*J. Functional Programming* **6** (3): 445–463, May 1996  

# *Combinators for parsing expressions*

STEVE HILL

*University of Kent, Canterbury, UK*

**Abstract**

This paper describes a scheme for constructing parsers based on the top-down combinator approach. In particular, it describes a set of combinators for parsing expressions described by ambiguous grammars with precedence and associativity rules. The new combinators embody the mechanical grammar manipulations typically employed to remove left-recursion and hence help to avoid the possibility of a non-terminating parser. A number of approaches to the problem are described—the most elegant and efficient method is based on continuation passing. As a practical demonstration, a parser for the expression part of the C programming language is presented. The expression combinators are general, and may be constructed from any suitable set of top-down combinators. A comparison with parser generators shows that the combinator approach is most applicable for rapid development.

**Capsule Review**

Designing libraries of higher-order functions (or combinators) to solve special programming problems is an important style of functional programming. Examples of such libraries include those for building parsers, pretty printers and graphical user interfaces. Hill's paper extends the parsing library. The extension is a small set of derived combinators tailored to building parsers for expression-like grammars. The new combinators encapsulate the tedious and error-prone grammar transformations that are required to take account of precedence and associativity rules. The new combinators are well-chosen, and developed in a stepwise manner which makes them easy to understand and use. A particularly nice aspect of the method is that precedence levels don't have to be named (or numbered); rather, a special combinator to handle precedences is introduced.

On the plus side, combinator parsers are typically easy to build and modify, and one has the full power of a functional language available to program semantic actions and extend the basic library of combinators. On the minus side, it is well-known that combinator parsers can be slow compared to automatically generated table-driven parsers. For many research applications, however, the performance will be adequate. In such situations Hill's new combinators will be a welcome and long-overdue addition to the parsing library, considerably reducing the time and effort required to obtain a working parser.

## 1 Introduction

This paper describes a set of high-level combinators for the construction of parsers for ambiguous expression grammars which are disambiguated by the use of associativity and precedence rules. The expression components of most modern program-

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

ming languages fall into this category. The principal advantages of the approach are:

- conciseness and transparency of expression,
- mechanical grammar manipulation is avoided,
- a rapid development cycle,
- parser non-termination is avoided, and
- acceptable efficiency.

There are two main approaches to the construction of parsers from a (possibly annotated) grammar. The first method, as exemplified by systems such as *yacc* (Johnson, 1978; Aho *et al.*, 1986), its early functional implementation (Peyton Jones, 1985) and the more recent Happy (Gill and Marlow, 1995) and Ratatosk (Mogensen, 1993), derives a table-driven automaton from a grammar, whose execution is steered by the input tokens. Parsers developed in this way are efficient. However, construction of a parser can be a lengthy many-stage process. First the grammar is written, then the parser generator processes the grammar to produce a module which can then be executed or included in a larger system. Errors in the grammar source file often do not become apparent until the end of this process, so many design iterations are quite common.

The second approach, and the one adopted here, is to build parsers from a set of directly executable combinators. The set of combinators usually includes the basic grammar constructs such as alternation and concatenation, and may be extended with derived combinators for repetition and multi-way choice. Depending on the requirements of the grammar, different implementations of the combinators may be chosen. The most common implementations use lists of results to represent parsers which may fail or succeed with a number of different results (Wadler, 1985). Parsers built using this style of combinator provide backtracking, and may be used to construct parsers from ambiguous grammars.

The functions in this paper are described with reference to the combinators described in Hutton (1992). Many authors have discussed or implemented similar sets, for example, Fairbairn (1986), Burge (1975) and Augustsson (1994). However, the choice of a particular set is not important to this work. The approach is general and other combinators sets may be used with equal facility. However, the resulting parsers may accept different languages according to the specific grammar and amount of backtracking provided by the combinators.

Combinator parsers have a number of weaknesses. In particular, they cannot handle grammars which contain left-recursion. However, it is possible to re-arrange a grammar to eliminate the left-recursion, whilst still accepting the same language. Unfortunately, clarity can be lost during translation, and since the manipulations are often done by hand, incorrect grammars can easily result. The combinators described here, embody the necessary grammar manipulations and hence provide a more secure method for building parsers whilst retaining the clarity of the original grammar. Another weakness of many combinator sets is their efficiency, which is discussed further in section 7.

Their significant advantage is that parsers described in this style are directly executable. This allows for a rapid turn-around from specifying the grammar to producing a working parser. It suggests that combinator-based parsers are best suited for early development and rapid prototyping where their performance is acceptable.

## 2 Background

The signature of the combinator set described in Hutton (1992) is given in figure 1. All code in this article is written in Gofer (Jones, 1991). This has necessitated some function renaming to avoid clashes with keywords and the standard prelude.

Hutton implements a parser as a function from a list of input tokens to a list of possible *parses*. A parse is a pair consisting of the remnant of the input and a result value constructed by the parser. The values produced by the parser might be parse trees, but in general can be of any type. It is possible to decorate parsers with semantic actions to process these results.

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 it is only of interest if a parser has succeeded and not what the result is, 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 and is the unit for the `alt` operator. The `succeed` parser succeeds immediately without consuming any input and returns a specified value. The `satisfy` and `literal` combinators succeed if the next token satisfies a predicate or is equal to a particular value respectively. In both cases, the matched token is returned.

The derived combinator, `many`, 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 (i.e. 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')
```

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

### 2.1 Example parser

This section demonstrates the construction of a parser, using Hutton's combinators, for a fragment of a simple programming language. The grammar is as follows:

$$
\begin{array}{lll}
\mathit{expr} & ::= & \ldots \\
\mathit{type} & ::= & \ldots \\
\mathit{decl} & ::= & \mathit{var} :: \mathit{type} \\
\mathit{assn} & ::= & \mathit{var} := \mathit{expr} \\
\mathit{line} & ::= & \mathit{decl} \mid \mathit{assn} \\
\mathit{prog} & ::= & \mathit{line}^*
\end{array}
$$

The conversion of this grammar to an executable parser is straightforward. Each rule is implemented as a function of type `Parser tok tree` where `tok` is the type of objects returned by the lexical stage, and `tree` is the type of object generated by the rule. Writing the parsers is mechanical, and follows the grammar closely. Semantic actions are added via the `using` combinator which in this case just serves to build the parse tree. The parser is thus:

```
data Expr = ...
data Type = ...
data Line = Assign [Char] Expr | Declare [Char] Type
type Prog = [Line]

parse_decl   = ((parse_var is_name `seqx` literal "::")
               `seq` parse_type) `using` mk_decl
               where
               mk_decl (n, t) = Declare n t

parse_assign = ((satisfy is_name `seqx` literal ":=")
               `seq` parse_expr) `using` mk_assign
               where
               mk_assign (n, e) = Assign n e

parse_line   = parse_decl `alt` parse_assign

parse_prog   = many parse_line
```

## 3 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 and prefix (and sometimes postfix) unary operators. To resolve ambiguity, each operator is typically 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 the infix operators $\oplus$ and $\otimes$. The expression $x \oplus y \oplus z$ can be read as

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

```
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
```

Fig. 1. Hutton's parsing combinators.

either $x \oplus (y \oplus z)$ or $(x \oplus y) \oplus z$ according to the associativity of $\oplus$. To avoid further ambiguity, operators with the same precedence must have the same associativity.

Consider a grammar for simple arithmetic expressions:

$$e \quad ::= \quad e + e \mid e * e \mid \ ( \ e \ ) \ \mid v$$

(where $v$ denotes a variable). This grammar is ambiguous. To resolve the ambiguity, precedences are assigned to the operators and the grammar re-expressed thus:

$$\begin{array}{lll} e & ::= & t \mid e + e \\ t & ::= & f \mid t * t \\ f & ::= & \ ( \ e \ ) \ \mid v \end{array}$$

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

which gives multiplication a higher precedence than addition. Unfortunately, this grammar is not suitable for implementation using a top-down parser since it involves left-recursion, which would lead to non-termination. Again, the grammar must be re-expressed, replacing recursion with iteration (Aho *et al.*, 1986) in the standard way:

$$\begin{array}{lcl} e & ::= & t\ (+\ t)^* \\ t & ::= & f\ (*\ f)^* \\ f & ::= & (\ e\ )\ \mid v \end{array}$$

where the notation $x^*$ denotes zero or more occurrences of $x$. This new grammar is suitable for a top-down parser. There is still an ambiguity regarding associativity, although clearly in this particular example it is not significant. The semantic actions associated with these productions will be responsible for resolving grouping. A suitable parser for the grammar is thus:

```
data Tree = Token Char | Times Tree Tree | Plus Tree Tree

parse_e :: Parser Char Tree

parse_e = (parse_t `seq` many (literal '+' `xseq` parse_t))
          `using` mk_plus
          where
          mk_plus (e, l) = foldl Plus e l

parse_t :: Parser Char Tree

parse_t = (parse_f `seq` many (literal '*' `xseq` parse_f))
          `using` mk_times
          where
          mk_times (e, l) = foldl Times e l

parse_f :: Parser Char Tree

parse_f = (literal '(' `xseq` (parse_e `seqx` literal ')')) `alt`
          ((satisfy is_token) `using` Token)
```

This parser, although adequate in terms of the language it accepts, is a far remove from the original grammar. The aim of this work is to provide a set of higher-level combinators which retain a clear correspondence between the grammar and its implementation.

Grammar manipulation by hand is mechanical, tedious and prone to error. Moreover, there is a virtually identical rule for every level of precedence. Instead, let us propose the following parameterised rule which captures the grammatical pattern for an infix binary operator at precedence level $n$:

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

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

In general, there will be a number of operators occupying each level of precedence, so a more general rule is required. If precedence level $n$ has $k$ operators, the rule is:

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

Similar rules for prefix and postfix unary operators can be derived. The rule for prefix operators is:

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

and that for postfix:

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

These rules could be used directly to implement parsers. However, the goal of this work is to provide a set of generic combinators that embody their patterns. In the schemes that follow, there will be no need to name each rule, nor will it be necessary to assign arbitrary numeric values to each level of precedence.

## 4 Implementation

The final form of the combinators, presented in section 5, was achieved via a number of stepwise refinements. This section briefly describes these steps to 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, many real languages require support for special features. and this motivates a move away from an approach based on algebraic data to a higher-order method.

### *4.1 Explicit data*

In the first method a grammar is represented 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 a type `Ptable` is defined as:

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

type Prule 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, more than one sort of rule is needed, since realistic grammars will feature at least binary and unary operators, as

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

well as sub-expressions and atoms. A more realistic `Prule` type is:

```
data Prule 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 now 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 ops rest
parse ((Binopr ops):rest) = binopr ops rest
...
parse [] = fail

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

### *4.2 Using functions*

A drawback of the previous approach is that a constructor is needed for each kind of operator committing the method to a fixed repertoire. 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! The entries in the table can be replaced with the parsing functions themselves, giving the new types:

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

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

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

Most popular non-strict functional languages (Hudak *et al.*, 1992; Turner, 1986; Jones, 1991) do not support recursive type synonyms. A data constructor must be used to 'break the loop'. The top-level parsing function now becomes:

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

parse (Prule f:fs) = f fs
parse [] = fail
```

and the parse table looks like this:

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

This method is 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 top-level `parse` function on the remnant of the parse table to deal with higher levels of precedence.

### *4.3 Using continuations*

The `parse` function in the previous section is still essentially an interpreter. A constructor that is not logically necessary is also required. Fortunately, there is a better, more direct approach. 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, i.e. 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 transformation techniques (Appel, 1992).

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:

```
parser :: Parse [Char] Expr

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

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

The $ symbol stands for function application (in Miranda† $id would have the same effect) and associates to the right. It is used to make parsers more readable. A programmer can regard $ simply as a combinator for combining the parsers at the various precedence levels, and need not be concerned with continuations or other implementation details.

## 5 The combinators

In section 3, rules were derived for operator precedence grammars suitable for implementation using a top-down parser. In this section, these definitions are translated into concrete code using the basic parsing combinators described by Hutton, and the continuation-based method described above. In the next section, the new combinators will be used to build a parser for the expression part of the C programming language.

The remainder of the section will require the use of the following parser:

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

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

This matches a token, throws it away and returns the value v. The new combinators will use it to recognise operators and convert them to their semantic actions. The tokens and their corresponding node constructors will be held as 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.

### *5.1 Unary operators*

To parse a unary prefix operator, the grammar (Equation (2)) is transliterated thus:

```
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
```

The parser is parameterised on a table of pairs where the first item is the token representing the prefix operator, and the second is the semantic action. 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

† Miranda is a trademark of Research Software Ltd.

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

prefix operators have been consumed, any higher precedence operators are then parsed.

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 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 similar, except that higher precedence operators are parsed first, and the resulting list is arranged in the opposite order. 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.

```
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 (flip ($)) e os

flip f x y = f y x
```

### *5.2 Binary operators*

When dealing with binary infix operators, associativity adds to the complexity of the problem. However, the *grammar* for operators which associate either to the left or to the right is identical, so the problem can be split into two separate parts.

The grammar can be handled by the following function derived from Equation (1) and parameterised on the semantic action to handle associativity:

```
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))
```

As with the unary operators, the `ops` argument is a table enumerating the operator tokens and their associated semantic action. 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.

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

Associativity is handled by specialising `binop` to handle left and right association according to the `assoc` parameter, thus:

```
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, the associativity functions must be defined. Their type is:

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

i.e. 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 required 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
```

### *5.3 Subexpressions and atoms*

There are two further combinators to consider: one for sub-expressions and one to handle the atoms. For generality, a generic sub-expression combinator which allows for different styles of parentheses is defined.

```
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 sub-expression combinator first matches the open brace followed by the sub-expression itself which is parsed by the function parameter `back`. This would normally be the parser for top-level expressions (although one is at liberty to use any suitably typed parser). Finally, the closing brace must also be matched. If the combinator fails, then it proceeds to the next level of precedence.

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

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
```

### *5.4 Example*

Recall the simple expression grammar from section 3. Using the new combinators described above gives the following parser:

```
parse :: Parser [Char] Tree

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

Note that this parser corresponds closely in structure to the simple grammar for expressions given at the beginning of section 3. Precedence and associativity is handled by the combinators, rather than by making complicated grammar transformations. In this way, the time taken to get from a grammar to a combinator parser is considerably reduced.

### *5.5 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 just the core set of combinators are used, it is guaranteed that the 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. In normal use, 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 the combinators such that they do not construct intermediate lists. The alternative definitions make use of the `into` parser described by Hutton, and are slightly more efficient. However, the definitions are more complicated than those shown here.

There is a performance overhead associated with the expression combinators, which in the examples that have been examined represents a slow-down of the order of 30%.

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

## 6 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. In this section, the combinators developed in the previous section are used to build a functional program similar to *cparen*.

The task is to construct a parse tree from a list of input tokens. It is assumed that lexical analysis has already been implemented (possibly using the lower level combinators described in section 2, or using a tool like *lex* (Lesk and Schmidt, 1975)). The trivial `unparse` function which converts the parse tree into a fully bracketed expression string is not shown. In fact, it would be possible to avoid constructing the parse tree at all, and instead apply the unparse operations directly as semantic actions.

The parse tree is represented by the following type of abstract C expressions, of which the following is a representative part:

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

Next, the parser is built 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 two versions of the 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.

The whole parser is presented in figure 2. For the most part, it is defined 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 terniary 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)
```

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

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

cparser1 =
        binopr [("=", Assign),
                ("+=", PlusAssign),
                ("-=", MinusAssign),
                ("*=", MulAssign),
                ("/=", DivAssign)]               $
         condop                                  $
         binopl [("||", Or)]                     $
         binopl [("&&", And)]                    $
         binopl [("|", BitOr)]                   $
         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
```

Fig. 2. The C expression parser.

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

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

To parse functions and arrays, another combinator is required 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 now a parsing combinator, i.e. 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. When invoked, each parser will have been applied to the *next* parser, so can handle higher precedence expressions. They return a pair consisting of the semantic action for the operator, and an operator argument value.

The genopl combinator can be used 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)]
```

The function genopl is needed 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, it is used 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.

## 7 Efficiency

Are the parsers generated using this method usable in practice? Table 1 provides a comparison between five versions of the *cparen* tool. The first three use the expression

https://doi.org/10.1017/S0956796800001799 Published online by Cambridge University Press

Table 1. *Comparison of parsing techniques*

| | Reductions | Cells | Turn-around (s) |
|---|---|---|---|
| Combinators (Hutton) | 50000 | 70000 | 0.9 |
| Combinators (Augustsson) | 39000 | 150000 | 1.2 |
| Combinators (Maybe) | 38000 | 50000 | 0.9 |
| Ratatosk | 5100 | 11000 | 24+62 |
| Happy | 1800 | 4700 | 335+19 |

combinators described in this article, built upon three underlying combinator sets; Hutton's set described in section 2, Augustsson's which are part of the HBC system (Augustsson, 1994) and a set derived from Hutton's based on the type:

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

data Maybe a = None | Just a
```

The final two parsers are built using parser generators. The first is constructed using the Ratatosk tool developed by Mogensen (1993) and the second is built using the Happy package of Gill and Marlow (1995). So that the comparisons are fair, all implementations use the same hand-written lexical analyser and pretty-printer. Reduction counts and cell-usage are gathered from implementations running under Gofer 2.30a. The parser generators were compiled using Glasgow Haskell version 0.23. All measurements were made on a SPARC-10 workstation with 64Mb of memory running SUNOS version 4.

The table provides three figures:

- The number of reductions is a measure of how long the parsers took to parse and pretty-print an expression containing 24 tokens from a representative set of precedence levels.
- The number of cells measures the turn-over in heap cells for the same 24-token expression.
- The turn-around is a measure of how long it takes to create a usable parser from a program script of grammar file. In the case of a combinator implementation this merely involves loading the script into the Gofer system. For the parser generators two figures are given: the first is the time required to generate the parser script from the grammar file, the second is the time taken to load the parser into Gofer.

The figures show that Happy generates by far the most efficient parser, but at the cost of a large turn-around time. Thus, Happy parsers are good for final production, but would not be practical, at present, for rapid software development. Even where the grammar remains static, it takes approximately half a minute on an unloaded machine for a generated parser to be loaded into Gofer.

The Ratatosk system generates parsers more rapidly. It also accepts the widest class of grammars, but the resulting parsers are bigger, mainly due to a larger state space. As with Happy, the generated parsers do not load quickly.

Both the Happy and Ratatosk parsers place strains on the language implementation —in Gofer, the type checker is particularly badly hit. Indeed, a new version of the Gofer system had to be compiled especially to handle them. Under Glasgow Haskell, prodigious heap spaces are required to compile the machine-generated parsers. The Happy parser required a 60 Mbyte heap and the Ratotosk parser could not be compiled even with an 80 Mbyte heap.

The combinator-based parsers are at least an order of magnitude slower than their machine generated counterparts and turn-over far more memory. However, the turn-around from writing a grammar to getting a working parser is rapid. In prototype situations where the grammar is often changed, they provide a significant advantage. They also provide templates for common grammatical constructions reducing the likelihood of errors in transcription.

## 8 Conclusions and future work

There is still scope for some refinement of the combinators. At present, they use the `literal` parser to match tokens. In a realistic example, each token may carry extra information such as line number and position. It would be a simple matter to extend the combinators to accept a projection function parameter to extract this information.

Many modern functional languages allow the programmer to define new operators and to assign associativity and precedence to them. A parser for such languages must defer disambiguation until this information is available—possibly not until the end of a script. Although the new combinator set cannot handle these languages directly, one approach would be to adopt a multi-pass parser. The first parse would gather the operator fixity, precedence and associativity, and from this information would construct a parser for the script using the expression combinators.

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 expression syntax was remarkable in that it necessitated the definition of only one extra combinator.

## Acknowledgements

I would like to thank David Barnes and Simon Thompson of the University of Kent for their comments and suggestions on the early drafts of this article. I would also like to thank the anonymous referees for their astute observations and sensible recommendations.

The source for the combinators and *cparen* tool are available via anonymous ftp from `ftp.ukc.ac.uk` in the directory `/pub/sah`.

## References

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

Appel, A. W. (1992) *Compiling with Continuations.* Combridge University Press.

Augustsson, L. (1994) The chalmers haskell compiler: hbc/hbi. (Available via anonymous ftp from `ftp.cs.chalmers.se`.)

Burge, W. H. (1975) *Recursive Programming Techniques.* Addison Wesley.

Fairbairn, J. (1986) Making form follow function. Technical Report 89, University of Cambridge, Computer Laboratory, June.

Gill, A. and Marlow, S. (1995) *Happy Manual.* (Available via anonymous ftp from `ftp.dcs.gla.ac.uk` in `pub/haskell/happy`.)

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

Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming* **2**(3) July.

Johnson, S. C. (1978) *Yacc: Yet Another Compiler Compiler*, July. UNIX on-line documentation.

Jones, M. P. (1991) *Introduction to Gofer 2.20.* (Available via anonymous ftp from *nebula.cs.yale.edu.*)

Lesk, M. E. and Schmidt, E. (1975) *Lex – A Lexical Analyser Generator*, July. UNIX on-line documentation.

Mogensen, T. (1993) *Ratatosk: A Parser Generator and Scanner Generator for Gofer.* (Avaliable via anonymous ftp from `ftp.diku.dk`.)

Peyton Jones, S. L. (1985) Yacc in sasl – an exercise in functional programming. *Software Practice and Experience* **15**(8):807–820.

Turner, D. A. (1986) An overview of Miranda. *SIGPLAN Notices* December.

Wadler, P. (1985) How to replace failure by a list of successes. In: *Lecture Notes in Computer Science 201.* Springer-Verlag.