Parser Combinators for Ambiguous Left-Recursive Grammars

Richard A. Frost[1], Rahmatullah Hafiz[1], and Paul Callaghan[2]

[1] School of Computer Science, University of Windsor, Canada
richard@uwindsor.ca
[2] Department of Computer Science, University of Durham, U.K.

Abstract. Parser combinators are higher-order functions used to build parsers as executable specifications of grammars. Some existing implementations are only able to handle limited ambiguity, some have exponential time and/or space complexity for ambiguous input, most cannot accommodate left-recursive grammars. This paper describes combinators, implemented in Haskell, which overcome all of these limitations.

Keywords: Parser combinators, ambiguity, left recursion, functional programming, natural-language parsing.

1 Introduction

In functional programming, higher order functions called parser combinators can be used to build basic parsers and to construct complex parsers for nonterminals from other parsers. Parser combinators allow parsers to be defined in an embedded style, in code which is similar in structure to the rules of the grammar. As such, implementations can be thought of as executable specifications with all of the associated advantages. In addition, parser combinators use a top-down parsing strategy which facilitates modular piecewise construction and testing.

Parser combinators have been used extensively in the prototyping of compilers and processors for domain-specific languages such as natural language interfaces to databases, where complex and varied semantic actions are closely integrated with syntactic processing. However, simple implementations have exponential time complexity and inefficient representations of parse results for ambiguous inputs. Their inability to handle left-recursion is a long-standing problem. These shortcomings limit the use of parser combinators especially in applications with large and complex grammars.

Various techniques have been developed by others to address some of these shortcomings. However, none of that previous work has solved all of them.

The parser combinators that we present here are the first which can be used to create executable specifications of ambiguous grammars with unconstrained left-recursion, which execute in polynomial time, and which generate compact polynomial-sized representations of the potentially exponential number of results for highly ambiguous input.

P. Hudak and D.S. Warren (Eds.): PADL 2008, LNCS 4902, pp. 167–181, 2008.

The combinators are based on an algorithm developed by Frost, Hafiz and Callaghan (2007). That algorithm combines memoization with existing techniques for dealing with left recursion. The memotables are modified to represent the potentially exponential number of parse trees in a compact polynomial sized representation using a technique derived from (Kay 1980) and (Tomita 1986). A novel technique is used to accommodate indirect as well as direct left recursion.

This paper has three objectives: 1) To make the algorithm of Frost, Hafiz and Callaghan known to a wider audience beyond the Computational Linguistics community. In particular by the functional and logic programming communities both of which have a long history of creating parsers as executable specifications (using parser combinators and Definite Clause Grammars respectively), 2) to introduce a library of parser combinators for immediate use by functional programmers, and 3) to illustrate how a declarative language facilitates the incremental implementation of a complex algorithm. Note that extension to include semantics will be straightforward, and that this work can be seen as an important step towards combinators that support general attribute grammars.

As example use of our combinators, consider the following ambiguous grammar from Tomita (1986). The nonterminal `s` stands for sentence, `np` for nounphrase, `vp` for verbphrase, `det` for determiner, `pp` for prepositional phrase, and `prep` for preposition. This grammar is left recursive in the rules for `s` and `np`.

```
s    ::= np vp | s pp      np   ::= noun  | det noun | np pp
pp   ::= prep np           vp   ::= verb np
det  ::= "a"   | "the"     noun ::= "i"   | "man" | "park" | "bat"
verb ::= "saw"             prep ::= "in"  | "with"
```

The Haskell code below defines a parser for the above grammar using our combinators `term`, `<+>`, and `*>`.

```
data Label = S | ... | PREP
s    = memoize S    $ np *> vp <+> s *> pp
np   = memoize NP   $ noun <+> det *> noun <+> np  *> pp
pp   = memoize PP   $ prep *> np
vp   = memoize VP   $ verb *> np
det  = memoize DET  $ term "a"  <+> term "the"
noun = memoize NOUN $ term "i"<+>term "man"<+>term "park" <+> term "bat"
verb = memoize VERB $ term "saw"
prep = memoize PREP $ term "in" <+> term "with"
```

The next page shows the "prettyprinted" output when the parser function `s` is applied to "`i saw a man in the park with a bat`". The compact representation corresponds to the several ways in which the whole input can be parsed as a sentence, and the many ways in which subsequences of it can be parsed as nounphrases etc. For example, the entry for `NP` shows that nounphrases were identified starting at positions 1, 3, 6, and 9. Some of which were identified as spanning positions 3 to 5, 8, and 11. Two were found spanning positions 3 to 11. The first of which consists of a `NP` spanning 3 to 5 followed by a `PP` spanning 5 to 11. (We define a span from `x` to `y` as consuming terminals from `x` to `y - 1`.)

```
NOUN [1 ->[2 ->[Leaf "i"]]
     ,4 ->[5 ->[Leaf "man"]]
     ,7 ->[8 ->[Leaf "park"]]
     ,10->[11->[Leaf "bat"]]]
DET  [3 ->[4 ->[Leaf "a"]]
     ,6 ->[7 ->[Leaf "the"]]
     ,9 ->[10->[Leaf "a"]]]
NP   [1 ->[2 ->[SubNode NOUN (1,2)]]
     ,3 ->[5 ->[Branch [SubNode DET  (3,4) , SubNode NOUN (4,5)]]
          ,8 ->[Branch [SubNode NP   (3,5) , SubNode PP   (5,8)]]
          ,11->[Branch [SubNode NP   (3,5) , SubNode PP   (5,11)]
               ,Branch [SubNode NP   (3,8) , SubNode PP   (8,11)]]]
     ,6 ->[8 ->[Branch [SubNode DET  (6,7) , SubNode NOUN (7,8)]]
          ,11->[Branch [SubNode NP   (6,8) , SubNode PP   (8,11)]]]
     ,9 ->[11->[Branch [SubNode DET  (9,10), SubNode NOUN (10,11)]]]]
PREP [5 ->[6 ->[Leaf "in"]]
     ,8 ->[9 ->[Leaf "with"]]]
PP   [8 ->[11->[Branch [SubNode PREP (8,9), SubNode NP (9,11)]]]
     ,5 ->[8 ->[Branch [SubNode PREP (5,6), SubNode NP (6,8)]]
          ,11->[Branch [SubNode PREP (5,6), SubNode NP (6,11)]]]]
VERB [2 ->[3 ->[Leaf "saw"]]]
VP   [2 ->[5 ->[Branch [SubNode VERB (2,3), SubNode NP (3,5)]]
          ,8 ->[Branch [SubNode VERB (2,3), SubNode NP (3,8)]]
          ,11->[Branch [SubNode VERB (2,3), SubNode NP (3,11)]]]]
S    [1 ->[5 ->[Branch [SubNode NP   (1,2), SubNode VP (2,5)]]
          ,8 ->[Branch [SubNode NP   (1,2), SubNode VP (2,8)]
               ,Branch [SubNode S    (1,5), SubNode PP (5,8)]]
          ,11->[Branch [SubNode NP   (1,2), SubNode VP (2,11)]
               ,Branch [SubNode S    (1,5), SubNode PP (5,11)]
               ,Branch [SubNode S    (1,8), SubNode PP (8,11)]]]]
```

Parsers constructed with our combinators have $O(n^3)$ worst case time complexity for non-left-recursive ambiguous grammars (where n is the length of the input), and $O(n^4)$ for left recursive ambiguous grammars. This compares well with $O(n^3)$ limits on standard algorithms for CFGs such as Earley-style parsers (Earley 1970). The increase to n^4 is due to expansion of the left recursive nonterminals in the grammar. Experimental evidence suggests that typical performance is closer to $O(n^3)$, possibly because few subparsers are left recursive and hence the $O(n^3)$ term predominates. Experimental evaluation involved four natural-language grammars from (Tomita 1986), four variants of an abstract highly-ambiguous grammar, and a medium-size natural-language grammar with 5,226 rules. The potentially-exponential number of parse trees for highly-ambiguous input are represented in polynomial space as in Tomita's algorithm.

We begin with background material followed by a detailed description of the Haskell implementation. Experimental results, related work, and conclusions are given in sections 4, 5 and 6. Formal proofs of termination and complexity, and the code of the initial Haskell implementation, are available at:

`cs.uwindsor.ca/~richard/PUBLICATIONS/APPENDICES_HASKELL.html`

2 Background

2.1 Top Down Parsing and Memoization

Top-down parsers search for parses using a top-down expansion of the grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. Simple implementations do not terminate for left-recursive grammars, and have exponential time complexity with respect to the length of the input for non-left-recursive ambiguous grammars.

The problem of exponential time complexity in top-down parsers constructed as sets of mutually-recursive functions has been solved by Norvig (1991). His technique is similar to the use of dynamic programming and state sets in Earley's algorithm (1970), and tables in the CYK algorithm of Cocke, Younger and Kasami. The key idea is to store results of applying a parser `p` at position j in a memotable and to reuse results whenever the same situation arises. It can be implemented as a wrapper function `memoize` which can be applied selectively to component parsers.

2.2 The Need for Left Recursion

Left-recursion can be avoided by transforming the grammar to a weakly equivalent non-left-recursive form (i.e. to a grammar which derives the same set of sentences, but does not generate the same set of parse trees). Such transformation has two disadvantages: 1) it is error prone, especially for non-trivial grammars, and 2) the loss of some parses (as illustrated in the example in (Frost et al 2007)) complicates the integration of semantic actions, especially in NLP.

2.3 An Introduction to Parser Combinators

The details in this description have been adapted to our approach, and are limited to recognition. We extend the technique to parsers later. Assume that the input is a sequence of tokens *input*, of length *#input* the members of which are accessed through an index j. Recognizers are functions which take an index j as argument and which return a set of indices. Each index in the result set corresponds to a position at which the parser successfully finished recognizing a sequence of tokens that began at position j. An empty result set indicates that the recognizer failed to recognize any sequence beginning at j. The result for an ambiguous input is a set with more than one element. This use of *indices* instead of the more conventional subsequence of input is a key detail of the approach: we need the positions to index into the memotables.

A recognizer `term 'x'` for a terminal `'x'` is a function which takes an index j as input, and if j is less than *#input* and if the token at position j in the input corresponds to the terminal `'x'`, it returns a singleton set containing $j + 1$, otherwise it returns the empty set. The `empty` recognizer is a function which always succeeds returning a singleton set containing the current position.

A recognizer for alternation `p|q` is built by combining recognizers for `p` and `q`, using the combinator `<+>`. When the composite recognizer is applied to index j, it applies `p` to j, applies `q` to j, and subsequently unites the resulting sets.

A composite recognizer corresponding to a sequence of recognizers `p q` on the right hand side of a grammar rule, is built by combining those recognizers using the parser combinator `*>`. When the composite recognizer is applied to an index j, it first applies `p` to j, then it applies `q` to each index in the set of results returned by `p`. It returns the union of these applications of `q`. The combinators `term`, `empty`, `<+>` and `*>` are defined (in functional pseudo code) as follows:

$$term \quad t \quad j = \begin{cases} \{\} & , j \geq \#input \\ \{j+1\} & , j^{th} \text{ element of } input = t \\ \{\} & , \text{otherwise} \end{cases}$$

$$empty \quad j = \{j\}$$

$$(p \quad \texttt{<+>} \quad q) \quad j = (p \quad j) \quad \cup \quad (q \quad j)$$

$$(p \quad \texttt{*>} \quad q) \quad j = \bigcup(map \quad q \quad (p \quad j))$$

The combinators can be used to define composite mutually-recursive recognizers. For example, the grammar `s ::= 'x' s s | empty` can be encoded as `s = (term 'x' *> s *> s) <+> empty`. Assuming the input is "`xxxx`", then:

```
(empty <+> term 'x')   2 => {2,3}
(term 'x' *> term 'x') 1 => {3}

s 0  => {4, 3, 2, 1, 0}
```

The last four values in the result for `s 0` correspond to proper prefixes of the input being recognized as an `s`. The result `4` corresponds to the case where the whole input is recognized as an `s`. Note that we have used sets in this explanation to simplify later development of the combinators.

3 The Combinators

3.1 Preliminaries

The actual implementation of the combinators `*>` and `<+>` for plain recognizers in Haskell is straightforward, and makes use of a library for Sets of `Ints`. An excerpt is given below. In this fragment and the ones that follow, we make some simplifications to ease presentation of the key details. Full working code is available from the URL given in Section 1. We omit details of how we access the input throughout this paper, treating it as a constant value.

```
type Pos    = Int
type PosSet = IntSet
type R      = Pos -> PosSet
```

```
(<+>)        :: R -> R -> R
p <+> q      = \r -> union (p r) (q r)
(*>)         :: R -> R -> R
p *> q       = \r -> unions $ map q $ elems $ p r
parse        :: R -> PosSet
parse p      = p 0
```

In the following we develop the combinators `*>` and `<+>` incrementally, by adding new features one at a time in each subsection. We begin with memoization, then direct left recursion, then indirect left recursion, then parsing (to trees). The revised definitions accompany new types which indicate a particular version of the combinator. The modifications to `<+>` are omitted when they are reasonably straightforward or trivial.

3.2 Memoizing Recognizers

We modify the combinators so that a memotable is used during recognition. At first the table is empty. During the process it is updated with an entry for each recognizer that is applied to a position. Recognizers to be memoized are labelled with values of a type chosen by the programmer. These labels usually appear as node labels in resulting parse trees, but more generality is possible, e.g. to hold limited semantic information. We require only that these labels be enumerable, i.e. have a unique mapping to and from `Ints`, a property that we use to make table lookups more efficient by converting label occurrences internally to `Ints` and using the optimized `IntMap` library.

The memotable is a map of memo label and start position to a result set. The combinators are lifted to the monad level and the memotable is the state that is threaded through the parser operations, and consulted and/or updated during the `memoize` operation. We use a standard state monad:

```
type ILabel  = Int
type RM memoLabel = Pos -> StateM (State memoLabel) PosSet
data StateM s t = State {unState:: s -> (t, s)}
type State nodeName = IntMap (IntMap PosSet)
(*>) :: RM l -> RM l -> RM l
p *> q = \r -> do end_p  <- p r
                  end_qs <- mapM q (elems end_p)
                  return $ unions end_qs
```

The `memoize` function makes the decision regarding reuse of results. It is implemented as a "wrapper" around other parsers, hence any sub-parser can be memoized. The function checks whether an entry exists in the memotable for the given parser label and position, returning the stored result if yes, otherwise it runs the parser and stores the results before returning them. `update_table` adds the new information to the table. Note that the update effect is to "overwrite" the previous information. The `insertWith`...`insert` combination merges into the outer table (`insertWith`) a new inner table that discards (`insert`) any previous entry for that label and start position. This is necessary to update the stored information as (left) recursion unwinds (see section 3.3):

```
memoize :: Enum l => l -> RM l -> RM l
memoize e_name parser pos
 = do mt <- get
      case lookupT i_name pos mt of
        Just res -> return res
        Nothing  -> do res <- parser pos
                       modify (update_table res)
                       return res
   where
     i_name = fromEnum e_name
     update_table :: PosSet -> State l -> State l
     update_table res = insertWith (\_ prev -> insert pos res prev)
                                       i_name (singleton pos res)
```

3.3 Accommodating Direct Left Recursion

To accommodate direct left recursion, we use “left-rec counts” c_{ij} denoting the number of times a recognizer r_i has been applied to an index j. For non-left-recursive recognizers c_{ij} will be at most one. For left-recursive recognizers, c_{ij} is increased on recursive descent. Application of a recognizer r_i to an index j is curtailed whenever c_{ij} exceeds the number of unconsumed tokens of the input plus 1. At this point no parse is possible (other than spurious parses from cyclic grammars — which we want to curtail anyway.) As an illustration, consider the following portion of the search space being created during the parse of two remaining tokens on the input (where N, P and Q are nodes in the parse search space corresponding to nonterminals. A, B and C are nodes corresponding to terminals or nonterminals):

```
           N
         /   \
        N     A
       /  \
      N    B
     /  \
    P    C
   /
  Q
 /
N
```

The last call of the parser for N should be curtailed owing to the fact that, irrespective of what A, B, and C are, either they must require at least one input token, or else they must rewrite to empty. If they all require a token, then the parse cannot succeed. If any rewrite to empty, then the grammar is cyclic (N is being rewritten to N). The last call should be curtailed in either case.

Curtailing a parse when a branch is longer than the length of the remaining input is incorrect as this can occur in a correct parse if recognizers are rewritten into other recognizers which do not have “token requirements to the right”. Also, we curtail the recognizer when the left-rec count exceeds the number of unconsumed tokens *plus 1*. The plus 1 is necessary for the case where the recognizer rewrites to empty on application to the end of the input.

This curtailment test is implemented by passing a “left-rec context” down the invocation tree. The context is a frequency table of calls to the memoized parsers encountered on the current chain.

```
type L_Context = [(ILabel, Int)]
type LRM memolabel = L_Context -> RM memolabel
```

Only `*>` and `memoize` need to be altered beyond propagating the information downwards. `memoize` checks before expanding a parser `p` to see if it has been called more than there are tokens left in the input, and if so, returns an empty result, otherwise continues as before though passing a context updated with an extra call to `p`. The alteration to `*>` controls what context should be passed to `q`: the current context should only be passed when `p` has consumed no tokens, i.e. has done nothing to break the left-recursive chain.

```
(*>) :: LRM l -> LRM l -> LRM l
p *> q = \ctxt r -> do end_p <- p ctxt r
                       let pass_ctxt e | e == r    = ctxt
                                       | otherwise = []
                       end_qs<-mapM (\e-> q (pass_ctxt e) e)(elems end_p)
                       return $ unions end_qs
memoize :: Enum l => l -> LRM l -> LRM l
memoize e_name p ctxt pos
 = do mt <- get
      case lookupT i_name pos mt of
           Just res -> return res
           Nothing  | depth_cutoff i_name ctxt >
                                  (length_input - pos + 1) -> empty
                    | otherwise -> do
                                     .. p (increment i_name ctxt) pos ..
   where i_name = fromEnum e_name
         depth_cutoff i e = case lookup i e of Nothing -> 0
                                               Just fe -> fe
```

Notice what happens when unwinding the left-recursive calls. At each level, `memoize` runs the parser and adds the results to the table for the given label and start position. This table update, as mentioned earlier, overwrites previous information at the start position, and therefore the table always contains the “best results so far”. Note that the algorithm accommodates cyclic grammars. It terminates for such grammars with information being stored in the memotable which can be subsequently used to identify cycles.

3.4 Accommodating Indirect Left Recursion

We begin by illustrating how the method above may return incomplete results for grammars containing indirect left recursion. Consider the following grammar, and subset of the search space, where the left and right branches represent the expansions of the first two alternate right-hand-sides of the rule for the nonterminal `S`, applied to the same position on the input:

```
S ::= S ..| Q | P | x                   S
P ::= S ..                         /    +    \
Q ::= T                         S ..          Q
T ::= P                         |             |
                                S ..          T
                                |             |
                                P             P
                                |             |
                                S ..          S ..
                                |
                            curtail S
```

Suppose that the branch for the left alternative is expanded before the right branch during the search, and that the left branch is curtailed due to the left-rec count for `S` exceeding its limit. The results stored for `P` on recursive ascent of the left branch is an empty set. The problem is that the later call of `P` on the right branch should not reuse the empty set of results from the first call of `P` as they are incomplete with respect to the position of `P` on the right branch (i.e. if `P` were to be reapplied to the input in the context of the right branch, the results would not necessarily be an empty set.) This problem is a result of the fact that, on the left branch, `S` caused curtailment of the results for `P` as well as for itself.

Our solution to this problem is as follows: 1) Pass left-rec contexts downwards as in subsection 3.3. 2) Generate the reasons for curtailment when computing results. For each result we need to know if the subtrees contributing to it have been curtailed through any left-rec limit, and if so, which recognizers caused the curtailment. 3) Store results in the memotable together with a subset of the current left-rec context corresponding to those recognizers that caused the curtailment at the current position, and 4) Whenever a stored result is being considered for reuse, the left-rec-context of that result is compared with the left-rec-context of the current node in the parse space. The result is only reused if, for each recognizer in the left-rec context of the result, the left-rec-count is smaller than or equal to the left-rec-count in the current context. This ensures that a result stored for application `P` of a recognizer at index j is only reused by a subsequent application `P'` of the same recognizer at the same position, if the left-rec context for `P'` would constrain the result more, or equally as much, as it had been constrained by the left-rec context for `P` at j. If there were no curtailment, the left-rec context of a result would be empty and that result can be reused anywhere irrespective of the current left-rec context.

This strategy extends the recognizer return type to include a set of labels that caused curtailments during that parse. Note that we only collect information about curtailment for the current position, so only collect results from `q` in the case where `p` consumed no input, i.e. where the endpoint of `p` is the same as the starting position.

```
type CurtailingNTs  = IntSet
type UpResult       = (CurtailingNTs, PosSet)
type State nodeName = IntMap (IntMap (L_Context,UpResult))
type CLRM memoLabel = L_Context -> Pos
```

```
                                 -> StateM (State memoLabel) UpResult

(*>) :: CLRM l -> CLRM l -> CLRM l
p *> q = \ctxt r -> do (cut,end_p) <- p ctxt r
                       let pass_ctxt e | e == r    = ctxt
                                       | otherwise = []
                           merge_cuts e prev new
                                        | e == r    = union prev new
                                        | otherwise = prev
                           join (prev_cut, prev_result) e
                            = do (new_cut, result) <- q (pass_ctxt e) e
                                 return ( merge_cuts e prev_cut new_cut
                                        , union prev_result result )
                       end_qs <- foldM join (cut, empty) end_p
                       return end_qs
```

The function <+> is modified to merge information from the subparsers:

```
(<+>) :: CLRM l -> CLRM l -> CLRM l
(p <+> q) inp cc  = do (cut1,m) <- p inp cc
                       (cut2,n) <- q inp cc
                       return ( union cut1 cut2 , union m n )
```

When retrieving results, memoize compares the current context with the pruned stored context. Reuse is only allowed if every label in the stored context appears in the current context and is not less constrained in the current context. Otherwise, the parser is run further in the current context to compute the results that were curtailed (and hence missing) in the earlier call.

```
pruneContext :: CurtailingNTs -> L_Context -> L_Context
pruneContext rs ctxt = [nc | nc@(n,c) <- ctxt, n ‘member‘ rs]
canReuse :: L_Context -> L_Context -> Bool
canReuse current stored
 = and [ or [ cc >= sc | (cn,cc) <- current, sn == cn ]
       | (sn,sc) <- stored ]
```

3.5 Building Parse Trees

Turning a recogniser into a parser is straightforward. A set of endpoints now becomes a map of endpoints to lists of trees that end at that point. The memotable type is altered to contain this new information: it stores tree results with their curtail set and a relevant L_context. The tree type is shown below.

```
data Tree l = Empty | Leaf Token | Branch [Tree l]  | ...
type ParseResult memoLabel = [(Int, [Tree memoLabel])]
type UpResult memoLabel = (CurtailingNTs, ParseResult memoLabel)
data Stored memoLabel = Stored { s_stored  :: UpResult memoLabel
                               , s_context :: L_Context
                               , s_results :: [(Int, Tree memoLabel)]}
type State memoLabel = IntMap (IntMap (Stored memoLabel))
type P memoLabel
    = L_Context -> Pos -> StateM (State memoLabel) (UpResult memoLabel)
```

term parsers now return a list of leaf values with suitable endpoints. The empty parser returns an empty tree. Alternative parses from <+> are merged by appending together the lists of trees ending at the same point. The maps are held in ascending endpoint order to give this operation an $O(n)$ cost.

Sequences require *> to join all results of p with all results of q, forming new branch nodes in the tree, and merging the resulting maps together. The former is achieved with addP which combines a particular result from p with all subsequent results from q, and with addToBranch which merges lists of trees from both the left and the right into a new list of trees. Notice that this operation is a cross-product: it must pair each tree from the left with each tree on the right. Tree merging or packing is done by concatenating results at the same endpoint.

```
addP :: [[Tree l]] -> ParseResult l -> ParseResult l
addP left_result right_output
 = [ (re , addToBranch left_result right_results)
                       | (re , right_results) <- right_output ]
addToBranch :: [[Tree l]] -> [[Tree l]] -> [[Tree l]]
addToBranch lts rts = [r ++ l | l <- lts, r <- rts]
```

The memoize function handles the rest of tree formation, both labelling and introducing sharing to avoid an exponential number of trees. Labelling attaches the memo label to the tree result. Sharing replaces the computed list of results with a single result that contains sufficient information to find the original list which will be stored in the memo table. This single result is then returned to higher parsers as a 'proxy' for the original list. To avoid recomputation, we also store the proxy in the memotable to be retrieved by subsequent parser lookups. This technique avoids exponential blow-up of the number of results propagated by parsers. An example of the resulting compact representation of parse trees has been given in Section 1.

It is important to note that the combinators support addition of semantics. The extension from trees to semantic values is straightforward via an "applicative functor" interface, e.g. with operator (<*>) :: P (a -> b) -> P a -> P b. A monadic interface may also be defined.

Test Set	#Input	#Parses	Our method				Tomita's method			
			G1	G2	G3	G4	G1	G2	G3	G4
Tomita's sent. set 1	19	346	0.02				4.79			
	26	1,464	0.03				8.66			
Tomita's sent. set 2	22	429	0.02	0.02	0.03	0.03	2.80	6.40	4.74	19.93
	31	16,796	0.02	0.02	0.05	0.08	6.14	14.40	10.40	45.28
	40	742,900	0.02	0.06	0.08	0.09	11.70	28.15	18.97	90.85

Fig. 1. Informal comparison with Tomita's results (timings in seconds)

4 Experimental Results

To provide evidence of low-order polynomial costs, we conducted a small scale evaluation with respect to: a) Four practical natural-language grammars (Tomita 1986, Appendix F, pages 171 to 184); b) Four variants of an abstract highly ambiguous grammar from Aho and Ullman (1972); and c) A medium size NL grammar for an Air Travel Information System maintained by Carroll (2003).

Our Haskell program was compiled using the Glasgow Haskell Compiler 6.6. We used a 3GHz/1Gb PC. The performance reported is the "MUT time" as generated in GHC runtime statistics, which is an indication of the time spent doing useful computation. It excludes time spent in garbage collection. We also run with an initial heap of 100Mb and do not fix an upper limit to heap size (apart from the machine's capacity).

Note that the grammars we have tested are inherently expensive owing to the dense ambiguity, and this is irrespective of which parsing method is used.

4.1 Tomita' Grammars

The grammars used were: G1 (8 rules), G2 (40 rules), G3 (220 rules), and G4 (400 rules) (Tomita 1986). We used two sets of input: a) the two most-ambiguous inputs from Tomita's sentence set 1 (page 185 App. G) of lengths 19 and 26 which we parsed with G3 (as did Tomita), and b) three inputs of lengths 4, 10, and 40, with systematically increasing ambiguity, from Tomita's sentence set 2.

Figure 1 shows our times and those recorded by Tomita for his algorithm, using a DEC-20 machine (Tomita 1986, pages 152 and 153 App. D). Clearly there can be no direct comparison against years-old DEC-20 times. However, we note that Tomita's algorithm was regarded, in 1986, as being at least as efficient as Earley's and viable for natural-language parsing using machines that were available at that time. The fact that our algorithm is significantly faster on current PCs supports the claim of viability for NL parsing.

4.2 Highly Ambiguous Abstract Grammars

We defined parsers for four variants of a highly-ambiguous grammar introduced by Aho and Ullman (1972): an unmemoized non-left–recursive parser `s`, a memoized version `ms`, a memoized left–recursive version `sml`, and a memoized left–recursive version with one sub-component also memoized `smml`:

```
s    =                term 'x' *> s  *> s     <+> empty
sm   = memoize SM   $ term 'x' *> sm *> sm    <+> empty
sml  = memoize SML  $ sml  *> sml *> term 'x' <+> empty
smml = memoize SMML $ smml *> (memoize SMML' $ smml *> term 'x') <+>empty
```

We chose these four grammars as they are highly ambiguous. The results in figure 2 show that our algorithm can accommodate massively ambiguous input involving the generation of large and complex parse forests. '*' denotes memory overflow and '-' denotes timings less than 0.005 seconds.

Input Length	No. of parses	s	sm	sml	smml
6	132	1.22	-	-	-
12	208,012	*	-	-	0.02
24	1.289e+12		0.08	0.13	0.06
48	1.313e+26		0.83	0.97	0.80

Fig. 2. Timings for highly-ambiguous grammars (time in seconds)

4.3 ATIS – A Medium Size NL Grammar

Here, we used a modified version of the ATIS grammar and test inputs generated by Carroll (2003), who extracted them from the DARPA ATIS3 treebank.

Our modifications include adding 634 new rules and 66 new nonterminals in order to encode the ATIS lexicon as CFG rules. The resulting grammar consists of 5,226 rules with 258 nonterminals and 991 terminals. Carroll's test input set contains 98 natural language sentences of average length 11.4 words. An example sentence is *"i would like to leave on thursday morning may fifth before six a.m."*.

Times to parse ranged from <1 second for the 5 shortest inputs, to between 12 and 19 seconds for the 5 longest inputs. The average time was 1.88 seconds. Given that our Haskell implementation is in an early stage of development, these results suggest that it may be possible to use our algorithm in applications involving large grammars.

5 Related Work

Our combinators implement the algorithm of Frost, Hafiz and Callaghan (2007). The relationship of that algorithm to work by others on left recursion is discussed in detail in their paper. The following is a brief summary: As in Shiel (1976), the algorithm passes information to parsers which is used in curtailment. The information passed is similar to the cancellation sets used by Nederhof and Koster (1993). The algorithm uses the memoization technique of Norvig (1991) to achieve polynomial complexity with parser combinators, as do Frost (1994), Johnson (1995), and Frost and Hafiz (2006). Note that Ford (2002) has also used memoization in functional parsing, but for constrained grammars. Lickman accommodates left-recursion using fixed points (1995), based on an unpublished idea by Wadler, but he does not address the problem of exponential complexity. Johnson (1995) integrates a technique for dealing with left recursion with memoization. However, the algorithm on which we base our combinators differs from Johnson's $O(n^3)$ approach in the technique that we use to accommodate left recursion. Also, the algorithm facilitates the construction of compact representations of parse results whereas Johnson's appears to be very difficult to extend to do this. As in Frost and Hafiz (2006) the algorithm integrates "left-recursion counts" with memoization, and defines recognizers as functions which take an index as argument and which return a set of indices. The algorithm is an improvement in that it can accommodate indirect as well as direct left recursion and can be used to create parsers in addition to recognizers.

Extensive research has been carried out on parser combinators. A comprehensive overview of that work can be found in (Frost 2006). Our approach owes much to that work. In particular, our combinators and motivation for their use follows from Burge (1975) and Fairburn (1986). Also, we use Wadler's (1985) notion of failure as an empty list of successes, and many of the ideas from Hutton and Meijer (1995) on monadic parsing.

6 Concluding Comments

We have developed a set of parser combinators which allow modular and efficient parsers to be constructed as executable specifications of ambiguous left-recursive grammars. The accommodation of left recursion greatly increases what can be done in this approach, and removes the need for non-expert users to painfully rewrite and debug their grammars to avoid left recursion. We believe that such advantages balance well against any reduction in performance, especially when an application is being prototyped, and in those applications where the additional time required for parsing is not a major factor in the overall time required when semantic processing, especially of ambiguous input, is taken into account. Experimental results indicate that the combinators are feasible for use in small to medium applications with moderately-sized grammars and inputs. The results also suggest that with further tuning, they may be used with large grammars.

Future work includes proof of correctness, analysis w.r.t. grammar size, improvements for very large grammars, detailed comparison with other combinators systems such as Parsec, reduction of reliance on monads in order to support some form of "on-line" computation, comparison with functional implementations of GLR parsers, and extension of the approach to build modular executable specifications of attribute grammars.

Acknowledgements

The authors would like to thank the referees for their careful reviews, constructive criticisms, and helpful suggestions. Richard Frost and Rahmatullah Hafiz thank the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Government of Ontario, respectively, for their support.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling: Parsing. Prentice-Hall, Englewood Cliffs (1972)
2. Burge, W.H.: Recursive Programming Techniques. Addison-Wesley, Reading (1975)
3. Carroll, J.: Efficiency in large-scale parsing systems – parser comparison (2003), `http://informatics.sussex.ac.uk/research/nlp/carroll/elsps.html`
4. Early, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2), 94–102 (1970)

5. Fairburn, J.: Making form follow function: An exercise in functional programming style. Cambridge Comp. Lab. Technical Report No. 89 (1986)
6. Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time. In: ICFP, pp. 36–47 (2002)
7. Frost, R.A.: Realization of natural language interfaces using lazy functional programming. ACM Comp. Surv. 38(4) Article 11 (2006)
8. Frost, R.A.: Using memoization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers. SIGPLAN Notices 29(4), 23–30 (1994)
9. Frost, R.A., Hafiz, R., Callaghan.: Modular and efficient top-down parsing for ambiguous left-recursive grammars. In: Proc. of the Tenth Int. Conf. on Parsing Technologies, pp. 109–120. ACL Press (2007)
10. Frost, R.A., Hafiz, R.: A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. SIGPLAN Notices 42(5), 46–54 (2006)
11. Hutton, G., Meijer, E.: Monadic parser combinators. J. of Functional Programming 8(4), 437–444 (1995)
12. Johnson, M.: Squibs and discussions: Memoization in top-down parsing. Computational Linguistics 21(3), 405–417 (1995)
13. Kuno, S.: The augmented predictive analyzer for context-free languages — its relative efficiency. Communications of the ACM 9(11), 810–823 (1966)
14. Lickman, P.: Parsing With Fixed Points. Master's Th., Oxford (1995)
15. Nederhof, M.J., Koster, C.H.A.: Top-down parsing for left-recursive grammars. Technical Report 93–10. Research Institute for Declarative Systems, Department of Informatics, Faculty of Mathematics and Informatics, Katholieke Universiteit, Nijmegen (1993)
16. Norvig, P.: Techniques for automatic memoization with applications to context-free parsing. Computational Linguistics 17(1), 91–98 (1991)
17. Shiel, B.A.: Observations on context-free parsing. Technical ReportsTR 12–76, Center for Research in Computing Technology, Aiken Computational Laboratory, Harvard University (1976)
18. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Boston, MA (1986)
19. Wadler, P.: How to replace failure by a list of successes. In: Jouannaud, J.-P. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985)