Parsing Using Combinators

Graham Hutton
Department of Computing Science
University of Glasgow
Scotland, UK

Abstract

In combinator parsing, the text of parsers resembles BNF notation. We present the basic method, and show how it may be extended to more practical parsers. We address the special problems of layout, and parsers with separate lexical and syntactic phases. In particular, an elegant new way of handling the offside rule is given. Many examples and comments are included.

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 as input, 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 [1].

Although there are many methods of parsing, one in particular has gained widespread acceptance for use in lazy functional languages. In *combinator parsing*, parsers are modelled directly as functions. Working in a higher-order language, it is quite natural to consider higher-order functions which combine or transform parsers. What may be surprising however, is that this approach leads to a parsing notation in which the text of parsers closely resembles BNF notation itself. In this manner, parsers are quick to build, and simple to understand and modify.

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. Arbitrary semantic actions may be added to parsers, allowing their results to be manipulated in any way we please. For example, we could imagine generating some form of abstract machine code as programs are parsed.

Although the underlying principles are widely known, dating back to [2], little has been written on combinator parsing itself. In this paper we present the basic method, and show how it may be extended to more practical parsers. The first section collects all the basic material, and includes many examples and comments. In the remainder of the paper, we address the special problems of layout, and parsers with separate lexical and syntactic phases. In particular, we present an elegant new

K. Davis et al. (eds.), *Functional Programming*

way of handling the offside rule. After working through this paper, the reader should have sufficient knowledge to build a parser for a language like Miranda[1].

The techniques we present may be used in any lazy functional language with a higher-order/polymorphic style type system. However, all our programming examples are given in Miranda. The particular features and library functions of Miranda are explained as they are used. A library of parsing functions taken from this paper may be obtained by e-mail from the author, at ⟨graham@cs.glasgow.ac.uk⟩.

2 Parsing Using Combinators

A parser may be viewed as a function from a string of symbols to a result value. To be able to combine parsers in sequence, we also need to know how much of the input was used in producing the result. Thus, combinator parsers return a pair, comprising a result value and the unused suffix of the input string. Sometimes a parser may not be able to produce a result. For example, it may be looking for a letter, but find a digit. A parser produces a result only if there is successful recognition of the input string as a sentence of the grammar. In such cases the parser is said to have *succeeded*, otherwise it has *failed*.

To help us see how parsers fit together, it is useful to consider their types. From what has been seen already, the basic type is $[symbol] \rightarrow (value, [symbol])$. However, we must also take into account that a parser may fail. Rather than defining a new algebraic 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 indicating success. Although this representation may seem a little strange at the moment, all will become clear later on.

Since we want to specify the type of any parser, regardless of the kind of symbols and values it works with, these types are included as extra parameters. In Miranda, such free type variables are indicated by stars.

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

For example, a parser for arithmetic expressions may 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), whose sole 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 does not depend upon the input, its result value must be completely pre-determined, so is included as an extra parameter to the parser.

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

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

[1]Miranda is a trademark of Research Software Limited.

This definition relies on partial application to work properly. The order of the arguments means that if succeed is only applied to its first argument, the result is a parser (i.e. a function) which always succeeds with this value. For example, (succeed 5) is a parser which always returns the value 5. Furthermore, even though succeed plainly has two arguments, its type would suggest that it has only one. There is no magic, the second argument has simply moved inside the type of the result, as would be clear upon expansion according to the `parser` synonym.

The second primitive parser is in some sense the opposite of the first, in that it always fails, regardless of the input.

```
fail :: parser * **

fail inp = []
```

The final primitive parser we define allows us to recognise individual symbols in the input string. Rather than enumerating the acceptable symbols, it is usually more convenient to provide the set implicitly, via a predicate which determines if an arbitrary symbol is a member. Not surprisingly, 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
```

This parser may be regarded as *the* primitive or canonical parser, through which all others factor. In simpler words, all other parsers will ultimately be defined in terms of `satisfy`. Notice how succeed and `fail` are used in this example. Although they are not strictly necessary, their presence makes the parser easier to read. It is also important to note that `satisfy` trivially fails if there is no input.

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 input string "345" gives the result `[('3',"45")]`. In the definition of `literal`, `(=x)` is an example of an *operator section*, in this case denoting the function which compares its argument with `x`. Sectioning is a useful syntactic convention which enables us to partially apply infix operators. It is explained in more detail in the Miranda manual.

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 combinator parsers resemble BNF notation, we define two

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 *combinators*[2] for short. We will use this term 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 [3]. In combinator parsing however, we use inclusive *either*. That is, it is acceptable for both parsers to succeed, in which case we return both results. In other words, instead of restricting parsers to a single result, we allow an arbitrary number. This explains why we decided to have a parser return a list of results.

Under this interpretation, `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 and define this combinator as an infix operator. Just as for sectioning, the infix notation is merely a syntactic convenience. In particular, (`x $f y`) is equivalent to (`f x y`) in all contexts.

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

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

The reader may wish to verify that failure is the identity element for alternation. That is, (`fail $alt p`) = (`p $alt fail`) = `p`. In practical terms this means that `alt` has the expected behaviour if only one of the parsers succeeds.

There are two interesting implications in allowing parsers to produce more than one result. Perhaps the most obvious is that we can handle ambiguous grammars, with all possible parses being produced if so required. The feature has proved particularly useful in natural language processing [4]. More often that not however, we are only interested in the single longest parse of an input 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. Although this may sound quite difficult, in practice it usually happens quite naturally without much special planning.

The second advantage is equally interesting. Under lazy evaluation, returning a list of results automatically gives us an equivalent to backtracking [2]. That is, even though a particular parse may fail, we do not necessarily need to start again. Some of the work in getting to failure may be re-used in trying the next parse. We need only go back to last `alt` where we haven't yet tried the second branch. In practical terms, this behaviour arises from the depth first traversal inherent in all top-down methods such as combinator parsing.

The `then` combinator corresponds to sequencing in BNF. The parser (`p1 $then p2`) recognises anything that both `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 of the parsers. They are combined (by pairing) to form a single result for each parse.

[2]Our informal use of this term is consistent with the standard λ-calculus meaning.

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

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

This 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. Combinator parsing is not dependant upon list comprehension notation, but much of the elegance will be lost if it is not available.

Since alternation and sequencing are the primitive combining forms, it is important to consider how they interact with themselves, and each other. Firstly, it is easy to verify that alternation is associative. That is, `(p $alt q) $alt r` = `p $alt (q $alt r)`. In practice this means we do not not need to worry about bracketing repeated alternation correctly. On the other hand, sequencing is not associative, due to the tupling of results from the component parsers. In Miranda, all infix operators defined 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)`.

When they appear together in the same expression, sequencing is normally assumed to have higher precedence than alternation. Unfortunately, in Miranda all user defined infix operators are deemed to have the same precedence. This means that extra bracketing will sometimes be needed to disambiguate expressions involving both `alt` and `then`. Even if such declarations are permitted in the language, it is usual to take a new line for each alternative in a lengthy expression.

2.3 Manipulating values

Part of the result from a combinator parser is a value. When two parsers are combined in sequence we get a pair of values. So that we are not limited to binary parse trees, we need some way of manipulating the default result values from parsers. This is the purpose of the `using` combinator. The parser `(p $using f)` has the same behaviour as `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 this combinator has no counterpart in pure BNF notation, it does have much in common with the $\{\cdots\}$ operator in YACC notation [1]. In fact, the `using` combinator does not restrict use to building parse trees. Arbitrary semantic actions can be used, allowing values to be manipulated in any way we please. For example, in section 2.4 we show how to make an evaluator for arithmetic expressions from a parser for expressions, simply by changing the actions. In the remainder of this section we define some useful parsers and combinators.

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 (:), 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 input "aaab" gives the output

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

Not surprisingly, the next parser corresponds to the other common iterative form, denoted by + in BNF notation. The parser (some p) has the same behaviour as (many p), except that it recognises one or more repetitions instead of zero or more. Note that (some p) may fail, whereas (many p) always succeeds.

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

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

Using this combinator we can build parsers for numbers and words. A number is simply a sequence of digits, and a word a sequence of 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 in section 2.1, allowing us to recognise specific strings, instead of single symbols. The parser (`string xs`) recognises the complete sequence of symbols `xs`. In particular, the parser fails if only a prefix of the list is available.

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

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

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 of these components. 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 sequential combinator are useful, which throw away either the left or right result values, as indicated by the position of the extra letter 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`. The two abbreviations `xthen` and `thenx` throw away selected components from a compound result. Sometimes we are not interested in the result values at all, only that the parser actually 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. Our example is adapted from [5]. Suppose we have some programs which manipulate arithmetic expressions, defined in Miranda as follows:

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

While this representation is reasonable from a programming point of view, even small expressions are difficult to understand in this form. The obvious solution is to define a *pretty-printer* which converts expressions to a more readable form for display. For example, it is easy to define a function `showexpr` such that

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

If expressions are to be viewed in short notation, it would seem equally reasonable that they should also be supplied in this form. While pretty-printing is notionally quite simple, the inverse operation of parsing is usually thought of as being much more involved. As we shall see however, a combinator parser for arithmetic expressions is no more complicated than a simple pretty-printer.

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

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

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 grammatical terms, this means introducing 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 sometimes 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 [1]. In section 4.1 we show how to transform a grammar to eliminate left recursion. For the present however, we will leave the grammar alone, and use 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. In fact, it is convenient to remove all non-numeric symbols within the parser itself, using the special sequential forms `xthen` and `thenx`.

```
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 ')')
```

All that remains is to define the semantic actions which build the parse tree. Since the parser has already stripped the non-numeric components, the arithmetic actions simply take a pair as their argument. In the following, `numval` is the standard Miranda function which converts a numeric string 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 the 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")]
```

We could imagine many other useful interpretations. For example, in [5] actions are given which transform arithmetic expressions to postfix (reverse-polish) notation, for evaluation on a stack-based machine.

3 Layout Conventions

Unlike the arithmetic expression example of the previous section, most programming languages have a fairly large and involved syntax. To make life a little more pleasant for the programmer, there is usually a set of layout rules which specify how *white-space* (spaces, tabs and newlines) may be used to improve readability. In this section we consider two common layout conventions, and show how they may be interpreted in combinator parsing.

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 program to be parsed it divided into its component tokens. This approach is developed in section 4.3.

For many simple parsers however, a separate lexer is not actually needed (as is the case for the arithmetic expression parser of the previous section), but we still might want to use white-space to make things easier to read. The `nibble` combinator provides a simple solution. The parser (`nibble p`) has the same behaviour as `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. Often α is chosen to be the right identity for $\otimes$. For example, `foldr (+) 0 [1,2,3]` = `1+(2+(3+0))` = 6. The infix dot (`.`) 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)
```

If we now take a simple parser, and place a single `nibble` wherever we would like to allow white-space, we have an analogue to free-format input, without altering the structure of the parser at all. In practice, `nibble` is most often used with reserved words and symbols, so that following abbreviation is useful.

```
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. The classic examples are "begin", "end" and semi-colon from Pascal. Experience has shown that such symbols can be a nuisance to the accomplished programmer, and a source of confusion to beginners.

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. For example, consider the following program:

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

It is plainly obvious from the indention that a and d are intended to be top-level definitions, with b and c local to a. The constraint which guarantees that we can always determine the structure of programs in this manner is usually given by Landin's *offside rule* [6], 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 extra symbols are not needed to separate definitions, or indicate block structuring. The offside 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. For example, the languages Occam [7] and Haskell [8] both take a slightly different approach.

3.3 The offside combinator

Inkeeping with the spirit of combinator parsing, we would like to define a new combinator which encapsulates the offside rule, and hides all the messy details. Given a parser, the `offside` combinator should transform it to reflect the additional constraints imposed by the offside rule. That is, the parser (`offside p`) should only succeed if p consumes precisely those symbols which are onside with respect to the first symbol in the input string.

Although the `offside` combinator is easy to imagine in principle, it is perhaps not so obvious how it should be implemented. The problem is that 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 rule we need to know the context of the input, to decide which of the following symbols are onside. Our solution to this problem is the key to the `offside` combinator. Rather than actually passing an extra argument to parsers, we will simply assume that each symbol in the input string has been paired with its row/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/column numbers are present only to guide the parser, it is reasonable to have `satisfy` throw this information away 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, since it is never offside. In fact, the white-space in the input is somewhat redundant, since having symbols paired with their position may be viewed as an extreme form of layout. It is therefore both convenient and reasonable to assume that white-space has been stripped from the input prior to parsing. Most parsers will have a separate lexical phase anyway, in which both comments and white-space are removed, so this is no great problem.

```
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
```

This combinator is quite unlike any of the others, so worth some explanation. The offside rule tells us that for `offside p` to succeed, it must consume precisely the onside symbols in the input string, so it is sufficient to only apply `p` 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 `offside` 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, the four standard Miranda functions used in the definition of `offside` are as follows. The function (`takewhile p`) returns the longest prefix of a list, in which property `p` holds of each element. The function `hd` returns the first element of a list, defined by `hd (x:xs) = x`. The function (`drop n`) removes the first `n` elements from the front of a list. Finally, (`#`) denotes the length of a list.

4 Building Complete 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 that lexers themselves may be built as combinator parsers. In this section we work through an extended example, showing that by careful choice of notation, two-phase parsers are no more complicated to build than one-phase parsers.

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, according to the following type. Even though local definitions are attached to definitions in our 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 may be characterised by the following BNF grammar. Ambiguity is resolved by the offside rule, which is applied to the body of definitions, so that special symbols to separate definitions and delimit scope are not needed. The non-terminals *var* and *num* denote variables and numbers respectively, defined in the usual way.

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

As we would expect, the left associativity of function application is expressed with a left recursive production rule (i.e. $expr ::= expr\ prim \mid prim$). As already mentioned in section 2.4, 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 canonical left recursive production rule:

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

What language is generated by α? By 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:

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

In the context of our example language, this law means that we may safely replace the *expr* production rule with $expr ::= prim\ prim^*$, which simplifies to $expr ::= prim^+$. In other words, by applying a simple transformation to the grammar, we have eliminated left recursion in favour of iteration.

4.2 Layout analysis

Recall that the `offside` combinator assumes that white-space in the input is replaced by row/column annotations. To this end, each character is paired with its position during a simple layout phase prior to lexical analysis. Both white-space and comments will be stripped by the lexer itself, 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. Each token is made up of two separate parts – a *tag* and an *attribute value*. Two strings only have the same tag if they may be treated as equal during syntax analysis. When more than one string matches a token, sufficient information to tell them apart is returned as the attribute value of the token. In practice, it is convenient to return the string itself. Thus,

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

If a token is uniquely determined by its tag, the empty-string should be used as its attribute value. For example, we could imagine `(Ident,"add")` and `(Lpar,"")` as tokens corresponding to the strings `"add"` and `"("`. While it is fine to refer to identifiers and numbers by their tag during syntax analysis, having to invent and use tags for reserved words and symbols is somewhat tedious. For this reason, rather than having separate tags for each reserved word and symbol, we will bundle them all together under the single tag `Symbol`, with the string itself as the attribute value:

```
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 should 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 all consumed symbols. Now we actually need some of this information, since tokens must be paired with their position. Rather than changing `satisfy` again, and having row/column numbers visible everywhere, we define a new combinator which encapsulates the process of pairing a token with its position. Since it will be applied once to each parser corresponding to a complete token, it is a convenient place in which to tag each token as well.

The `tok` combinator takes a parser and a tag, and modifies the parser so that strings are built into tokens, and paired with the position of their first character. In other words, the `tok` combinator changes 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 result `((Symbol,"where"),(r,c))` 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 tokens are always at least one character in length, to guarantee termination of the lexer.

We now turn our attention to lexical analysis itself. Thinking for a moment about what the lexer actually does, it should be clear that the general structure will be 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))
```

This structure will remain the same for all lexers. All that will change is the parsers and tags. It is therefore convenient to wrap the general structure up inside a definition which takes care of all the messy details. Given a list of parsers and tags `[(p1,t2),(p2,t2),...]`, the `lex` combinator builds a lexical analyser.

```
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 the `tok` abbreviation, we may define the lexer for our language as follows:

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

lexer = lex [( some (any literal " \t\n") , Junk   ),
            ( string "where"              , Symbol ),
            ( word                        , Ident  ),
            ( number                      , Number ),
            ( any string ["(",")","="]    , Symbol )]
```

One of the secondary functions of the lexer is to resolve lexical conflicts. There are basically two kinds. First of all, classes may overlap. For example, reserved words are usually also acceptable as identifiers. Secondly, some strings may be interpreted as different numbers of tokens. For example, ">=" could be seen either as a single token representing the operator ($\geq$), or as two entirely separate tokens.

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 (e.g. reserved words appear before identifiers). Apart from the necessary ordering, in a large lexer it is a good idea to order tokens by probability of occurrence. This simple step can considerably improve performance.

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 the 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, defined by filter p xs = [x | x <- xs ; p x]. For example, filter (>5) [1,6,2,7] has value [6,7].

4.5 Syntax analysis

Lexical analysis has made the initial jump from characters to tokens. Syntax analysis completes the parsing process, by combining tokens to form a parse tree. For things like identifiers and numbers, their precise value is not important during syntax analysis, only that they are in fact identifiers and numbers. Thus, we can imagine an analogue to the literal primitive, which matches any token of a given class, regardless of its attribute value. In fact, once we have parsed a token, 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, the parser (kind t) only recognises tokens of class t, returning their attribute value (i.e. spelling) if successful.

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

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

For example, (kind Ident) matches any identifier, returning its spelling if successful. Because reserved words and symbols are bundled under the single tag Symbol, the kind function is not much use in these cases. We need a special function which matches on the spelling of symbols. Thus, the parser (lit xs) only admits the token (Symbol,xs). Again the tag is redundant after parsing.

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

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

For example, (lit "(") only matches a left parenthesis. We may now build the syntax analyser itself. Recall the BNF description of our example language.

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

As for the expression parser in section 2.4, we simply cast this description in combinator notation, and include semantic actions to build the parse tree. However we must take into account the offside rule, which is used to disambiguate the grammar. All we need do is apply the `offside` combinator to the body of definitions.

```
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 ")")
```

The `opt` abbreviation used in this parser corresponds to the $[\cdots]$ notation in BNF, denoting an optional phrase. It is defined by `p $opt v = p $alt (succeed v)`. To complete the parser, we must define the remaining semantic actions. But first, the somewhat strange action (`foldl1 Apply`) merits some explanation.

Recall that the original grammar in section 4.1 used left recursion to express the left associativity of application. By transforming the grammar slightly, the recursion was replaced by iteration. In combinator parsing, iteration corresponds to `many` and `some`. These operators produce a list as their result. What we really want in this case is a left recursive application spine. That is, 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 quite straightforward, simply converting the default result values to the internal form. 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 composing the four phases. For simplicity, we ignore the possibility of errors, assuming that both lexical and syntactic analysis are always successful. Since only the first result value from each parser is required, the function (`fst.hd`) is used to select this component.

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

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

There are two points to be noted from this example. Firstly, through careful choice of notation, two-phase parsers are no more complicated to build than one-phase parsers. Secondly, the `offside` combinator succinctly implements the assumption that the offside rule is used to disambiguate the grammar.

5 Summary

We started with a tutorial on the basic aspects of combinator parsing, going on to develop the notation further, so that parsers for real programming languages could be built, with separate lexical and syntactic phases. Special mention was given to layout conventions, and in particular the offside rule. We demonstrated that a parser for a Miranda style language could be built quickly and simply.

Many interesting and important aspects of combinator parsing have been omitted. In particular, it is well known that combinator parsers may have unexpected space and time behaviour, and are often not as lazy as we would expect. Simple solutions to these problems are discussed in both [2] and [5].

Acknowledgements

Thanks are due to John Launchbury, David Murphy, Duncan Sinclair and Mary Sheeran for their comments on various drafts of this paper. John in particular has contributed a great deal. His comments, and interest in combinator parsing, radically changed both the form and content of the paper.

References

[1] Alfred Aho, Ravi Sethi and Jeffrey Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Philip Wadler. *How to Replace Failure by a List of Successes.* FPCA 85, LNCS 201, 1985.

[3] John Fairbairn. *Making Form Follow Function.* Glasgow University, 1986.

[4] Richard Frost and John Launchbury. *Constructing Natural Language Interpreters in Lazy Functional Languages.* Glasgow University, 1988.

[5] John Launchbury. *Parsing in a Functional Language using Higher Order Combinators.* Glasgow University, 1988. (Unpublished)

[6] Peter Landin. *The Next 700 Programming Languages.* CACM Vol. 9, March 1966.

[7] Geraint Jones. *Programming in occam.* Prentice-Hall International, 1987.

[8] Paul Hudak and Philip Wadler (editors). *Report on the Programming Language Haskell.* Glasgow University and Yale University. 1989. (Draft)