# *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[1]; 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 $\varepsilon$ 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`)

[1] 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 $n$th 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:

DOCKET ALARM

# Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

## Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

## Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

## Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

## API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

**LAW FIRMS**
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

**FINANCIAL INSTITUTIONS**
Litigation and bankruptcy checks for companies and debtors.

**E-DISCOVERY AND LEGAL VENDORS**
Sync your system to PACER to automate legal marketing.