpr ((Num 3) $Mul ((Num 6) $Add (Num 1))) = "3*(6+1)"
```

While such pretty–printing is notionally quite simple, the inverse operation, parsing, is usually thought of as being much more involved. As we shall see however, building a combinator parser for arithmetic expressions is no more complicated than implementing the `showexpr` function.

Before we start thinking about parsing, we must define a BNF grammar for expressions. To begin with, the definition for the type `expr` may itself be cast in BNF notation. All we need do is include parenthesised expressions as an extra case:

$$\begin{array}{lll} expn & ::= & expn + expn \mid expn - expn \mid \\ & & expn * expn \mid expn/expn \mid \\ & & digit^{+} \mid (expn) \end{array}$$

Although this grammar could be used as the basis of the parser, in practice it is useful to impose a little more structure. To simplify expressions, multiplication and division are normally assumed to have higher precedence than addition and subtraction. For example, $3 + 5 * 2$ is interpreted as $3 + (5 * 2)$. In terms of our grammar, we introduce a new non–terminal for each level of precedence:

$$\begin{array}{lll} expn & ::= & term + term \mid term - term \mid term \\ term & ::= & factor * factor \mid factor/factor \mid factor \\ factor & ::= & digit^{+} \mid (expn) \end{array}$$

While addition and multiplication are clearly associative, division and subtraction are normally assumed to associate to the left. The natural way to express this convention in the grammar is with left recursive production rules (such as $expn ::= expn - term$). Unfortunately, in top–down methods such as combinator parsing, it is well known that left–recursion leads to non–termination of the parser (Aho86). In section 4.1 we show how to transform a grammar to eliminate left–recursion. For the present however, we will leave the grammar as above, and use extra parenthesis to disambiguate expressions involving repeated operations.

Now that we have a grammar for expressions, it is a simple step to build a combinator parser. The BNF description is simply re–written in combinator notation, and augmented with semantic actions to manipulate the result values:

```
expn   = ((term $then literal '+' $xthen term) $using plus)  $alt
         ((term $then literal '-' $xthen term) $using minus) $alt
         term

term   = ((factor $then literal '*' $xthen factor) $using times)  $alt
         ((factor $then literal '/' $xthen factor) $using divide) $alt
         factor

factor = (number $using value) $alt
         (literal '(' $xthen expn $thenx literal ')')
```

Note that the parser makes use of the special sequential combining forms **xthen** and **thenx** to strip non–numeric components from result values. In this way, the arithmetic actions simply take a pair of expressions as their argument. In the definitions given below for the actions, **numval** is the standard Miranda function which converts a string of digits to the corresponding number.

```
value  xs    = Num (numval xs)
plus   (x,y) = x $Add y
minus  (x,y) = x $Sub y
times  (x,y) = x $Mul y
divide (x,y) = x $Div y
```

This completes the parser. For example, `expn "2+(4-1)*3"` gives

```
[( Add (Num 2) (Mul (Sub (Num 4) (Num 1)) (Num 3)) , ""         ),
 ( Add (Num 2) (Sub (Num 4) (Num 1))               , "*3"       ),
 ( Num 2                                           , "+(4-1)*3" )]
```

More than one result is produced because the parser is not forced to consume all the input. As we would expect however, the longest parse is returned first. This behaviour results from careful ordering of the alternatives in the parser.

Although a parse tree is the natural output from a parser, there is no such restriction in combinator parsing. For example, simply by replacing the standard semantic actions with the following set, we have an evaluator for arithmetic expressions.

```
value  xs    = numval xs
plus   (x,y) = x + y
minus  (x,y) = x - y
times  (x,y) = x * y
divide (x,y) = x div y
```

Under this interpretation,

```
expn "2+(4-1)*3" = [(11,""), (5,"*3"), (2,"+(4-1)*3")]
```

3 Layout Conventions

Most programming languages have a set of layout rules, which specify how white–space (spaces, tabs and newlines) may be used to improve readability. In this section we show how two common layout conventions may be handled in combinator parsers.

3.1 Free-format input

At the syntactic level, programs comprise a sequence of tokens. Many languages adopt *free–format input*, imposing few restrictions on the use of white–space — it is not permitted inside tokens, but may be freely inserted between them, although it is only strictly necessary when two tokens would otherwise form a single larger token. White–space is normally stripped out along with comments during a separate lexical phase, in which the source program is divided into its component tokens. This approach is developed in section 4.3.

For many simple parsers however, a separate lexer is not required (as is the case for the arithmetic expression parser of the previous section), but we still might want to allow the use of white–space. The `nibble` combinator provides a simple solution. The parser (`nibble p`) has the same behaviour as the parser `p`, except that it eats up any white–space in the input string before or afterwards:

```
nibble :: parser char * -> parser char *

nibble p = white $xthen p $thenx white
           where white = many (any literal " \t\n")
```

The `any` combinator used in this definition can often be used to simplify parsers involving repeated use of `literal` or `string`. It is defined as follows:

```
any :: (* -> parser ** ***) -> [*] -> parser ** ***

any p = foldr (alt.p) fail
```

The library function `foldr` captures a common pattern of recursion over lists. It takes a list, a binary operator $\otimes$ and a value α, and replaces each constructor "`:`" in the list by $\otimes$, and the empty list `[]` at the end by α. For example, `foldr (+) 0 [1,2,3]` = `1+(2+(3+0))` = `6`. As in this example, α is often chosen to be the right identity for $\otimes$. The infix dot "`.`" used in `any` denotes function composition, defined by `(f.g) x = f (g x)`. It should be clear that `any` has the following behaviour:

```
any p [x1,x2,...,xn] = (p x1) $alt (p x2) $alt ... $alt (p xn)
```

In practice, `nibble` is often used in conjunction with the `string` combinator. The following abbreviation is useful in this case:

```
symbol :: [char] -> parser char [char]

symbol = nibble.string
```

For example, applying the parser (`symbol "hi"`) to the string `"   hi   there"`, gives `("hi","there")` as the first result.

There are two points worth noting about free–format input. First of all, it is good practice to indent programs to reveal their structure. Although free–format input allows us to do this, it does not prevent us doing it wrongly. Secondly, extra symbols are usually needed in programs to guide the parser in determining their structure. Classic examples are "begin", "end" and semi–colon from Pascal.

3.2 The offside rule

Another approach to layout, as adopted by many functional languages, is to constrain the generality of free–format input just enough so that extra symbols to guide the parser are no longer needed. This is normally done by imposing a weak indentation strategy, and having the parser make intelligent use of layout to determine the structure of programs. Consider for example the following program:

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

It is clear from the indention that `a` and `d` are intended to be global definitions, with `b` and `c` local to `a`. The constraint which guarantees that we can always determine the structure of programs in this way is usually given by Landin's *offside rule* (Landin66), defined as follows:

> If a syntactic class obeys the offside rule, every token of an object of the class must lie either directly below, or to the right of its first token. A token which breaks this rule is said to be *offside* with respect to the object, and terminates its parse.

In Miranda, the offside rule is applied to the body of definitions, so that special symbols to separate definitions, or indicate block structuring, are not required. The offside rule does not force a specific way of indenting programs, so we are still free to use our own personal styles. It is worthwhile noting that there are other interpretations of the offside rule. In particular, the proposed standard functional language, Haskell, takes a slightly different approach (Hudak90).

3.3 The offside combinator

In keeping with the spirit of combinator parsing, we would like to define a single combinator which encapsulates the offside rule. Given a parser `p`, we can imagine a parser `offside p` with the same behaviour, except that it is required to consume precisely those symbols which are onside with respect to the first symbol parsed.

At present, parsers only see a suffix of the entire input string, having no knowledge of what has already been consumed by previous parsers. To implement the `offside` combinator however, we need some context information, to decide which symbols in the input are onside. Our approach to this extra information is the key to the `offside` combinator. Rather than actually passing an extra argument to parsers, we will assume that each symbol in the input string has been paired with its row and column position at some stage prior to parsing.

To simplify to types of parsers involving the offside rule, we use the abbreviation (`pos *`) for a symbol of type `*` paired with its position.

```
pos * == (*,(num,num))
```

Since the input string is now assumed to contain the position of each symbol, the primitive parsing function `satisfy` must be changed slightly. As row and column numbers are present only to guide the parser, it is reasonable to have `satisfy` strip this information from consumed symbols. In this manner, the annotations in the input string are of no concern when building parsers, being entirely hidden within the parsing notation itself. The other parsers defined in terms of `satisfy` need a minor change to their types, but otherwise remain the same.

```
satisfy :: (* -> bool) -> parser (pos *) *

satisfy p []     = fail []
satisfy p (x:xs) = succeed a xs , p a
                 = fail xs      , otherwise
                   where (a,(r,c)) = x
```

We are now able to define the `offside` combinator. The only complication is that white–space must be treated as a special case, in never being offside. To avoid this problem, we assume that white–space has been stripped from the input prior to parsing. No layout information is lost, since each symbol in the input is paired with its position. In reality, most parsers will have a separate lexical phase anyway, in which both comments and white–space are stripped.

```
offside :: parser (pos *) ** -> parser (pos *) **

offside p inp = [(v,inpOFF) | (v,[]) <- p inpON]
               where
                  inpON  = takewhile (onside (hd inp)) inp
                  inpOFF = drop (#inpON) inp
                  onside (a,(r,c)) (b,(r',c')) = r'>=r & c'>=c
```

The offside rule tells us that for the parser (`offside p`) to succeed, it must consume precisely the onside symbols in the input string. As such, in the definition above it is sufficient to apply the parser `p` only to the longest onside prefix (`inpON`). The pattern `(v,[])` in the list comprehension filters out parses which do not consume all such symbols. For successful parses, we simply return the result value `v`, and remaining portion of the input string (`inpOFF`). It is interesting to note that the `offside` combinator does not depend upon the structure of the symbols in the input, only that they are paired with their position. For example, it is irrelevant whether symbols are single characters or complete tokens.

For completeness, we briefly explain the four standard Miranda functions used in `offside`. Given a list, the function (`takewhile p`) returns the longest prefix in which predicate `p` holds of each element. The function `hd` selects the first element of a list, and is defined by `hd (x:xs) = x`. The function (`drop n`) retains all but the first `n` elements of a list. Finally, "`#`" is the length operator for lists.

4 Building Realistic Parsers

Many simple grammars can be parsed in a single phase, but most programming languages need two distinct parsing phases — lexical and syntactic analysis. Since lexical analysis is nothing more than a simple form of parsing, it is not surprising to find that lexers themselves may be built as combinator parsers. In this section we work through an extended example, which shows how to build two–phase combinator parsers, and demonstrates the use of the `offside` combinator.

4.1 Example language

We develop a parser for a small programming language, similar in form to Miranda. The following program shows all the syntactic features we are considering:

```
f x y = add a b
        where
           a = 25
           b = sub x y
answer = mult (f 3 7) 5
```

If a program is well–formed, the parser should produce a parse tree of type `script`, as defined below. Even though local definitions are attached to definitions in the language, it is normal to have them at the expression level in the parse tree.

```
script ::= Script [def]
def    ::= Def var [var] expn
expn   ::= Var var | Num num | expn $Apply expn | expn $Where [def]

var == [char]
```

The context–free aspects of the syntax are captured by the BNF grammar below. The non–terminals *var* and *num* correspond to variables and numbers, defined in the usual way. Ambiguity is resolved by the offside rule, applied to the body of definitions to avoid special symbols to separate definitions and delimit scope.

$$
\begin{array}{lcl}
prog & ::= & defn^* \\
defn & ::= & var^+ \text{ ``=" } body \\
body & ::= & expr \text{ [``}\texttt{where}\text{" } defn^+\text{]} \\
expr & ::= & expr\ prim \mid prim \\
prim & ::= & var \mid num \mid \text{``("}\, expr \text{``)"}
\end{array}
$$

As we would expect, application associating to the left in our language is expressed by a left–recursive production rule in the grammar (*expr*). As already mentioned in section 2.4 however, left–recursion and top–down parsing methods do not mix. If we are to build a combinator parser for this grammar, we must first eliminate the left–recursion. Consider the left–recursive production rule

$$\alpha \quad ::= \quad \alpha\beta \mid \gamma$$

in which it is assumed that γ does not begin with an α. The assumption ensures that the production has a non–recursive *base case*. (For the more general situation when there is more than one recursive production for α, the reader is referred to (Aho86).) What language is generated by α? Unwinding the recursion a few times, it is clear that a single γ, followed by any number of βs is acceptable. Thus, we would assert that $\alpha ::= \gamma\beta^*$ is equivalent to $\alpha ::= \alpha\beta \mid \gamma$. The proof is simple:

$$\begin{array}{lcll}
\gamma\beta^* & = & \gamma\ (\beta^*\beta \mid \varepsilon) & \{ \text{ properties of } * \ \} \\
& = & \gamma\beta^*\beta \mid \gamma\varepsilon & \{ \text{ distributivity } \} \\
& = & (\gamma\beta^*)\ \beta \mid \gamma & \{ \text{ properties of sequencing } \} \\
& = & \alpha\beta \mid \gamma & \{ \text{ definition of } \alpha \ \}
\end{array}$$

In our example language, this allows us to replace the left–recursive *expr* production rule with $expr ::= prim\ prim^*$, which in turn simplifies to $expr ::= prim^+$. While the languages accepted by the left–recursive and iterative production rules are provably equivalent, the parse trees will in fact be different. This problem can fixed by a simple *action* in the parser; we return to this point at the end of section 4.5.

4.2 Layout analysis

Recall that the `offside` combinator assumes white–space in the input is replaced by row and column annotations on the symbols. To this end, each character is paired with its position during a simple layout phase prior to lexical analysis. White–space itself will be stripped by the lexer, as is normal practice.

```
prelex = pl (0,0)
         where
            pl (r,c) []     = []
            pl (r,c) (x:xs) = (x,(r,c)) : pl (r,tab c) xs , x = '\t'
                            = (x,(r,c)) : pl  (r+1,0)  xs , x = '\n'
                            = (x,(r,c)) : pl  (r,c+1)  xs , otherwise
            tab c = ((c div 8)+1)*8
```

4.3 Lexical analysis

The primary function of lexical analysis is to divide the input string into its component tokens. In our context, each token comprises a tag, and a string. Two strings have the same tag only if they may be treated as equal during syntax analysis.

```
token == (tag,[char])
```

For example, we could imagine `(Ident,"add")` and `(Lpar,"(")` as tokens corresponding to the strings `"add"` and `"("`. According to the last sentence of the previous paragraph, each reserved word or symbol such requires a unique tag. To avoid this tedium, we choose to bundle them together as tokens with the tag `Symbol`:

```
tag ::= Ident | Number | Symbol | Junk
```

For example, the tokens `(Number,"123")` and `(Symbol,"where")` correspond to the strings `"123"` and `"where"`. The special tag `Junk` is used for things like white–space and comments, which are required to be stripped before syntax analysis.

Like all other parsers, lexers will ultimately be defined in terms of the primitive parsing function `satisfy`. Earlier we decided that this was a good place to throw away the position of consumed symbols. Now we actually need some of this information, since the `offside` combinator requires each token to be paired with its position. Our solution is to define a new combinator, `tok`, which encapsulates the process of pairing a token with its position. Since `tok` will be applied once to each parser for complete tokens, it is convenient to include the tag as an extra parameter to `tok`. We see then that `tok` provides a means to change a parser with result type `[char]` into a parser with result type `(pos token)`.

```
tok :: parser (pos char) [char] -> tag -> parser (pos char) (pos token)

(p $tok t) inp = [(((t,xs),(r,c)),out) | (xs,out) <- p inp]
                 where (x,(r,c)) = hd inp
```

For example, `(string "where" $tok Symbol)` is a parser which produces the pair `((Symbol,"where"),(r,c))` as its first result if successful, where `(r,c)` is the position of the "`w`" character in the input string. Notice that `tok` may fail with parsers which admit the empty string, in trying to select the position of the first character when none of the input string is left. It is reasonable to ignore this problem however, since to guarantee termination of the lexer, the empty–string must not be admissible as a token.

We turn our attention now to lexical analysis itself. Thinking for a moment about what the lexer actually does, it should be clear that the general structure is as follows, where each `p`*i* is a parser, and `t`*i* a tag.

```
many ((p1 $tok t1) $alt (p2 $tok t2) $alt ... $alt (pn $tok tn))
```

We find it convenient then to define a combinator which builds parsers of this form. Given a list `[(p1,t1),(p2,t2),...]` of parsers and tags, the `lex` combinator builds a lexer as above.

```
lex :: [(parser (pos char) [char],tag)] -> parser (pos char) [pos token]

lex = many.(foldr op fail)
      where (p,t) $op xs = (p $tok t) $alt xs
```

The standard functions "`.`" and `foldr` were explained in section 3.1. Using `lex`, we now define a lexer for our language.

```
lexer :: parser (pos char) [pos token]

lexer = lex [( some (any literal " \t\n") , Junk   ),
```

```
( string "where"              , Symbol ),
( word                        , Ident  ),
( number                      , Number ),
( any string ["(",")","="]    , Symbol )]
```

A secondary function of a lexer is to resolve lexical conflicts. There are basically two kinds. First of all, lexical classes may overlap. For example, reserved words are usually also admissible as identifiers. Secondly, some strings may be interpreted as different numbers of tokens. For example, `">="` could be seen either as a representation of the operator "$\geq$", or as the separate operators "$>$" and "$=$".

In our lexer, there is only one such conflict, the reserved word `"where"`. We arrange for the correct interpretation by ordering the tokens according to their relative priorities. In this case for example, reserved words appear before identifiers in the lexer. Ordering the remaining, non–conflicting, tokens by probability of occurrence can considerably improve the performance of the lexer.

4.4 Scanning

Since there is no natural identity element for the list constructor "`:`" used by `many` to build up the list of tokens, white–space and comments are not removed by the lexer itself, but tagged as junk to be removed afterwards. The `strip` function takes the output from a lexer, and removes all tokens with `Junk` as their tag:

```
strip :: [pos token] -> [pos token]

strip = filter ((~=Junk).fst.fst)
```

The standard function (`filter p`) retains only those elements of a list which satisfy the predicate `p`, and is defined by `filter p xs = [x | x <- xs ; p x]`. For example, applying `filter (>5)` to the list `[1,6,2,7]` gives the list `[6,7]`.

4.5 Syntax analysis

Lexical analysis makes the initial jump from characters to tokens. Syntax analysis completes the parsing process, by combining tokens to form a parse tree. For most tokens, only the tag part is important during syntax analysis. Thus we define (`kind t`) as a parser which recognises any token with tag `t`, regardless of its string part. Once a token has been consumed by a parser, its tag becomes somewhat redundant, in much the same way as its position becomes redundant after being consumed by the `satisfy` primitive. To this end, (`kind t`) returns only the string part of a consumed token:

```
kind :: tag -> parser (pos token) [char]

kind t = (satisfy ((=t).fst)) $using snd
```

Because all reserved words and symbols share the single tag `Symbol`, the `kind` function is no use in these cases. We need a special function which matches on the string part of a token. Thus, we define (`lit xs`) as a parser which only admits the token (`Symbol,xs`). As for `kind`, the tag part of a consumed token is discarded:

```
lit :: [char] -> parser (pos token) [char]

lit xs = literal (Symbol,xs) $using snd
```

Recall now the BNF grammar for our example language.

$$
\begin{array}{lcl}
prog & ::= & defn^* \\
defn & ::= & var^+ \text{ “=” } body \\
body & ::= & expr \; [\text{“}\mathtt{where}\text{” } defn^+] \\
expr & ::= & prim^+ \\
prim & ::= & var \mid num \mid \text{“(”}\, expr \,\text{“)”}
\end{array}
$$

Just as in the arithmetic expression example of section 2.4, we build a parser by simply casting the grammar in combinator notation, and including semantic actions to build the parse tree:

```
prog = many defn $using Script
defn = (some (kind Ident) $then lit "=" $xthen offside body) $using defnFN
body = (expr $then ((lit "where" $xthen some defn) $opt [])) $using bodyFN
expr = some prim $using (foldl1 Apply)
prim = (kind Ident  $using Var)   $alt
       (kind Number $using numFN) $alt
       (lit "(" $xthen expr $thenx lit ")")
```

Recall that the offside rule is applied to the body of definitions in our example language. In direct correspondence, see that the `offside` combinator is applied to `body` in the `defn` parser above. The `opt` combinator used in the definition of `body` above corresponds to the $[\cdots]$ notation in BNF, denoting an optional phrase:

```
opt :: parser * ** -> ** -> parser * **

p $opt v = p $alt (succeed v)
```

Before defining the remaining semantic actions, the somewhat strange action (`foldl1 Apply`) in the `expr` parser merits some explanation. Recall that the original grammar in section 4.1 used left–recursion to express the left associativity of application. By applying a simple transformation, left recursion was eliminated in favour of iteration. In combinator parsing, iteration corresponds to `many` and `some`. These operators produce a list as their result. What we really want from the `expr` parser is a left–recursive application spine: if the result were the list `[x1,x2,x3,x4]`, it should be transformed to (((`x1 @ x2`) `@ x3`) `@ x4`), where `@` denotes the application constructor `$Apply`. To do this, we use a directed reduction as for the `any` combinator in section 3.1, except that this time the operator should be bracketed to the left instead of the right. That is, `foldl` should be used instead of `foldr`. In fact we use `foldl1`, which is precisely the same, except that it only works with non–empty lists, and hence we don't need to supply a base case.

Of the three remaining semantic actions, the first two are straightforward, simply converting results to the appropriate types. The final action takes into account that local declarations are found at the expression level in the parse tree, while they are attached to definitions in the grammar.

```
defnFN (f:xs,e) = Def f xs e
numFN  xs       = Num (numval xs)

bodyFN (e,[])   = e
bodyFN (e,d:ds) = e $Where (d:ds)
```

4.6 The complete parser

The complete parser is obtained by simply composing four functions — `prelex` (pairing symbols with their position), `lexer` (lexical analysis), `strip` (removing white–space and comments), and `prog` (syntax analysis). We ignore the possibility of errors, assuming that the lexical and syntactic analysis are always successful. The function (`fst.hd`) selects the first result from lexical and syntactic phases.

```
parse :: [char] -> script

parse = fst.hd.prog.strip.fst.hd.lexer.prelex
```

5 More combining forms

We conclude our introduction to combinator parsing by presenting a few extra combining forms that have proved useful, allowing us to make parsers more lazy, give more informative error messages, and manipulate result values in some new ways.

5.1 Improving laziness

While combinator parsers are simple to build, some such parsers are not as lazy as we would expect. Recall the `many` combining form from section 2.3. For example, applying the parser `many (literal 'a')` to the string `"aaab"` gives the list

```
[("aaa","b"),("aa","ab"),("a","aab"),("","aaab")]
```

Since Miranda is lazy, we would expect the `a`'s in the first result to become available one at a time, as they are parsed in the input string. This is not however what happens. In practice the string `"aaa"` is not made available until it has been entirely constructed. The implication is that parsers defined using `many` at the top level, such as lexers, cannot rely on lazy evaluation to produce components of the result lists on a supply and demand basis. We refer the reader to (Wadler85) for a more detailed explanation of the laziness problem with `many`; Wadler's solution is a new combinator which guarantees that a parser succeeds at least once.

5.2 Limiting success

Combinator parsers as presented in this article return a list of results if successful. Being able to return more than one result allows us to build parsers for ambiguous grammars, with all possible parses being produced for an ambiguous input string. Natural languages are commonly ambiguous. Programming languages are for the most part completely unambiguous; at most one parse of any input string is possible. When working with un–ambiguous grammars, it may be preferable to use a special type for failure/success of a parser, rather than returning a list of results.

```
maybe * ::= Fail | OK *

parser * ** == [*] -> maybe (**,[*])
```

Redefining the primitive parsers and combining forms is straightforward.

5.3 Error reporting

A simple extension of the `maybe` type above can be used to good effect in reporting errors during parsing. If a combinator parser is applied to an input string containing an error, the result will often be outright failure to parse the input. Sometimes however a prefix of the input may be parsed successfully, in which case the unconsumed suffix of the input is returned as part of the result from the parser. Using the unconsumed input to produce an error message is likely to be uninformative; the position in the input where the longest parse ends may be far away from the error. The problem can be solved by distinguishing between failure and an error during parsing; in both cases we return a message giving the reason for an unsuccessful parse:

```
maybe * ::= Fail [char] | Error [char] | OK *
```

Redefining the primitive parsers and combining forms is again straightforward: `Fail` and `Error` values should be treated identically, except that in the definition of the `alt` combining form, the second parser may be applied if the first parser fails, but not if it produces an error. Error values are created using `nofail`:

```
nofail :: parser * ** -> parser * **

(nofail p) inp = f (p inp)
                 where
                    f (Fail xs) = Error xs
                    f other     = other
```

The parser (`nofail p`) has the same behaviour as the parser `p`, except that failure of `p` gives rise to an error. Two common ways in which `nofail` is used are (`p $then nofail q`) and (`p $alt q $alt nofail r`). In the first case, failure of parser `q` after success of parser `p` gives an error rather than just failure. In the second case, failure of any alternative to succeed gives rise to an error. Experience has shown that careful use of `nofail` can result in reasonably informative error reporting.

5.4 Result values

In parsers built using the `then` combining form, the right–hand parser has no access to the result produced by the left–hand parser; the results produced by the two parsers are paired within `then`. Sometimes it is useful to have a parser take not just a sequence of symbols as input, but also the result from some other parser. A new combining form proves very useful in building such parsers:

```
into :: parser * ** -> (** -> parser * ***) -> parser * ***

(p $into f) inp = g (p inp)
                  where
                     g (OK (v,inp')) = f v inp'
                     g other         = other
```

We assume for convenience now that Miranda is extended with λ–expressions, written as `\v.e`. The parser `(p $into \v.q)` accepts the same strings as the parser `(p $then q)`, but the treatment of result values is different: if parser `p` is successful, its result value is bound to variable `v`, and is thus avialable to parser `q`; if parser `q` is in turn successful, the result from the composite parser `(p $into \v.q)` is the result of parser `q`. Contrast with the parser `(p $then q)`, whose result is the pair of results from parsers `p` and `q`. There are many interesting and useful applications of the `into` combining form. For example, it can be used to define `using` and `then`:

```
p $using f = p $into \v. succeed (f v)

p $then q = p $into \v. q $using \w.(v,w)
```

Another application: imagine a parser of the following form.

```
((p $then q) $using f) $alt ((p $then r) $using g)
```

If on the left–side of the `alt` the parser `p` is successful but the parser `q` fails, then on the right–side of the `alt` the parser `p` will be re–applied to the same input string. This is clearly inefficient. The standard solution is to *factorise* out `p`, giving a parser of the form `(p $then (q $alt r)) $using h`. The new action `h` is some combination of actions `f` and `g`. A common application of such a grammar transformation is with language constructs which have an optional component. Examples of such constructs are "if" expressions, having an optional "else" part, and definitions in Miranda, having an optional "where" part. A problem with the above transformation is that the action `h` requires some means of telling which of parsers `q` and `r` was successful, to decide which of actions `f` and `g` should be applied to the result. This may necessitate parsers `q` and `r` having to encode their result values in some way. A much cleaner treatment of the actions after factorisation is to make the result of parser `p` available to the parser `(q $alt r)` using the `into` combining form:

```
p $into \v. ((q $using f v) $alt (r $using g v))
```

In the original parser, the actions `f` and `g` took a pair of results as their argument; in the parser above, the actions must be curried to take their arguments one at a time.

Another application of `into`: parsing infix operators that associative to the left. Consider a parser of the form `some p`. Such a parser produces lists as its result values. Lists are an example of a right–recursive structure; a list is either empty, or comprises a value and another list. Suppose we wanted a parser that admitted the same strings as `some p`, with the results being returned in the same order, but in a left–recursive rather than a right–recursive structure. Such a parser is

```
some p $using foldl1 f
```

where `f` is some left–recursive binary constructor. The use of `foldl1` above was explained towards the end of section 4.5. A drawback of this approach is the building of the intermediate list prior to applying the `foldl1` operator. The need for such an intermediate structure can be avoided by rewriting the parser using `into`, accumulating a left–recursive structure as the input is parsed:

```
p $into manyp
where manyp v = (p $into \w. manyp (f v w)) $alt succeed v
```

We conclude by noting an interesting relationship between `into` and the Categorical notion of a *monad*. Combinator parsers give rise to a monad; we refer the reader to (Wadler90) for a full explanation. In this context, the `into` combining form is very closely related to the composition operator in the *Kleisli* category induced by the monad of parsers. (The identity operator is `succeed`.) Being precise, the composition operator is defined as follows.

```
(p $compose q) v = p v $into q
```

An equivalent definition is:

```
(p $compose q) v = succeed v $into p $into q
```

Acknowledgements

Thanks to Paul Hudak and John Launchbury for their comments and suggestions.

References

Aho, A., Sethi, R., Ullman, J., (1986), Compilers – Principles, Techniques and Tools, Addison–Wesley.

Fairbairn, J., (1986), Making Form Follow Function, Technical Report 89, University of Cambridge Computer Laboratory, June 1986.

Frost, R., Launchbury, J., (1988), Constructing Natural Language Interpreters in Lazy Functional Languages, Glasgow University.

Hudak, P., Wadler, P. (editors), (1990), Report on the Programming Language Haskell, Glasgow University and Yale University.

Johnsson, T., (1987), Attribute Grammars as a Functional Programming Paradigm, FPCA 87, LNCS 274.

Landin, P., (1966), The Next 700 Programming Languages, CACM Vol. 9, March 1966.
Wadler, P., (1985), How to Replace Failure by a List of Successes, FPCA 85, LNCS 201.
Wadler, P., (1990), Comprehending Monads, FPCA 90.