Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars

Richard A. Frost and Rahmatullah Hafiz
School of Computer Science
University of Windsor
Canada
rfrost@cogeco.ca

Paul C. Callaghan
Department of Computer Science
University of Durham
U.K.
P.C.Callaghan@durham.ac.uk

Abstract

In functional and logic programming, parsers can be built as modular executable specifications of grammars, using parser combinators and definite clause grammars respectively. These techniques are based on top-down backtracking search. Commonly used implementations are inefficient for ambiguous languages, cannot accommodate left-recursive grammars, and require exponential space to represent parse trees for highly ambiguous input. Memoization is known to improve efficiency, and work by other researchers has had some success in accommodating left recursion. This paper combines aspects of previous approaches and presents a method by which parsers can be built as modular and efficient executable specifications of ambiguous grammars containing unconstrained left recursion.

1 Introduction

Top-down parsers can be built as a set of mutually-recursive processes. Such implementations are modular in the sense that parsers for terminals and simple non-terminals can be built and tested first. Subsequently, parsers for more complex non-terminals can be constructed and tested. Koskimies (1990), and Nederhof and Koster (1993) discuss this and other advantages of top-down parsing.

In functional and logic programming, top-down parsers can be built using parser combinators (e.g. see Hutton 1992 for a discussion of the origins of parser combinators, and Frost 2006 for a discussion of their use in natural-language processing) and definite clause grammars (DCGs) respectively. For example, consider the following grammar, in which `s` stands for sentence, `np` for nounphrase, `vp` for verbphrase, and `det` for determiner:

```
s    ::= np vp
np   ::= noun | det noun
vp   ::= verb np
det  ::= 'a' | 't'
noun ::= 'i' | 'm' | 'p' | 'b'
verb ::= 's'
```

A set of parsers for this grammar can be constructed in the Haskell functional programming language as follows, where `term`, `` `orelse` ``, and `` `thenS` `` are appropriately-defined higher-order functions called parser combinators. (Note that backquotes surround infix functions in Haskell).

```
s    = np `thenS` vp
np   = noun `orelse` (det `thenS` noun)
vp   = verb `thenS` np
det  = term 'a' `orelse` term 't'
noun = term 'i' `orelse` term 'm'
                `orelse` term 'p'
                `orelse` term 'b'
verb = term 's'
```

Note that the parsers are written directly in the programming language, in code which is similar in structure to the rules of the grammar. As such, the implementation can be thought of as an executable specification with all of the associated advantages. In particular, this approach facilitates modular piecewise construction and testing of component parsers. It also allows parsers to be defined to return semantic values directly instead of intermediate parse results, and parsers to be parameterized in order to accommodate context-sensitive lan-

Proceedings of the 10th Conference on Parsing Technologies, pages 109–120,
Prague, Czech Republic, June 2007.

guages (e.g. Eijck 2003). Also, in functional programming, the type checker can be used to catch errors in parsers attributed with semantic actions.

Parser combinators and DCGs have been used extensively in applications such as prototyping of compilers, and the creation of natural language interfaces to databases, search engines, and web pages, where complex and varied semantic actions are closely integrated with syntactic processing. However, both techniques are based on top-down recursive descent search with backtracking. Commonly used implementations have exponential complexity for ambiguous languages, cannot handle left-recursion, and do not produce compact representations of parse trees. (Note, a left-recursive grammar is one in which a non-terminal `p` derives an expansion `p ..` headed with a `p` either directly or indirectly. Application of a parser for such a grammar results in infinite descent.) These shortcomings limit the use of parser combinators and DCGs especially in natural-language processing.

The problem of exponential time complexity in top-down parsers constructed as sets of mutually-recursive functions has been solved by Norvig (1991) who uses memotables to achieve polynomial complexity. Norvig's 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 basic idea in Norvig's approach is that when a parser is applied to the input, the result is stored in a memotable for subsequent reuse if the same parser is ever reapplied to the same input. In the context of parser combinators, Norvig's approach can be implemented using a function `memoize` to selectively "memoize" parsers.

In some applications, the problem of left-recursion can be overcome by transforming the grammar to a weakly equivalent non-left-recursive form. (i.e. to a grammar which derives the same set of sentences). Early methods of doing this resulted in grammars that are significantly larger than the original grammars. This problem of grammar size has been solved by Moore (2000) who developed a method, based on a left-corner grammar transformation, which produces non-left recursive grammars that are not much larger than the originals. However, although converting a grammar to a weakly-equivalent form is appropriate in some applications (such as speech recognition) it is not appropriate in other applications. According to Aho, Sethi, and Ullman (1986) converting a grammar to non-left recursive form makes it harder to translate expressions containing left-associative operators. Also, in NLP it is easier to integrate semantic actions with parsing when both leftmost and rightmost parses of ambiguous input are being generated. For example, consider the first of the following grammar rules:

```
np   ::= noun  | np conj np
conj ::= "and" | "or"
noun ::= "jim" | "su" | "ali"
```

and its non-left-recursive weakly equivalent form:

```
np  ::= noun np'
np' ::= conj np np' | empty
```

The non-left-recursive form loses the leftmost parses generated by the left-recursive form. Integrating semantic actions with the non-left-recursive rule in order to achieve the two correct interpretations of input such as `["john", "and", "su", "or", "ali"]` is significantly harder than with the left-recursive form.

Several researchers have recognized the importance of accommodating left-recursive grammars in top-down parsing, in general and in the context of parser combinators and DCGs in particular, and have proposed various solutions. That work is described in detail in section 3.

In this paper, we integrate Norvig's technique with aspects of existing techniques for dealing with left recursion. In particular: a) we make use of the length of the remaining input as does Kuno (1965), b) we keep a record of how many times each parser is applied to each input position in a way that is similar to the use of cancellation sets by Nederhof and Koster (1993), c) we integrate memoization with a technique for dealing with left recursion as does Johnson (1995), and d) we store "left-recursion counts" in the memotable, and encapsulate the memoization process in a programming construct called a monad, as suggested by Frost and Hafiz (2006).

Our method includes a new technique for accommodating indirect left recursion which ensures correct reuse of stored results created through curtailment of left-recursive parsers. We also modify the memoization process so that the memotable represents the potentially exponential number of parse trees in a compact polynomial sized form using a technique derived from the chart parsing methods of Kay (1980) and Tomita (1986).

As an example use of our method, consider the following ambiguous left-recursive grammar from Tomita (1985) in which `pp` stands for prepositional phrase, and `prep` for preposition. This grammar is left recursive in the rules for `s` and `np`. Experimental results using larger grammars are given later.

```
s    ::= np vp | s pp
np   ::= noun | det noun | np pp
pp   ::= prep np
vp   ::= verb np
det  ::= 'a' | 't'
noun ::= 'i' | 'm' | 'p' | 'b'
verb ::= 's'
prep ::= 'n' | 'w'
```

The Haskell code below defines a parser for the above grammar, using our combinators:

```
s    = memoize "s"   ((np `thenS` vp)
                `orelse` (s `thenS` pp))
np   = memoize "np" (noun
               `orelse` (det `thenS` noun)
               `orelse` (np  `thenS` pp))
pp   = memoize "pp"    (prep `thenS` np)
vp   = memoize "vp"    (verb `thenS` np)
det  = memoize "det"   (term 'a'
                    `orelse` term 't')
noun = memoize "noun" (term 'i'
                    `orelse` term 'm'
                    `orelse` term 'p'
                    `orelse` term 'b')
verb = memoize "verb" (term 's')
prep = memoize "prep" (term 'n'
                    `orelse` term 'w')
```

The following shows the output when the parser function `s` is applied to the input string `"isamntpwab"`, representing the sentence “I saw a man in the park with a bat”. It is a compact representation of the parse trees corresponding 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. We discuss this representation in more detail in subsection 4.4.

```
apply s "isamntpwab" =>

"noun"
  1  ((1,2),   [Leaf "i"])
  4  ((4,5),   [Leaf "m"])
  7  ((7,8),   [Leaf "p"])
 10  ((10,11), [Leaf "b"])
"det"
  3  ((3,4),   [Leaf "a"])
  6  ((6,7),   [Leaf "t"])
  9  ((9,10),  [Leaf "a"])
"np"
  1 ((1,2), [SubNode ("noun", (1,2))])
  3 ((3,5), [Branch [SubNode ("det", (3,4)),
                     SubNode ("noun",(4,5))]])
    ((3,8), [Branch [SubNode ("np",  (3,5)),
                     SubNode ("pp",  (5,8))]])
    ((3,11),[Branch [SubNode ("np",  (3,5)),
                     SubNode ("pp",  (5,11))],
             Branch [SubNode ("np",  (3,8)),
                     SubNode ("pp",  (8,11))]])
  6 ((6,8), [Branch [SubNode ("det", (6,7)),
                     SubNode ("noun",(7,8))]])
    ((6,11),[Branch [SubNode ("np",  (6,8)),
                     SubNode ("pp",  (8,11))]])
  9 ((9,11),[Branch [SubNode ("det", (9,10)),
                     SubNode ("noun",(10,11))]])
"prep"
  5 ((5,6), [Leaf "n"])
  8 ((8,9), [Leaf "w"])
"pp"
  8 ((8,11),[Branch [SubNode ("prep",(8,9)),
                     SubNode ("np",  (9,11))]])
  5 ((5,8), [Branch [SubNode ("prep",(5,6)),
                     SubNode ("np",  (6,8))]])
    ((5,11),[Branch [SubNode ("prep",(5,6)),
                     SubNode ("np",  (6,11))]])
"verb"
  2 ((2,3), [Leaf "s"])
"vp"
  2 ((2,5), [Branch [SubNode ("verb",(2,3)),
                     SubNode ("np",  (3,5))]])
    ((2,8), [Branch [SubNode ("verb",(2,3)),
                     SubNode ("np",  (3,8))]])
    ((2,11),[Branch [SubNode ("verb",(2,3)),
                     SubNode ("np",  (3,11))]])
"s"
  1 ((1,5), [Branch [SubNode ("np",  (1,2)),
                     SubNode ("vp",  (2,5))]])
    ((1,8), [Branch [SubNode ("np",  (1,2)),
                     SubNode ("vp",  (2,8))],
             Branch [SubNode ("s",   (1,5)),
                     SubNode ("pp",  (5,8))]])
    ((1,11),[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))]]
```

Our method has two disadvantages: a) it has $O(n^4)$ time complexity, for ambiguous grammars, compared with $O(n^3)$ for Earley-style parsers (Earley 1970), and b) it requires the length of the input to be known before parsing can commence.

Our method maintains all of the advantages of top-down parsing and parser combinators discussed earlier. In addition, our method accommodates arbitrary context-free grammars, terminates correctly and correctly reuses results generated by direct and indirect left recursive rules. It parses ambiguous languages in polynomial time and creates polynomial-sized representations of parse trees.

In many applications the advantages of our approach will outweigh the disadvantages. In particular, the additional time required for parsing will not be a major factor in the overall time required when semantic processing, especially of ambiguous input, is taken into account.

We begin with some background material, showing how our approach relates to previous work by others. We follow that with a detailed description of our method. Sections 5, 6, and 7 contain informal proofs of termination and complexity, and a brief description of a Haskell implementation of our algorithm. Complete proofs and the Haskell code are available from any of the authors.

We tested our implementation on four natural-language grammars from Tomita (1986), and on four abstract highly-ambiguous grammars. The results, which are presented in section 8, indicate that our method is viable for many applications, especially those for which parser combinators and definite clause grammars are particularly well-suited.

We present our approach with respect to parser combinators. However, our method can also be implemented in other languages which support recursion and dynamic data structures.

2 Top-Down Backtracking Recognition

Top-down recognizers can be implemented as a set of mutually recursive processes which search for parses using a top-down expansion of the grammar rules defining non-terminals while looking for matches of terminals with tokens on the input. Tokens are consumed from left to right. Backtracking is used to expand all alternative right-hand-sides of grammar rules in order to identify all possible parses. In the following we assume that the input is a sequence of tokens `input`, of length `l_input` the members of which are accessed through an index `j`. Unlike commonly-used implementations of parser combinators, which produce recognizers that manipulate subsequences of the input, we assume, as in Frost and Hafiz (2006), that recognizers are functions which take an index `j` as argument and which return a set of indices as result. Each index in the result set corresponds to the position at which the recognizer 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. Multiple results are returned for ambiguous input.

According to this approach, a recognizer `term_t` for a terminal `t` is a function which takes an index `j` as input, and if `j` is greater than `l_input`, the recognizer returns an empty set. Otherwise, it checks to see if the token at position `j` in the input corresponds to the terminal `t`. If so, it returns a singleton set containing `j + 1`, otherwise it returns the empty set. For example, a basic recognizer for the terminal `'s'` can be defined as follows (note that we use a functional pseudo code throughout, in order to make the paper accessible to a wide audience. We also use a list lookup offset of 1):

```
term_s      = term 's'
where term t  j
 = {}      , if j > l_input
 = {j + 1}, if jth element of input = t
 = {}      , otherwise
```

The `empty` recognizer is a function which always succeeds returning its input index in a set:

```
empty j = {j}
```

A recognizer corresponding to a construct `p | q` in the grammar is built by combining recognizers for `p` and `q`, using the parser combinator `` `orelse` ``. When the composite recognizer is applied to index `j`, it applies `p` to `j`, then it applies `q` to `j`, and subsequently unites the resulting sets.:

```
(p `orelse` q) j = unite (p j) (q j)
```

e.g, assuming that the input is `"ssss"`, then

```
(empty `orelse` term_s) 2 => {2, 3}
```

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 `` `thenS` ``. 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`.

```
(p `thenS` q) j = union (map q (p j))
```

e.g., assuming that the input is `"ssss"`, then

```
(term_s `thenS` term_s) 1 => {3}
```

The combinators above can be used to define composite mutually-recursive recognizers. For example, the grammar `sS ::= 's' sS sS | empty` can be encoded as follows:

```
sS = (term_s `thenS` sS `thenS` sS)
     `orelse` empty
```

Assuming that the input is `"ssss"`, the recognizer `sS` returns a set of five results, the first four corresponding to proper prefixes of the input being recognized as an `sS`. The result `5` corresponds to the case where the whole input is recognized as an `sS`.

```
sS   1  => {1, 2, 3, 4, 5}
```

The method above does not terminate for left-recursive grammars, and has exponential time complexity with respect to `l_input` for non-left-recursive grammars. The complexity is due to the fact that recognizers may be repeatedly applied to the same index during backtracking induced by the operator `` `orelse` ``. We show later how complexity can be improved, using Norvig's memoization technique. We also show, in section 4.4, how the combinators `` term, `orelse`, and `thenS` `` can be redefined so that the processors create compact representations of parse trees in the memotable, with no effect on the form of the executable specification.

3 Left Recursion and Top-Down Parsing

Several researchers have proposed ways in which left-recursion and top-down parsing can coexist:

1) Kuno (1965) was the first to use the length of the input to force termination of left-recursive descent in top-down parsing. The minimal lengths of the strings generated by the grammar on the continuation stack are added and when their sum exceeds the length of the remaining input, expansion of the current non-terminal is terminated. Dynamic programming in parsing was not known at that time, and Kuno's method has exponential complexity.

2) Shiel (1976) recognized the relationship between top-down parsing and the use of state sets and tables in Earley and SYK parsers and developed an approach in which procedures corresponding to non-terminals are called with an extra parameter indicating how many terminals they should read from the input. When a procedure corresponding to a non-terminal `n` is applied, the value of this extra parameter is partitioned into smaller values which are passed to the component procedures on the right of the rule defining `n`. The processor backtracks when a procedure defining a non-terminal is applied with the same parameter to the same input position. The method terminates for left-recursion but has exponential complexity.

3) Leermakers (1993) introduced an approach which accommodates left-recursion through "recursive ascent" rather than top-down search. Although achieving polynomial complexity through memoization, the approach no longer has the modularity and clarity associated with pure top-down parsing. Leermakers did not extend his method to produce compact representations of trees.

4) Nederhof and Koster (1993) introduced "cancellation" parsing in which grammar rules are translated into DCG rules such that each DCG non-terminal is given a "cancellation set" as an extra argument. Each time a new non-terminal is derived in the expansion of a rule, this non-terminal is added to the cancellation set and the resulting set is passed on to the next symbol in the expansion. If a non-terminal is derived which is already in the set then the parser backtracks. This technique prevents non-termination, but loses some parses. To solve this, for each non-terminal `n`, which has a left-recursive alternative 1) a function is added to the parser which places a special token $\underline{n}$ at the front of the input to be recognized, 2) a DCG corresponding to the rule `n ::=` $\underline{n}$ is added to the parser, and 3) the new DCG is invoked after the left-recursive DCG has been called. The approach accommodates left-recursion and maintains modularity. An extension to it also accommodates hidden left recursion which can occur when the grammar contains rules with empty right-hand sides. The shortcoming of Nederhof and Koster's approach is that it is exponential in the worst case and that the resulting code is less clear as it contains additional production rules and code to insert the special tokens.

5) Lickman (1995) defined a set of parser combinators which accommodate left recursion. The method is based on an idea by Philip Wadler in an unpublished paper in which he claimed that fixed points could be used to accommodate left recursion. Lickman implemented Wadler's idea and provided a proof of termination. The method accommodates left recursion and maintains modularity and clarity of the code. However, it has exponential complexity, even for recognition.

6) Johnson (1995) appears to have been the first to integrate memoization with a method for dealing with left recursion in pure top-down parsing. The basic idea is to use the continuation-passing style of programming (CPS) so that the parser computes multiple results, for ambiguous cases, incrementally. There appears to have been no attempt to extend Johnson's approach to create compact representations of parse trees. One explanation for this could

be that the approach is somewhat convoluted and extending it appears to be very difficult. In fact, Johnson states, in his conclusion, that "an implementation attempt (to create a compact representation) would probably be very complicated."

7) Frost and Hafiz (2006) defined a set of parser combinators which can be used to create polynomial time recognizers for grammars with direct left recursion. Their method stores left-recursive counts in the memotable and curtails parses when a count exceeds the length of the remaining input. Their method does not accommodate indirect left recursion, nor does it create parse trees.

Our new method combines many of the ideas developed by others: as with the approach of Kuno (1965) we use the length of the remaining input to curtail recursive descent. Following Shiel (1976), we pass additional information to parsers which is used to curtail recursion. The information that we pass to parsers is similar to the cancellation sets used by Nederhof and Koster (1993) and includes the number of times a parser is applied to each input position. However, in our approach this information is stored in a memotable which is also used to achieve polynomial complexity. Although Johnson (1995) also integrates a technique for dealing with left recursion with memoization, our method differs from Johnson's $O(n^3)$ approach in the technique that we use to accommodate left recursion. Also, our approach facilitates the construction of compact representations of parse trees whereas Johnson's appears not to. In the Haskell implementation of our algorithm, we use a functional programming structure called a monad to encapsulate the details of the parser combinators. Lickman's (1995) approach also uses a monad, but for a different purpose. Our algorithm stores "left-recursion counts" in the memotable as does the approach of Frost and Hafiz (2006). However, our method accommodates indirect left recursion and can be used to create parsers, whereas the method of Frost and Hafiz can only accommodate direct left recursion and creates recognizers not parsers.

4 The New Method

We begin by describing how we improve complexity of the recognizers defined in section 2. We then show how to accommodate direct and indirect left recursion. We end this section by showing how recognizers can be extended to parsers.

4.1 Memoization

As in Norvig (1991) a memotable is constructed during recognition. At first the table is empty. During the process it is updated with an entry for each recognizer r_i that is applied. The entry consists of a set of pairs, each consisting of an index `j` at which the recognizer r_i has been applied, and a set of results of the application of r_i to `j`.

The memotable is used as follows: whenever a recognizer r_i is about to be applied to an index `j`, the memotable is checked to see if that recognizer has ever been applied to that index before. If so, the results from the memotable are returned. If not, the recognizer is applied to the input at index `j`, the memotable is updated, and the results are returned. For non-left-recursive recognizers, this process ensures that no recognizer is ever applied to the same index more than once.

The process of memoization is achieved through the function `memoize` which is defined as follows, where the `update` function stores the result of recognizer application in the table:

```
memoize label r_i j
 = if lookup label j succeeds,
      return memotable result
   else apply r_i to j,
      update table, and return results
```

Memoized recognizers, such as the following, have cubic complexity (see later):

```
msS = memoize "msS"((ms `thenS` msS
                          `thenS` msS)
                    `orelse` empty)
 ms = memoize "ms" term_s
```

4.2 Accommodating direct left recursion

In order to accommodate direct left recursion, we introduce a set of values c_ij denoting the number of times each recognizer r_i has been applied to the index `j`. For non-left-recursive recognizers this "left-rec count" will be at most one, as the memotable lookup will prevent such recognizers from ever being applied to the same input twice. However, for left-recursive recognizers, the left-rec count is increased on recursive descent (owing to the fact that the memotable is only updated on recursive ascent after the recognizer has been applied). Application of a recognizer `r` to an index `j` is failed whenever the left-rec count exceeds the number of unconsumed tokens of the input plus 1. At this point no parse is possible (other than spurious parses which could occur with circular grammars — which we want to reject). As illustration, consider the following branch 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 non-terminals, and `A`, `B` and `C` to terminals or non-terminals):

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

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

Note that failing 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". For example, we cannot fail the parse at `P` or `Q` as these could rewrite to `empty` without indicating circularity. Also note that we curtail the recognizer when the left-rec count exceeds the number of unconsumed tokens *plus 1*. The plus 1 is necessary to accommodate the case where the recognizer rewrites to empty on application to the end of the input.

To make use of the left-rec counts, we simply modify the `memoize` function to refer to an additional table called `ctable` which contains the left-rec counts c_ij, and to check and increment these counters at appropriate points in the computation: if the memotable lookup for the recognizer r_i and the index `j` produces a result, that result is returned. However, if the memotable does not contain a result for that recognizer and that index, c_ij is checked to see if the recognizer should be failed because it has descended too far through left-recursion. If so, `memoize` returns an empty set as result with the memotable unchanged. Otherwise, the counter c_ij is incremented and the recognizer r_i is applied to `j`, and the memotable is updated with the result before it is returned. The function `memoize` defined below, can now be applied to rules with direct left recursion.

```
memoize label r_i j  =
 if lookup label j succeeds
    return memotable results
 else if c_ij > (l_input)-j+1, return {}
     else increment c_ij, apply r_i to j,
          update memotable,
              and return results
```

4.3 Accommodating indirect left recursion

We begin by illustrating how the method described 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 non terminal `S`, applied to the same position on the input:

```
S ::= S then ..| Q | P | x         S
P ::= S then .                   /    \
Q ::= T                    S then ..  Q
T ::= P                    |          |
                           S then ..  T
                           |          |
                           P          P
                           |          |
                         S then..   S then ..
                          |
                        fail S
```

Suppose that the left branch occurs before the right branch, and that the left branch was failed due to the left-rec count for `S` exceeding its limit. The results stored for `P` on recursive ascent of the left branch would be 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 re-applied 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 `S` caused curtailment of the results for `P` as well as for itself. This problem can be solved as follows:

1) Pass left-rec contexts down the parse space. We need additional information when storing and considering results for reuse. We begin by defining the "left-rec-context" of a node in the parse search space as a list of the following type, containing for each index, the left-rec count for each recognizer, including the current recognizer, which have been called in the search branch leading to that node:

```
[(index,[(recog_label,left_rec_count)])]
```

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 a left-rec limits, and if so, which recognizers, at which indices, caused the curtailment. A list of `(recog_label, index)` pairs which caused curtailment in any of the subtrees is returned with the result. `` `orelse` `` and `` `thenS` `` are modified, accordingly, to merge these lists, in addition to merging the results from subtrees.

3) Store results in the memotable together with a subset of the current left-rec context corresponding to those recognizers which caused the curtailment. When a result is to be stored in the memotable for a recognizer `P`, the list of recognizers which caused curtailment (if any) in the subtrees contributing to this result is examined. For each recognizer `S` which caused curtailment at some index, the current left-rec counter for `S` at that index (in the left-rec context for `P`) is stored with the result for `P`. This means that the only part of the left-rec context of a node, that is stored with the result for that node, is a list of those recognizers and current left-rec counts which had an effect on curtailing the result. The limited left-rec context which is stored with the result is called the "left-rec context of the result".

4) Consider results for reuse. Whenever a memotable 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 search. The result is only reused if, for each recognizer and index in the left-rec context of the result, the left-rec-count is smaller than or equal to the left-rec-count of that recognizer and index in the current context. This ensures that a result stored for some 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.

4.4 Extending recognizers to parsers

Instead of returning a list of indices representing successful end points for recognition, parsers also return the parse trees. However, in order that these trees be represented in a compact form, they are constructed with reference to other trees that are stored in the memotable, enabling the explicit sharing of common subtrees, as in Kay's (1980) and Tomita's (1986) methods. The example in section 1 illustrates the results returned by a parser.

Parsers for terminals return a leaf value together with an endpoint, stored in the memotable as illustrated below, indicating that the terminal `"s"` was identified at position 2 on the input:

```
"verb" 2 ((2,3),[Leaf "s"])
```

The combinator `` `thenS` `` is extended so that parsers constructed with it return parse trees which are represented using reference to their immediate subtrees. For example:

```
"np" ......
3 ((3,5),[Branch[SubNode("det", (3,4)),
                 SubNode("noun",(4,5))]])
```

This memotable entry shows that a parse tree for a nounphrase `"np"` has been identified, starting at position `3` and finishing at position `5`, and which consists of two subtrees, corresponding to a determiner and a noun.

The combinator `` `orelse` `` unites results from two parsers and also groups together trees which have the same begin and end points. For example:

```
"np" .........
3 ((3,5),[Branch[SubNode("det", (3,4)),
                 SubNode("noun",(4,5))]])
  ((3,8), [Branch[SubNode("np",  (3,5)),
                 SubNode("pp",   (5,8))]])
  ((3,11),[Branch[SubNode("np",  (3,5)),
                 SubNode("pp",   (5,11))],
          Branch[SubNode("np",   (3,8)),
                 SubNode("pp",   (8,11))]])
```

which shows that four parses of a nounphrase `"np"` have been found starting at position 3, two of which share the endpoint 11.

An important feature is that trees for the same syntactic category having the same start/end points are grouped together and it is the group that is referred to by other trees of which it is a constituent. For example, in the following the parse tree for a `"vp"` spanning positions 2 to 11 refers to a group of subtrees corresponding to the two parses of an `"np"` both of which span positions 3 to 11:

```
"vp" 2 (["np"],[])
 ((2,5), [Branch[SubNode("verb",(2,3)),
                  SubNode("np",  (3,5))]])
 ((2,8), [Branch[SubNode("verb",(2,3)),
                  SubNode("np",  (3,8))]])
 ((2,11),[Branch[SubNode("verb",(2,3)),
                  SubNode("np",  (3,11))]])
```

5 Termination

The only source of iteration is in recursive function calls. Therefore, proof of termination is based on the identification of a measure function which maps the arguments of recursive calls to a well-founded ascending sequence of integers.

Basic recognizers such as `term 'i'` and the recognizer `empty` have no recursion and clearly terminate for finite input. Other recognizers that are defined in terms of these basic recognizers, through mutual and nested recursion, are applied by the `memoize` function which takes a recognizer and an index `j` as input and which accesses the `memotable`. An appropriate measure function maps the index and the set of left–rec values to an integer, which increases by at least one for each recursive call. The fact that the integer is bounded by conditions imposed on the maximum value of the index, the maximum values of the left-rec counters, and the maximum number of left-rec contexts, establishes termination. Extending recognizers to parsers does not involve any additional recursive calls and consequently, the proof also applies to parsers. A formal proof is available from any of the authors.

6 Complexity

The following is an informal proof. A formal proof is available from any of the authors.

We begin by showing that memoized non-left-recursive and left-recursive recognizers have a worst-case time complexities of $O(n^3)$ and $O(n^4)$ respectively, where n is the number of tokens in the input. The proof proceeds as follows: `` `orelse` `` requires $O(n)$ operations to merge the results from two alternate recognizers provided that the indices are kept in ascending order. `` `then` `` involves $O(n^2)$ operations when applying the second recognizer in a sequence to the results returned by the first recognizer. (The fact that recognizers can have multiple alternatives involving multiple recognizers in sequence increases cost by a factor that depends on the grammar, but not on the length of the input). For non-left-recursive recognizers, `memoize` guarantees that each recognizer is applied at most once to each input position. It follows that non-left recursive recognizers have $O(n^3)$ complexity. Recognizers with direct left recursion can be applied to the same input position at most `n` times. It follows that such recognizers have $O(n^4)$ complexity. In the worst case a recognizer with indirect left recursion could be applied to the same input position `n * nt` times where `nt` is the number of nonterminals in the grammar. This worst case would occur when every nonterminal was involved in the path of indirect recursion for some nonterminal. Complexity remains $O(n^4)$.

The only difference between parsers and recognizers is that parsers construct and store parts of parse trees rather than end points. We extend the complexity analysis of recognizers to that of parsers and show that for grammars in Chomsky Normal Form (CNF) (i.e. grammars whose right-hand-sides have at most two symbols, each of which can be either a terminal or a non-terminal), the complexity of non-left recursive parsers is $O(n^3)$ and of left-recursive parsers it is $O(n^4)$. The analysis begins by defining a "parse tuple" consisting of a parser name `p`, a start/end point pair `(s, e)`, and a list of parser names and end/point pairs corresponding to the first level of the parse tree returned by `p` for the sequence of tokens from `s` to `e`. (Note that this corresponds to an entry in the compact representation). The analysis then considers the effect of manipulating sets of parse tuples, rather than endpoints which are the values manipulated by recognizers. Parsers corresponding to grammars in CNF will return, in the worst case, for each start/end point pair (s, e) , $(((e - s) + 1) * t^2)$ parse tuples, where `t` is the number of terminals and non-terminals in the grammar. It follows that there are O(n) parse tuples for each parser and begin/endpoint pair. Each parse tuple corresponds to a bi-partition of the sequence starting at `s` and finishing at `e` by two parsers (possibly the same) from the set of parsers corresponding to terminals and non-terminals in the grammar. It is these parse tuples that are manipulated by `` `orelse` `` and `` `thenS` ``. The only effect on complexity of these operations is to increase the complexity of `` `orelse` `` from O(n) to O(n^2), which is the same as the complexity of `` `thenS` ``. Owing to the fact that the complexity of `` `thenS` `` had the highest degree in the application of a compound recognizer to an index, increasing the complexity of `` `orelse` `` to the same degree in parsing has no effect on the overall complexity of the process.

The representation of trees in the memotable has one entry for each parser. In the worst case, when the parser is applied to every index, the entry has `n` sub-entries, corresponding to `n` begin points. For each of these sub-entries there are up to `n` sub-sub-entries, each corresponding to an end point of the parse. Each of these sub-entries contains O(n) parse tuples as discussed above. It follows that the size of the compact representation is O(n^3).

7 Implementation

We have implemented our method in the pure functional programming language Haskell. We use a monad (Wadler 1995) to implement memoization. Use of a monad allows the memotable to be systematically threaded through the parsers while hiding the details of table update and reuse, allowing a clean and simple interface to be presented to the user. The complete Haskell code is available from any of the authors.

8 Experimental Results

In order to provide evidence of the low-order polynomial costs and scalability of our method, we conducted a limited evaluation with respect to four practical natural-language grammars used by Tomita (Appendix F, 1986) when comparing his algorithm with Earley's, and four variants of an abstract highly ambiguous grammar from Aho and Ullman (1972). Our Haskell program was compiled using the Glasgow Haskell Compiler 6.6 (the code has not yet been tuned to obtain the best performance from this patform). We used a 3GHz/1GB PC in our experiments.

8.1 Tomita's Grammars

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

noun verb det noun (prep det noun)*

The results, which are tabulated in figure 1, show our timings and those recorded by Tomita for his original algorithm and for an improved Earley method, using a DEC-20 machine (Tomita 1986, Appendix D).

Considered by themselves our timings are low enough to suggest that our method is feasible for use in small to medium applications, such as NL interfaces to databases or rhythm analysis in poetry. Such applications typically have modest grammars (no more than a few hundred rules) and are not required to parse huge volumes of input.

Clearly there can be no direct comparison against years-old DEC-20 times, and improved versions of both of these algorithms do exist. However, we point to some relevant trends in the results. The increases in times for our method roughly mirror the increases shown for Tomita's algorithm, as grammar complexity and/or input size increase. This suggests that our algorithm scales adequately well, and not dissimilarly to the earlier algorithms.

8.2 Highly ambiguous abstract grammars

We defined four parsers as executable specifications of four variants of a highly-ambiguous grammar introduced by Aho and Ullman (1972) when discussing ambiguity: an unmemoized non-left–recursive parser `s`, a memoized version `ms`, a memoized left–recursive version `sml`, and a left–recursive version with all parts memoized. (This improves efficiency similarly to converting the grammar to Chomsky Normal Form.):

Input length	No. of Parses	Our algorithm (complete parsing)-PC				Tomitas (complete parsing)-DEC 20				Earleys (recognition only)-DEC 20			
		G1	G2	G3	G4	G1	G2	G3	G4	G1	G2	G3	G4
Input from Tomitas sentence set 1. Timings are in seconds.													
19	346			0.02				4.79				7.66	
26	1,464			0.03				8.66				14.65	
Input from Tomitas sentence set 2. Timings are in seconds.													
22	429	0.00	0.00	0.03	0.05	2.80	6.40	4.74	19.93	2.04	7.87	7.25	42.75
31	16,796	0.00	0.02	0.05	0.09	6.14	14.40	10.40	45.28	4.01	14.09	12.06	70.74
40	742,900	0.03	0.08	0.11	0.14	11.70	28.15	18.97	90.85	6.75	22.42	19.12	104.91

Figure 1: Informal comparison with Tomita/Earley results

```
s    = (term 'a' `thenS` s `thenS` s)
       `orelse` empty
sm   = memoize "sm"
       ((term 'a' `thenS` sm `thenS` sm)
       `orelse` empty)
sml  = memoize "sml"
       ((sml `thenS` sml
             `thenS` term 'a')
       `orelse` empty)
smml = memoize "smml"
       ((smml `thenS`
         (memoize "smml_a"
           (smml `thenS` term 'a')))
       `orelse` empty)
```

We chose these four grammars as they are highly ambiguous. According to Aho and Ullman (1972), `s` generates over `128` billion complete parses of an input consisting of 24 `'a'`'s. Although the left-recursive grammar does not generate exactly the same parses, it generates the same number of parses, as it matches a terminal at the end of the rule rather than at the start.

Input length	No. of parses excluding partial parses	Seconds to generate the packed representation of full and partial parses			
		s	sm	sml	smml
6	132	1.22	0.00	0.00	0.00
12	208,012	out of space	0.00	0.00	0.02
24	1.29e+12		0.08	0.13	0.06
48	1.313e+26		0.83	0.97	0.80

Figure 2: Times to compute forest for n

These results show that our method can accommodate massively-ambiguous input involving the generation of large and complex parse forests. For example, the full forest for `n=48` contains 1,225 choice nodes and 19,600 branch nodes. Note also that the use of more memoization in `smml` reduces the cost of left-rec checking.

9 Concluding Comments

We have extended previous work of others on modular parsers constructed as executable specifications of grammars, in order to accommodate ambiguity and left recursion in polynomial time and space. We have implemented our method as a set of parser combinators in the functional programming language Haskell, and have conducted experiments which demonstrate the viability of the approach.

The results of the experiments suggest that our method is feasible for use in small to medium applications which need parsing of ambiguous grammars. Our method, like other methods which use parser combinators or DCGs, allows parsers to be created as executable specifications which are "embedded" in the host programming language. It is often claimed that this embedded approach is more convenient than indirect methods which involve the use of separate compiler tools such as yacc, for reasons such as support from the host language (including type checking) and ease of use. The major advantage of our method is that it increases the type of grammars that can be accommodated in the embedded style, by supporting left recursion and ambiguity. This greatly increases what can be done in this approach to parser construction, and removes the need for non-expert users to painfully rewrite and debug their grammars to avoid left recursion. We believe such advantages balance well against any reduction in performance, especially when an application is being prototyped.

The Haskell implementation is in its initial stage. We are in the process of modifying it to improve efficiency, and to make better use of Haskell's lazy evaluation strategy (e.g. to return only the first `n` successful parses of the input).

Future work includes proof of correctness, analysis with respect to grammar size, testing with larger natural language grammars, and extending the approach so that language evaluators can be constructed as modular executable specifications of attribute grammars.

Acknowledgements

Richard Frost acknowledges the support provided by the Natural Sciences and Engineering Research Council of Canada in the form of a discovery grant.

References

1. Aho, A. V. and Ullman, J. D. (1972) *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Prentice-Hall.

2. Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques and Tools*. Addison-Wesley Longman Publishing Co.

3. Camarao, C., Figueiredo, L. and Oliveira, R.,H. (2003) Mimico: A Monadic Combinator Compiler Generator. *Journal of the Brazilian Computer Society* 9(1).

4. Earley, J. (1970) An efficient context-free parsing algorithm.*Comm. ACM* 13(2) 94–102.

5. Eijck, J. van (2003) Parser combinators for extraction. In Paul Dekker and Robert van Rooy, editors, *Proceedings of the Fourteenth Amsterdam Colloqium* ILLC, University of Amsterdam. 99–104.

6. Frost, R. A. (2006) Realization of Natural-Language Interfaces using Lazy Functional Programming. *ACM Comput. Surv.* 38(4).

7. Frost, R. A. and Hafiz, R. (2006) A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time. *SIGPLAN Notices* 42 (5) 46–54.

8. Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming* 2 (3) 323–343.

9. Johnson, M. (1995) Squibs and Discussions: Memoization in top-down parsing. *Computational Linguistics* 21(3) 405–417.

10. Kay, M. (1980) Algorithm schemata and data structures in syntactic processing. *Technical Report CSL-80–12,* XEROX Palo Alto Research Center.

11. Koskimies, K. (1990) Lazy recursive descent parsing for modular language implementation. *Software Practice and Experience* 20 (8) 749–772.

12. Kuno, S. (1965) The predictive analyzer and a path elimination technique. *Comm. ACM* 8(7) 453–462.

13. Leermakers, R. (1993) *The Functional Treatment of Parsing*. Kluwer Academic Publishers, ISBN0–7923–9376–7.

14. Lickman, P. (1995) Parsing With Fixed Points. *Master's Thesis*, University of Cambridge.

15. Moore, R. C. (2000) Removing left recursion from context-free grammars. In *Proceedings, 1st Meeting of the North American Chapter of the Association for Computational Linguistics, Seattle, Washington, ANLP–NAACL 2000*. 249–255.

16. Nederhof, M. J. and Koster, C. H. A. (1993) Top-Down Parsing for Left-recursive Grammars. *TechnicalReport* 93–10 Research Institute for Declarative Systems, Department of Informatics, Faculty of Mathematics and Informatics, Katholieke Universiteit, Nijmegen.

17. Norvig, P. (1991) Techniques for automatic memoisation with applications to context-free parsing. *Computational Linguistics* 17(1) 91–98.

18. Shiel, B. A. (1976) Observations on context-free parsing. *Technical Report* TR 12–76, Center for Research in Computing Technology, Aiken Computational Laboratory, Harvard University.

19. Tomita, M. (1986) *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Boston, MA.

20. Wadler, P. (1995) Monads for functional programming, *Proceedings of the Baastad Spring School on Advanced Functional Programming,* ed J. Jeuring and E. Meijer. Springer Verlag LNCS 925.