Continuation Passing Combinators for Parsing Precedence Grammars

Steve Hill, University of Kent, UK

November 10, 1994

Abstract

We describe a scheme for constructing parsers for precedence grammars based on the combinators described in [4]. The new combinators provide a robust method for building parsers and help avoid the possibility of a non-terminating parser. Efficiency is improved via an optimisation to the grammar. A number of approaches to the problem are described – the most elegant and efficient method is based on continuation passing. A parser for the expression part of the C programming language is presented.

1 Introduction

In this paper, we use the parsing combinators described in [4] to construct a set of higher-level combinators designed specifically to handle precedence grammars. This set is open ended – we have chosen to implement functions to handle the most common expressions syntaxes; specifically, there are combinators for infix binary operators, prefix and postfix unary operators, subexpressions and atoms. In Section 5 we develop some specialised combinators to handle some of the more unusual constructions in C.

Hutton's combinators provide a powerful set of primitives with which one may rapidly construct top-down parsers. They provide for backtracking and hence can cope with ambiguous grammars. We summarise them in Figure 1. In fact, parsers written in this style go back at least as far as [2]. All code in this paper is written in Gofer [5], whose syntax is similar to Haskell [3]. This has necessitated some function renaming to avoid clashes with keywords and the standard prelude.

Briefly, a parser is implemented by a function from a list of input tokens to a list of possible parses, where a parse is a pair consisting of the remnant of the input list and the value constructed by the parser. The value part of a parser might be a parse tree, or a basic value such as a number. The combinators use the "list of successes" technique [8] to provide backtracking, so can handle ambiguous grammars.

Parsers can be combined sequentially: `p1 ‘seq‘ p2` denotes a parser which accepts parses from `p1` followed by parses from `p2` (usually written as juxtaposition in BNF), or by using an alternation: `p1 ‘alt‘ p2` denotes a parser which accepts parses from both `p1` and `p2` (usually denoted by | in BNF). There are some useful variants on `seq` which discard the values from one or other of the parsers; they are used when we are only interested in the fact that a parser succeeds, for example when parsing a keyword. Related to these is the `return` combinator which applies a parser replacing the result value with the specified new value.

There are a number of basic parsers. The `fail` parser always fails. It is useful since it is an identity for the `alt` operator. The `succeed` parser succeeds immediately without consuming any input. This is often used to return a terminating value, such as `[]`. The `satisfy` and `literal` combinators succeed if a token satisfies a predicate or is equal to a particular value respectively. In both cases, the matched token is returned.

The `many` combinator repeatedly applies a parser until it fails returning a list of result values and corresponds to the $*$ operator in BNF. The `using` combinator applies a function (*ie.* a semantic action) to the value part of the parse. It consumes no input. The `anyof` combinator applies a function to a list of values to produce a list of parsers. These parsers are then combined using alternation. For example:

```
abc = anyof literal ['a', 'b', 'c']
```

is a parser that accepts either `'a'`, `'b'` or `'c'` which we could have written in a long-winded fashion as:

```
abc = literal 'a' ‘alt‘ (literal 'b' ‘alt‘ literal 'c')
```

We do not use the `into` combinator in this paper, but we have used it to construct versions of our combinators which avoid the construction of intermediate lists – see Section 4.5. The `into` function is similar to sequential composition except that the second parser is applied to the result from the first.

Clearly, parsers for precedence grammars may be constructed directly using these combinators. However, this often involves mechanical manipulation of the grammar and the construction of a large number of parsing rules all of which are similar. The combinators presented in this paper embody these routine manipulations, and provide a robust and quick method for building reasonably complex parsers.

2 The Grammar of Expressions

Often the most complex part of the grammar for a programming language deals with expressions. Expressions in most programming languages are built from a number of infix binary operators and prefix (and sometimes postfix) unary operators. To resolve ambiguity, typically each operator is assigned a precedence and an associativity, The expression $x \oplus y \otimes z$ can be read as $(x \oplus y) \otimes z$ or $x \oplus (y \otimes z)$ according to the precedences of $\oplus$ and $\otimes$. The expression $x \oplus y \oplus z$ can be read as either $x \oplus (y \oplus z)$ or $(x \oplus y) \oplus z$ according to the associativity of $\oplus$.

Let us consider a grammar for simple expressions:

$$e \quad ::= \quad e \text{ “}+\text{” } e \mid e \text{ “}*\text{” } e \mid \text{ “(” } e \text{ “)” } \mid v$$

(where v denotes a variable). This grammar is ambiguous. In order to construct a top-down parser for it we need to re-express the grammar. We use our knowledge of the precedence of the operators to derive a new grammar:

$$e \quad ::= \quad t \mid e \text{ “}+\text{” } e$$

$$t \quad ::= \quad f \mid t \text{ “}*\text{” } t$$

$$f \quad ::= \quad \text{“(” } e \text{ “)” } \mid v$$

which gives multiplication a higher precedence than addition. Unfortunately this grammar is not suitable for a top-down parser since it involves left-recursion, which would lead to non-termination. Again, we must re-express the grammar. Our tactic is to replace recursion with iteration in the following way:

$$e \quad ::= \quad t \, (\text{“}+\text{” } \; t) *$$

$$t \quad ::= \quad f \, (\text{“}*\text{” } \; f) *$$

$$f \quad ::= \quad \text{“(” } e \text{ “)” } \mid v$$

```
type Parser t v = [t] -> [(v,[t])]

-- Immediately succeed.  Consumes no tokens.
succeed :: v -> Parser t v

-- Always fails.
fail :: Parser ts v

-- Succeeds if predicate is True.
satisfy :: (t -> Bool) -> Parser t t

-- Match a literal token
literal :: Eq t => t -> Parser t t

-- Alternation, parses from either p1 or p2 or both.
alt :: Parser t v -> Parser t v -> Parser t v

-- Sequential composition, parses of p1 followed by p2
-- Variants throw away result from first or second parser.
seq :: Parser t v1 -> Parser t v2 -> Parser t (v1, v2)
xseq :: Parser t v1 -> Parser t v2 -> Parser t v2
seqx :: Parser t v1 -> Parser t v2 -> Parser t v1

-- Apply semantic action to value
using :: Parser t v1 -> (v1 -> v2) -> Parser t v2

-- Repetition, keep applying parser until it fails.
many :: Parser t v -> Parser t [v]

-- Throws away parse tree returns supplied value instead.
return :: Parser t v1 -> v2 -> Parser t v2

-- Monadic style combinator - result passed to next parser
into :: Parser t v1 -> (v1 -> Parser t v2) -> Parser t v2

-- Combines a list of parsers with alternation.
-- Parsers obtained by applying function to a list of values.
anyof :: (a -> Parser t v) -> [a] -> Parser t v
```

Figure 1: Hutton's Parsing Combinators

where the notation $x*$ denotes zero or more occurrences of x. There are other re-arrangements of the grammar which accept the same language, but they either give the wrong associativity or are still left-recursive. This new grammar is suitable for a top-down parser, but there is still ambiguity regarding associativity (although clearly in this example it is not important because the operators are associative). The semantic actions associated with these productions will be responsible for resolving this ambiguity.

Grammar manipulation is mechanical, tedious and prone to error. We have also ended up with one rule for every level of precedence all of which are essentially the same. Instead, let us propose the following parametrised rule which captures the grammatical pattern for an infix binary operator at precedence level n:

$$e_n \quad ::= \quad e_{n-1}\ (\oplus_n\ e_{n-1})*$$

Most programming languages have a number of operators occupying each level of precedence, so we need to generalise this rule in the following fashion; if precedence level n has k operators, the rule is:

$$\begin{array}{lll} e_n & ::= & e_{n-1}\ (\oplus_n^1\ e_{n-1})*\ \mid \\ & & e_{n-1}\ (\oplus_n^2\ e_{n-1})*\ \mid \\ & & \ldots \\ & & e_{n-1}\ (\oplus_n^k\ e_{n-1})* \end{array}$$

Note that the first and last non-terminal in each alternative is the same. A more efficient parser may be constructed by factoring these terms out leading to:

$$e_n \quad ::= \quad e_{n-1}\ ((\oplus_n^1 \mid \oplus_n^2 \mid \ldots \oplus_n^k)\ e_{n-1})* \tag{1}$$

Similar manipulations lead to the following rules for prefix and postfix unary operators. The rule for prefix operators is:

$$e_n \quad ::= \quad (\oplus_n^1 \mid \oplus_n^2 \ldots \oplus_n^k)*\ e_{n-1} \tag{2}$$

and that for postfix:

$$e_n \quad ::= \quad e_{n-1}\ (\oplus_n^1 \mid \oplus_n^2 \ldots \oplus_n^k)* \tag{3}$$

In the next section, we show a number of approaches to the implementation of these rules.

3 Representing the Grammar

The final form of our combinators is presented in Section 4. First we describe two earlier approaches which we hope will provide a better insight into their motivation and operation. Each method provides a toolkit for constructing parsers for expressions involving at least infix binary and prefix unary operators. However, it is clear that many real languages require support for peculiar features. This motivates our move away from an approach based on algebraic data to a higher order method.

3.1 Explicit Data

In the first method we represent a grammar as a table (or list). The table enumerates the tokens corresponding to operators in the grammar, and associates these with semantic actions (for example, to build a parse tree or evaluate an expression). Thus we define a type `Ptable`:

```
type Ptable token exp = [Rule token exp]

type Rule token exp = [(token, exp->exp->exp)]
```

where the rules in the parse table are listed in increasing order of precedence. The parser examines a `Ptable` processing each level of precedence in turn attempting to match expressions involving the specified tokens. It constructs the parse tree from the operators paired with each token. In practice, we need more than one sort of rule, since we wish to handle binary and unary operators, as well as sub-expressions and atoms. A more realistic `Rule` type might be:

```
data Rule token exp =
    Binopr [(token, exp->exp->exp)]  |
    Binopl [(token, exp->exp->exp)]  |
    Prefix [(token, exp->exp)]  |
    Postfix [(token, exp->exp)] |
    Subexp [(token, token)] |
    Atom
```

A parser is constructed by applying an interpreter to the grammar table, for example:

```
parser =
    parse
    [
        Binopl [("+", Plus), ("-", Minus)],
```

```
        Binopl [("*", Times), ("/", Divide)],
        Binopr [("$", Apply)],
        Subexp [("(",")")]
        Atom
    ]
```

Each entry in the table is processed by a different function. The functions corresponding to each rule type take the remnant of the parse table as an argument. They can then call the parser again in order to parse higher precedence rules.

```
parse :: Ptable token exp -> Parse token exp

parse ((Binopl ops):rest) =
    binopl rest ops
parse ((Binopr ops):rest) =
    binopr rest ops
...
parse [] =
    fail

binopl ptable ops =
    parse ptable `seq` ...
```

3.2 Using Functions

The problem with the previous approach is that we need a constructor for each sort of operator. We also suffer an interpretive overhead. Notice that the constructors merely serve to identify the function that should be used to parse a particular level of precedence. In a functional language we shouldn't be afraid of using functions! We can replace the entries in our table with the parsing functions themselves, giving the new types:

```
type Ptable token exp = [Rule token exp]

data Rule token exp = Rule (Ptable -> Parse Char Expr)
```

Unfortunately, neither the type system of Miranda[1] [7] nor of Haskell [3] allow recursive type synonyms. We are forced to use a data constructor to "break the loop". The parsing function now becomes:

```
parse :: Ptable token exp -> Parse token exp

parse (Prule f:fs) =
```

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

```
    f fs
parse [] =
    fail
```

and the parse table looks like this:

```
parser =
    parse
    [
        Prule (binopl [("+", Plus),("-", Minus)])
        ...
```

This method is also more flexible. Any parsing function with the correct type can be slotted into the parse table. The intention is that these functions should process their own precedence level, and where appropriate call the `parse` function on the remnant of the parse table to deal with higher levels of precedence.

3.3 Using Continuations

The `parse` function in the previous section is still essentially an interpreter. We also have to use a constructor that is not logically necessary – it merely serves to keep the type system happy. Fortunately, we can do better. The value that is passed to each rule function (the remnant of the parse table) is a *representation* of the computation that is required in order to parse any higher precedence operators. Why do we need a representation? Why not pass this computation explicitly *ie* as a function?

The type of a typical parsing function now becomes:

```
binopr :: ... -> Parse token exp -> Parse token exp

binopr ... next = ... next ...
```

The parameter `next` *is* the function to parse the next highest level of precedence – it is a continuation. This is not the only instance where continuations have proved useful in compiling techniques [1].

A parser is now constructed by applying the lowest precedence parser to the next level's parser which is in turn applied to the next and so on. For example:[2]

[2] The `$` symbol stands for function application – it associates to the right. In Miranda `$id` would have the same effect.

```
parser :: Parse [Char] Expr

parser = binopl [("+", Plus), ("-", Minus)]   $
         binopl [("*", Times), ("/", Divide)] $
         ...
         atom
```

4 The Combinators

In Section 2, we derived rules for operator precedence grammars suitable for a top-down parser. In this section, we convert these definitions into concrete code using the continuation-based method described above. In the next section, we will use the combinators to build a realistic parser for the expression part of the C programming language.

We require our set of basic combinators to deal with the following constructs:

- infix binary operators with left and right associativity,
- prefix and postfix unary operators,
- subexpressions and
- atoms.

Note that the set of combinators is not fixed. New combinators can be defined as the need arises – in fact, we will develop some in the next section.

Before we define any combinators, let us first define a simple parser.

```
litret :: Eq t => (t, v) -> Parser t v

litret (t, o) = literal t `return` o
```

This parser matches a token, throws it away and returns the value `o`. Our parsing combinators will use it to recognise operators and convert them to their semantic actions. The tokens and their corresponding node constructors will be held by a list of pairs, so the parser:

```
anyof litret ops
```

where `ops` is such a list, is a parser that accepts the listed tokens and converts them to their associated value.

4.1 Unary Operators

Let us begin with unary prefix operators. The parser is parametrised on a table of pairs. The first item is the token representing the prefix operator, and the second is the semantic action (or node constructor if we are building a parse tree). To parse a unary prefix operator, we use the grammar given earlier. A transliteration of the grammar (Equation 2) leads to:

```
prefix :: Eq t => [(t, v->v)] -> Parse t v -> Parse t v

prefix ops next
    = (many (anyof litret ops) `seq` next) `using` build
      where
      build (os, e) = foldr ($) e os
```

Here the `many` parser is applied to a parser that tries to match the tokens at this level of precedence, replacing them with their semantic actions when succesful. Once the prefix operators have been consumed, we parse any higher precedence operators. Thus, the result of the parser is a pair consisting of a list of semantic actions of type `v->v` and a value of type `v`. The function `build` combines these together using function application in the following manner:

$$build([\oplus_1, \oplus_2 \ldots \oplus_k], e) = \oplus_1(\oplus_2 \ldots (\oplus_k e) \ldots)$$

The postfix parser is very similar. The grammar is adjusted – the higher precedence parser is invoked first followed by a parser for a list of postfix operators (see Equation 3). The `build` function is also different since the list is built in a different sense – the first element of the list should be applied first rather than last.

```
postfix :: Eq t => [(t, v->v)] -> Parser t v -> Parser t v

postfix ops next
    = (next `seq` (many (anyof litret ops))) `using` build
      where
      build (e, os) = foldl (converse ($)) e os
```

The `converse` function is defined as:

```
converse f x y = f y x
```

It is interesting to note that earlier versions of Miranda [6] had a version of `foldl` which behaved as:

```
oldfoldl op = foldl (converse op)
```

which is precisely what we require here.

4.2 Binary Operators

When dealing with binary infix operators, we have the added complexity of associativity. However, the grammar for left- and right-associative operators is identical, so we can tackle associativity independently. Let us deal with the grammar first.

```
binop :: Eq t => Assocfn v -> [(t, v->v->v)] ->
                 Parser t v -> Parser t v

binop assoc ops next
    = (next `seq` op2) `using` assoc
      where
      op2 = (many (anyof litret ops `seq` next))
```

The binary operator parser is defined from the grammar (Equation 1). As with the unary operators, the `ops` argument is a table enumerating the operator tokens and their associated semantic actions and the `next` parameter is a parser for the next level of precedence. The `binop` function looks for an expression with higher precedence followed by a sequence of operators and expressions. The function `assoc` is used to re-arrange the resulting list according to the associativity of the operators.

We can now tackle the associativity problem. We can specialise `binop` to handle left and right association according to the `assoc` parameter, so:

```
binopr :: Eq t => [(t, v->v->v)] -> Parser t v -> Parser t v

binopr = binop assocr

binopl :: Eq t => [(t, v->v->v)] -> Parser t v -> Parser t v

binopl = binop assocl
```

Finally, we need to define the associativity functions. Their type is:

```
type Assocfn v = (v, [(v->v->v, v)]) -> v
```

that is, they consume a value and a list of operator value pairs combining them into a single value either grouping to the left or to the right. Informally, the operations we require are:

$$assocr\ (e_0, [(\oplus_1, e_1), (\oplus_2, e_2) \ldots (\oplus_k, e_k)]) = e_0 \oplus_1 (e_1 \oplus_2 (e_2 \ldots \oplus_k e_k) \ldots))$$

$$assocl\ (e_0, [(\oplus_1, e_1), (\oplus_2, e_2) \ldots (\oplus_k, e_k)]) = (\ldots((e_0 \oplus_1 e_1) \oplus_2 e_2) \ldots \oplus_k e_k)$$

and these can be defined formally as:

```
assocr (e1, (op, e2) : l)
    = op e1 (assocr (e2, l))
assocr (e, [])
    = e

assocl (e, l)
    = foldl f e l
      where
      f e1 (op, e2) = op e1 e2
```

4.3 Subexpressions and Atoms

We have two further parsers to consider. We need a combinator to deal with sub-expressions and another to parse the atoms of our expressions. We will define a generic sub-expression combinator which allows for different styles of parentheses.

```
subexp :: Eq t => Parser t v -> [(t,t)] ->
          Parser t v -> Parser t v

subexp back bs next
    = anyof subexp' bs `alt` next
      where
      subexp' (op, cl)
          = (literal op `xseq` back) `seqx` literal cl
```

The subexpression combinator first matches the open brace. The sub-expression itself is parsed by the function parameter `back` which would normally be the parser for top-level expressions (although one is at liberty to use any suitably typed parser). Finally, we match the closing brace. If we fail to match a subexpression, we proceed to the next level of precedence.

The final combinator is responsible for parsing atoms. This parser will be used as the final level of precedence, so has no `next` parameter. The atom parser has two parameters, a recogniser and a semantic action. The recogniser checks that the next input token is a valid atom, and the semantic action is then applied to recognised tokens.

```
atom :: (t -> Bool) -> (t -> v) -> Parser t v

atom rec leaf =
    satisfy rec `using` leaf
```

4.4 Example

Recall the simple expression grammar from Section 2. We can now use our combinators to construct a parser. We need to assign a precedence level and associativity to each operator. Let us say that addition has the lower precedence and that both addition and multiplication group to the left, as is customary. A suitable parser is then given by:

```
parse :: Parser [Char] Tree

parse = binopl [("+", Plus)] $
        binopl [("*", Times)] $
        subexpr [("(",")")] $
        atom isAtom Atom
```

4.5 Discussion

When using these combinators, the implementation of a wide range of common expression grammars is quick and simple. A parser can be written directly from the language grammar and precedence rules. Moreover, provided that we use just the core set of combinators, we are assured that our parser will terminate (assuming that the semantic actions do). The parsers for infix and prefix operators embody the grammar transformations required to remove left-recursion. The sub-expression combinator could introduce a loop, but since it always consumes a token there is no possibility of non-termination. The atom parser will terminate provided that the recogniser does.

It is worth noting that it is possible to define our combinators such that they do not construct intermediate lists. The alternative definitions make use of the `into` parser, and are slightly more efficient. However, the definitions are more complicated than those shown here.

5 Example: Parsing C Expressions

The C language has a notoriously complex expression syntax. This is evidenced by the existence of a tool *cparen* which parses C expressions and outputs them fully parenthesised. We have used the combinators developed in the previous section to build a functional program similar to *cparen*.

In this section, we will describe the parser from our *cparen* program. Its task is to construct a parse tree from a list of input tokens. We will assume that a lexical analysis has taken place (our lexical analyser is in fact built using the lower level combinators described in Section 1). We do not show the trivial `unparse` function which converts the parse tree into a fully bracketed expression string. In fact, it would be possible to avoid constructing the parse tree at all, and instead apply the unparse operations as semantic actions.

We first define a data type to represent C expressions, of which the following is a part:

```
data CExp =
        Comma CExp CExp |
        Assign CExp CExp |
        PlusAssign CExp CExp |
        ...
        Func CExp CExp |
        Arglist [CExp] |
        CondOp CExp CExp CExp |
        Atom [Char]
```

Next, we build the parser using the combinators from the previous section. It is worth noting at this point that the syntax of C expressions is rather peculiar in its treatment of function arguments. The comma symbol has two meanings in C. It is used to delimit function argument lists, but it is also an operator. The expression `a, b` has the value `b` but, as a side-effect, it also evaluates `a`. So an expression `f(a, b)` could be parsed as either a function call with two arguments, or a call with one expression argument `(a, b)`. In fact, the former interpretation is intended. This peculiarity requires us to have two versions of our parser – implemented as two entry points. The first parses expressions including the comma operator. The second is used when parsing function arguments, and requires that comma expressions be parenthesised.

We present the parser in Figure 2. For the most part, we are able to define the parser in terms of the combinators described in the previous section. However, there are a few syntactic constructs that require additional definitions. The first of these is the ternary conditional operator. A parser for this operator is:

```
condop :: Parser [Char] CExp -> Parser [Char] CExp

condop next
    = (condop' `using` mkCondop) `alt` next
      where
      condop' = toquery `seq` (tocolon `seq` cparser)
      toquery = next `seqx` literal "?"
```

```
      tocolon = cparser `seqx` literal ":"
      mkCondOp (e1, (e2, e3)) = CondOp e1 e2 e3
```

In order to parse functions and arrays, we develop another combinator which is a generalisation of `binopl` with the following type:

```
genopl :: Eq t => [(Parser t v -> Parser t v, v->v->v)] ->
          Parser t v -> Parser t v
```

The table given to `genopl` contains a list of pairs. The second element is, as before, the semantic action. The first element is a parsing combinator *ie.* it is a parser which takes an argument parser for higher precedence expressions.

```
genopl ops next
    = (next `seq` op2) `using` assocl
      where
      op2 = many (foldr1 alt (map mkParser ops))
      mkParser (p, o) = succeed o `seq` p next
```

To explain – we apply mkParser to each of the list entries to produce a list of parsers. Each parser will have been applied to the *next* parser, so can handle higher precedence expressions. These parsers return a pair consisting of the semantic action for the operator, and an operator argument value. For completeness, the companion function `genopr` with right associativity can be defined in a similar manner.

We can use `genopl` to obtain the same effect as `binopl` as, for example:

```
parser = genopl [(oparg "->", Pointer), (oparg ".", Dot)]

oparg t next = literal t `xseq` next
```

The `oparg` parser recognises an infix operator (`genopl` will already have parsed the first argument), followed by an expression of higher precedence. Thus the above could have been written as:

```
parser = binopl [("->", Pointer), (".", Dot)]
```

We need to use `genopl` when operators with a conventional infix syntax have the same precedence level as other expression forms not handled by the basic combinators. In the C expression parser, for example, we use it to parse functions and arrays which occupy the same level of precedence as the structure element referencing operators. For example, arrays are parsed with the function:

```
array next = (literal "[" `xseq` cparser) `seqx` literal "]"
```

Notice that the `next` parser is not used since the array parser calls the top-level expression parser to process its argument. Note also that `genopl` will have already parsed the expression denoting the address of the array. The parser for functions is similar, except that it must parse a list of arguments. Moreover, it has to use `cparser1` to avoid the comma ambiguity described earlier.

6 Conclusions

We believe that these higher-level combinators provide a useful addition to the parser writer's toolbox. They allow parsers for reasonably complex grammars to be constructed rapidly and accurately. Once our combinator set had reached its final form, it took approximately an afternoon's work to write the functional *cparen* tool. Further work will reveal whether there are other common syntactic patterns that deserve their own combinators. The experiment with C was remarkable in that it lead to the definition of only two extra combinators. Although we have used Hutton's set of basic combinators in this paper, it is possible to base our combinators on other sets – in particular sets that provide for less backtracking will be more efficient.

```
cparser =
        binopl [(",", Comma)]                         $
        cparser1

cparser1 =
        binopr [("=", Assign),
                ("+=", PlusAssign),
                ("-=", MinusAssign),
                ("*=", MulAssign),
                ("/=", DivAssign)]                    $
         condop                                       $
         binopl [("||", Or)]                          $
         binopl [("&&", And)]                         $
         binopl [("|", BitOr)]                        $
         binopl [("^", BitEor)]                       $
         binopl [("^", BitEor)]                       $
         binopl [("&", BitAnd)]                       $
         binopl [("==", Equal),
                 ("!=", NotEqual)]                    $
         binopl [("<", Less),
                 ("<=", LessEq),
                 (">", Greater),
                 (">=", GreaterEq)]                   $
         binopl [("<<", LeftShift),
                 (">>", RightShift)]                  $
         binopl [("+", Plus),
                 ("-", Minus)]                        $
         binopl [("*", Times),
                 ("/", Divide),
                 ("%", Mod)]                          $
         prefix [("++", PreInc),
                ("--", PreDec),
                ("!", Not),
                ("~", BitNot),
                ("*", Indirect),
                ("+", UnaryPlus),
                ("-", UnaryMinus),
                ("&", Address)]                       $
        postfix[("++", PostInc),
                ("--", PostDec)]                      $
        genopl [(oparg "->", Pointer),
                (oparg ".", Dot),
                (array, Array),
                (func, Func)]                         $
        subexp cparser [("(",")")]                    $
        atom isAtom Atom
```

Figure 2: The C Expression Parser

References

[1] Andrew W. Appel. *Compiling with Continuations*. Combridge University Press, 1992.

[2] W. H. Burge. *Recursive Programming Techniques*. Addison Wesley, 1975.

[3] P. Hudak, S. Peyton Jones, and P.L Wadler (editors). Report on the functional programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[4] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3), July 1992.

[5] Mark P. Jones. *Introduction to Gofer 2.20*, 1991. Available via ftp from *nebula.cs.yale.edu*.

[6] Research Software Ltd. *Miranda System Manual*, 1990. On-line manual section 28.

[7] D. A. Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.

[8] Philip Wadler. How to replace failure by a list of successes. In *Lecture Notes in Computer Science 201*. Springer-Verlag, 1985.