A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time

Richard A. Frost and Rahmatullah Hafiz
School of Computer Science, University of Windsor
401 Sunset Avenue, Windsor, Ontario Canada ON N9B3P4

rfrost@cogeco.ca

ABSTRACT

Top-down backtracking language processors are highly modular, can handle ambiguity, and are easy to implement with clear and maintainable code. However, a widely-held, and incorrect, view is that top-down processors are inherently exponential for ambiguous grammars and cannot accommodate left-recursive productions. It has been known for many years that exponential complexity can be avoided by memoization, and that left-recursive productions can be accommodated through a variety of techniques. However, until now, memoization and techniques for handling left recursion have either been presented independently, or else attempts at their integration have compromised modularity and clarity of the code.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features — *Control structures; Processors — Parsing;* F.4.2 [**Mathematical Logic and Formal Languages**]:Grammars and Other Rewriting Systems-*grammar types; parsing;I.2.7 [***Artificial Intelligence***]: Natural-language Processing-language models; language parsing and understanding*

Keywords

Top-down parsing, left-recursion, memoization, backtracking, parser combinators.

1 INTRODUCTION

Top-down backtracking language processors have a number of advantages compared to other methods: 1) they are general and can be used to implement ambiguous grammars, 2) they are easy to implement in any language which supports recursion. Associating semantic rules with the recursive functions that implement the syntactic productions rules of the grammar is straightforward, 3) they are highly modular (Koskomies [8]) and components can be tested independently and easily reused, 4) the structure of the code is closely related to the structure of the grammar of the language to be processed, and 5) in functional programming, higher-order functions, called parser combinators, can be defined so that language processors can be implemented as executable specifications of grammars, and in Logic Programming, Definite Clause Grammars (DCGs) can be used to the same effect.

However, a naive implementation of top-down processing, results in the processor repeating much of its work. In the worst case, this results in exponential complexity for highly-ambiguous grammars, even for recognition which is known to be polynomial. In addition, a naive implementation cannot accommodate left-recursive grammar productions as the top-down search would result in infinite descent.

This paper contains an informal description of a new top-down parsing algorithm which accommodates ambiguity and left recursion in polynomial time. The algorithm is described using the notation of set theory, and informal proofs of termination and complexity are provided. Results of an implementation in programming language Haskell are presented, more details of which are available in a technical report available from the School of Computer Science at the University of Windsor [2]. A formal description of the algorithm together with more-detailed proofs of partial correctness, termination and complexity is in preparation.

1.2 Misconceptions

For many years it was assumed that the exponential complexity of top-down recognition of ambiguous sentences was inevitable. However, in 1991, Norvig [13] showed that polynomial complexity for top-down recognizers, built in LISP, could be achieved by use of memoization in which the results of each step of the process are stored in a memo table and made use of by subsequent steps.

It is also widely believed that top-down language processors cannot accommodate left-recursive productions. Using the search terms "top-down" and "left-recursion" on Google returns over 14,000 hits. Review of the results shows the extent to which it continues to be assumed that left-recursion must be eliminated (by rewriting the grammars) before top-down processing can be used. However,

such rewriting is not strictly necessary, and several researchers have proposed ways in which left-recursion can be accommodated:

1. Kuno [9] appears to have been the first to use the length of the input to force termination of left-recursive descent in top-down processing. 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. However, Kuno's method is exponential in the worst case.

2. Shiel [14] noticed the relationship between top-down and Chart parsing 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 rule defining a non-terminal `n` is applied, the value of this extra parameter is partitioned into smaller values which are then 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 is exponential in the worst case.

3. Leermakers [10] has developed a functional approach to memoized parsing which avoids the left-recursion problem through "recursive ascent" rather than a top-down search process. Although maintaining polynomial complexity, the approach compromises modularity and clarity of the code.

4. In earlier work, one of the authors of this paper noticed that rewriting left-recursive recognizers to non-left-recursive form is relatively simple but that rewriting attributed grammars (which contain semantic as well as syntactic rules) can be very difficult. To avoid this difficulty, a method was developed in which non-left-recursive recognizers are used as guards to prevent non-termination of the left-recursive executable attribute grammars which they guard [4]. However, the method is exponential in the worst case.

5. Nederhof and Koster [12] have developed a method called "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. Every time that 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 of left-recursion. However, by itself, it would miss certain parses. Therefore, the method also requires that for each non-terminal N, which has a left-recursive alternative 1) a function is added to the parser which places a special token $\underline{\mathtt{N}}$ at the front of the input to be Recognized, 2) a DCG corresponding to the rule `N ::=` $\underline{\mathtt{N}}$ is added to the parser, and 3) the new DCG is invoked after the left-recursive DCG has been called. The approach accommodates explicit 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.

6. Lickman [11] has developed a technique by which pure functional monadic parser combinators can be modified to accommodate left recursion. The method is based on an idea put forward by Wadler in an unpublished paper in which he claimed that fixed points could be used to accommodate left recursion. Lickman fleshes out Wadler's idea by providing a formal mathematical justification of termination. The method involves constructing a fixed-point combinator for the set monad and then using this function to build an efficient fixed-point combinator for the parser monad (again based on an idea by Wadler). Lickman has also developed a program which automatically generates parsers in the pure functional programming language Haskell from the BNF specification of the grammar. The method accommodates left recursion whilst maintaining modularity and clarity of the code. However, it has exponential complexity.

7. Johnson [6] has developed a method by which memoized top-down parser combinators can accommodate left recursion in the impure-functional programming language Scheme. The basic idea is to use the CPS, continuation-passing style, of programming so that the parser computes multiple results, for ambiguous cases, incrementally. Johnson demonstrates how CPS can be integrated with memoization so that polynomial complexity and termination with left recursion can be achieved with top-down parsing. Surprisingly, Johnson's paper has not been widely cited and his approach does not appear to have been used by others. One explanation for this could be that the approach is somewhat convoluted and extending it to return packed representations of parse trees, as in Tomita's Chart parser [27], could be too complicated.

8. Camarao, Figueiredo, and Oliveiro [1] claim to have built a monadic combinator compiler generator called Mimico which accommodates left recursion. However it does not handle ambiguous grammars.

1.3 Left-Recursion?

There are two reasons why we want to implement left-recursive grammars: firstly, it is often easier to add attribute computations to language processors that implement left-recursive grammars. As a trivial, but illustrative example, consider a processor which converts numbers represented as character strings to their values. The left-recursive formulation is as follows:

```
number ::= digit
           number.VAL = digit.VAL
         | number' digit
           number.VAL = (10 * number'.VAL
                           + digit.VAL)

digit  ::= '0'
           digit.VAL = 0
         |   '1'
           digit.VAL = 1  etc.
```

The right-recursive formulation is more complex and requires an additional attribute.

Secondly, and perhaps more importantly, the advantages of top-down backtracking parsers make them ideally suited for the investigation of compositional theories of natural language. Such investigation is necessary in order to provide more-powerful natural-language interfaces than are currently available. For example, although some NL interfaces can handle various constructs containing transitive verbs, the processing of verb adjuncts is still very limited (e.g "When and with what did Hall discover Phobos?"). There is no widely-accepted linguistic theory which accounts for verb adjuncts. Ideally a compositional Montague-like theory will be developed but this will require an environment in which variations of grammars and semantic rules can be investigated. Because natural language is inherently ambiguous, and both leftmost and rightmost parses are required, the accommodation of left-recursive grammars will facilitate such investigation.

1.4 Overview

The goal of this research is to develop a method by which top-down parsers can accommodate ambiguity and left recursive grammars and be efficient enough for prototyping natural-language processors whilst maintaining modularity and clarity of code, None of the approaches, referred to earlier, have achieved all of these objectives. However, they have shed light on the problem and the solution that we have developed owes much to this earlier work.

The new algorithm uses memoization to improve complexity in a manner similar to that proposed by Norvig [13]. The new idea, introduced for the first time in this paper, is to integrate a bound into the memoization process, which is used to fail a parse branch when that branch contains a cycle introduced through left recursion. This is similar to the approaches proposed by Kuno [9], Shiel [14] and Lickman [11]. However, the new approach allows left-recursive productions to be accommodated whilst achieving polynomial complexity and preserving the modularity and clarity of the processors.

The memotable that is created during the parsing process contains much of the information that is required to construct the potentially exponential number of parse trees. We show how more information can be gathered by memoizing parsers which correspond to each alternative right-hand side of the productions in the grammar. The result is a useful polynomial-sized compact representation of the parse trees.

2 TOP-DOWN PARSING/RECOGNITION

We describe top-down backtracking parsing from a perspective that corresponds to the construction of such parsers as recursive-descent processors. For simplicity, we begin by describing the algorithm with respect to recognizers, and discuss parsers later.

We assume that the input is a sequence of tokens `input`, of length $\texttt{input}_{\#}$, the members of which are accessed through an index `j`. Irrespective of the programming language used, the recursive-descent approach can be thought of as requiring a recognizer to be built for each terminal of the grammar, and the subsequent combination of these and other recognizers to build recognizers for the non-terminals of the grammar. For ambiguous grammars, the recognizers can be thought of as 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 finished successfully 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`.

As a running example, we consider a recognizer corresponding to the grammar *sS ::= 's' sS sS | empty* and `input = "ssss"`. We have chosen to use this example grammar throughout the paper as it is highly ambiguous. According to Aho and Ullman `sS` generates

$$\binom{2n}{n}/(n+1)$$

different leftmost parses of strings consisting of `n` s's. For example, for `n = 16` there are over `35` million parses. Although natural language is not as ambiguous as this, large numbers of parses can be generated during lexical analysis. We give a natural-language example later.

2.1 Recognizers for single tokens

A recognizer `term_t` for a single terminal *t* of the grammar takes an index `j` as input. If `j` is greater than the length of the 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'` is defined as follows

```
term_s  j  = {}      , if j > length of input
           = {j + 1}, if input!j = 's'
           = {}      , otherwise
```

2.2 Empty

The `empty` recognizer always succeeds and returns its input index in a singleton set as result.

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

2.3 Alternate recognizers

A recognizer corresponding to a construct *p | q* in the grammar is built by combining recognizers for *p* and *q*. When the composite recognizer is applied to an index `j`, it first applies `p` to `j`, then applies `q` to `j`, and then unites the results. We introduce the operator `orelse` to denote the process of combining alternate recognizers. This operator can be implemented in various ways depending on the programming language used.

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

For example, assuming that the input is `"ssss"`

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

2.4 Sequence recognizers

A recognizer corresponding to a construct *p q* in the grammar is built by combining recognizers for *p* and *q*. When the composite recognizer is applied to an index `j`, it first applies `p` to `j`, then applies `q` to each index in the set of the results returned by `p`. It returns the union of each of these applications of `q`. We introduce the operator `then` to denote the process of sequencing recognizers.

```
(p then q) j = ⋃(map q (p j))
```

For example, assuming that the input is `"ssss"`

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

2.5 An example recognizer

The operators `orelse` and `then` can be used to define recognizers through recursion and mutual recursion. For example, the following recognizer `sS` corresponds to the example grammar *sS ::= 's' sS sS | empty*:

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

Assuming that the input is `"ssss"`, the recognizer `sS` returns a set of 5 results, the first 4 results corresponds to proper prefixes of the input being recognized as an `sS`. The result `5` corresponds to the case were the whole input is recognized as an `sS`.

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

2.6 Limitations of the approach described so far

The reader may have noticed that the number of entries in the output list of the recognizer is less than the number of possible parses. This is owing to the fact that the results generated during the process are united. Although this reduces the amount of work done in recognition, the process still has exponential time complexity with respect to the length of the input. This is because recognizers may be repeatedly applied to the same index during the backtracking process which is induced by the operator `orelse`. In section 3, we show how Norvig's method can be used to achieve polynomial complexity "memoizing" the recognition functions so that they reuse previously-computed results.

The second limitation is that the approach cannot be used to build recognizers that correspond directly to left recursive grammars. That is grammars in which a nonterminal `p` derives the expression `p ...`. Application of the corresponding recognizers would result in infinite descent. We show how to avoid this problem, without having to transform the grammars, in section 5.

We conclude thsi section by noting that we have tried to make the above description of top-down parsing independent of programming-language or paradigm. However, our formalism is influenced by the "parser combinator" style which has been developed by the functional-programming community. See for example Hutton [5], Koopman and Plasmeijer [7], and Wadler [16].

3 MEMOIZATION

Norvig [22] has shown how the worst-case complexity of top-down recognition can be improved from exponential to cubic through a process of memoization. The basic idea

is that a memotable is constructed during the recognition process. At the beginning of the process 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, the first component of each pair is an index j at which the recognizer r_i has been applied, the second component is the 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 index and the memotable is updated with those results before they are returned by the recognizer. For non-left-recursive recognizers, this process ensures that no recognizer is ever applied to the same index more than once.

One method of implementing memoization, that was suggested by Norvig, is to have a global memotable, and to encapsulate the recognizers which are to be memoized in a function which performs the memotable lookup and update. In general the process can be implemented in various ways depending on the programming language used. We introduce the operator `memoize` to indicate that a recognizer has been memoized. This operator takes a string which denotes the name of the recognizer, together with the recognizer itself as arguments. The name is used for memotable lookup and update. For example, consider the following memoized recognizer:

```
msS = memoize "msS"
        ((ms then msS then msS)
          orelse empty)

ms = memoize "ms" term_s
```

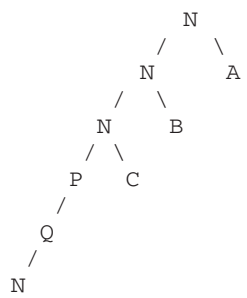
The operator memoize is defined as follows:

```
memoize name ri j
   if lookup succeeds,
      return memotable result
   else
      apply ri to j
      update' table with results
      return results
```

The recognizer `msS` is the same as `sS` in all respects except that it has cubic complexity (see later).

4 ACCOMMODATING LEFT RECURSION

In order to accommodate left recursive productions, we simply use another table `ctable` during the memoization process. The new table contains a set of values value c_{ij} denoting the number of times each recognizer r_i has been applied to an index j. For non-left-recursive recognizers this 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 count is increased on recursive descent (owing to the fact that the memotable is only updated on the recursive ascent after the recognizer has been applied). Application of a recognizer N to an input j is failed whenever the application count already exceeds the length of the remaining input plus 1. When this happens no parse is possible (other than spurious parses which could occur with circular grammars). As illustration, consider the following branch being created during the parse of two remaining tokens on the input:



The last call of 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, orelse 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.

Notice that simply failing a parse when a branch is longer than the length of the 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.

To make use of the new table, we simply modify the `memoize` operator to check and increment the c_{ij} counters at appropriate points in the computation: if the memotable lookup for the recognizer r_i and the index j produces a non-empty result, that result is returned with the two tables unchanged. However, if the memotable does not contain a result for that recognizer and that input, 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 tables 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.

```
memoize name ri j
  if lookup succeeds,
      return memotable results
  else
    if cij > (input length)-j+1
       return {},
    else
       increment cij counter in ctable
       apply ri to j
       update' memotable with results
       return results
```

The memotable update function `update'` is slightly different from the `update` function given in section 3. This is owing to the fact that, on the recursive descent a recognizer is only applied to an input if there is no entry in the memotable. In the original memoization function, the results returned by the recognizer on ascent are simply added to the memotable. However, if left-recursion occurs, the memotable may have been updated by the same recognizer for the same input lower in the parse tree. Therefore on ascent it is necessary to unite the results of each application of the recognizer with memotable results which may have been added by calls lower in the parse tree.

Using this approach, recognizers may now be defined using explicit (as in `mSL`) or hidden (as in `mZ`) left recursion. For example:

```
mSL = memoize "mSL"
       ((mSL then mSL then2 ms)
        orelse2 empty2)

ms = memoize "ms" term_s

mZ = memoize "mZ" (mz orelse mY)

mY = memoize "mY" (mZ then mSL)
mz = memoize "mz" term_z
```

The following are example applications of `mZ`, assuming that the tables were initially empty. The second component is the table showing for each recognizer the number of times it visited each position in the input. The last component is the table showing for each recognizer the positions at which it was applied and the results of that application.

```
Grammar mZ ::= 'z'| mY
        mY ::= mZ mSL

input = "zss"

mZ 1 ([],[]) => {2,3,4}
ctable =
  {("mZ", {(1,3)}),
   ("mz", {(1,1)}),
   ("mY", {(1,4)}),
   ("mSL",{(2,2),(3,1),(4,0)}),
   ("ms", {(2,1),(3,1),(4,1)})}
memotable =
  {("mz", {(1,{2})}),
   ("mY", {(1,{2,3,4})}),
   ("mZ", {(1,{2,3,4})}),
   ("mSL",{(2,{2,3,4}),(3,{3,4}),(4,{4})}),
   ("ms" ,{(2,{3}),(3,{4}),(4,{})})})
```

5 INFORMAL DISCUSSION OF TERMINATION

Basic recognizers such as `term_s` and the recognizer `empty` clearly terminate for finite input. Other recognizers that are defined through mutual and nested recursion are applied by the `memoize` function which takes a recognizer and an index `j` as input and which accesses two tables `ctable` and `memotable`. If a recognizer has an entry in `memotable` for the index `j`, it is not applied and therefore we do not need to consider the size of the arguments. If it does not have an entry in `memotable`, we must consider two cases of possible recursion: 1) it is not a left-recursive call and therefore at least one other recognizer must have been applied before it which consumed at least one token and increased the index by at least one before the call, 2) it is a left-recursive call and the index argument has not been changed. In this case, `memoize` increments the left-recursion counter in `ctable` for that recognizer and that index before the recursive call is made. Therefore an appropriate measure function maps the index and `ctable` values to a number which increases by at least one for each recursive call. The fact that the number is bounded by conditions imposed on the size of the index and on the sizes of the left-recursion counters establishes termination.

6 COMPLEXITY

In the following complexity analysis, we assume that the sets of results are represented as ordered lists, as are the entries in the tables. We now show 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 $n = input_{\#}$.

Assumption 2 — Elementary operations: We assume that the following operations require a constant amount of time:

1. Testing if two values are equal, less than, etc.
2. Extracting the value of a tuple.
3. Adding an element to the front of a list.
4. Obtaining the value of the `ith` element of a list whose length depends on $R_{\#}$ but not on $input_{\#}$.

Assumption 3 — Merging of lists depends on their length.

Lemma 4 — Memotable lookup and update, checking and incrementing left-recursion counters: From lemma 1 and the definition of `memoize`, `memotable` has size $O(n^2)$ and `ctable` has size $O(n)$ and . The function `lookup` is $O(n)$ requiring a search of `memotable` for the recognizer name and then a search of the $O(n)$ list of results (one for each index). The function `update` is $O(n)$ requiring the same $O(n)$ search as `lookup` plus a possible $O(n)$ merge of results. Checking for the value of a left-recursion counter in `ctable` and increment of such a counter is clearly $O(n)$.

Lemma 5 — Basic recognizers Application of a basic recognizer is at most $O(n)$ requiring the use of an index `j` into the input. Application of `empty` is also $O(n)$, simply enclosing a single index in a list.

Lemma 6 — Alternation: Assuming that the recognizers r_p and r_q have been applied to an index `j` and that the results have already been computed, application of a memoized recognizer r_p `orelse` r_q to `j` involves the following steps:

1. one memotable `lookup` — $O(n)$
2. and, if the recognizer has not been applied before:
   a. one left-recursion counter check — $O(n)$
   b. and, if the counter check permits:
      - merging of two result lists — $O(n)$
      - one memotable `update` — $O(n)$

Lemma 7 — Sequencing: Assume that the recognizer r_p has been applied to an index `j` and that the results `res` have been computed. In the worst case, `res = [j, j+1,j+2, .. n+1]`. Assume also that $\forall$`j'`$\in$`res` r_q `j'` has been computed. Then, application of a memoized recognizer (`then`) to an index `j` involves:

1. one memotable `lookup` — $O(n)$
2. and, if the recognizer has not been applied before:
   a. one left-recursion counter check — $O(n)$
   b. and, if the counter check permits:
      - application of r_q to each index in `res` and merging of the result lists— $O(n^2)$.
      - one memotable `update` — $O(n)$

Proof of $O(n^3)$ complexity for non-left-recursive recognizers.
In the worst case, each recognizer $r_i \in R$ is applied to each of the `n` indices at most once. The cost of an application to one index is:

Case 1: For basic recognizers the cost is $O(n)$ — Lemma 5.
Case 2: For recognizers of the form (r_p `orelse` r_q) the cost is $O(n)$ — Lemma 6.
Case 3: For recognizers of the form (r_p `then` r_q) the cost is $O(n^2)$ — Lemma 7.
In practice recognizers can be a combination of more than two recognizers. However, from definition 2 the number of component recognizers is finite and is independent of `n`.

It follows that the total cost is $O(n^3)$.

Proof of $O(n^4)$ complexity for left-recursive recognizers.
In the worst case, each recognizer $r_i \in R$ is applied to each of the `n` indices at most `n` times before being curtailed. It follows that the total cost is $O(n^4)$.

7 IMPLEMENTATION AND EXPERIMENTAL RESULTS

The approach described in this paper has been implemented using parser combinators in the pure functional programming language Haskell using an method called "monadic memoization" [3]. Details, including proofs of termination and complexity, are available in Frost and Hafiz [2]. An example recognizer was constructed corresponding to the grammar *`sS ::= 's' sS sS | empty`* and applied to sequences of `'s'`s of varying length. The results in the table at the end of this paper were obtained using the Haskell interpreter Hugs 98 on a PC with 0.5 GB of RAM. The results appear to support our claim that we have avoided exponential behaviour.

It should be noted that the example grammar that we have used so far is highly ambiguous, far exceeding any ambiguity found in natural language. Also, many constructs in natural language are defined without use of left recursion. Consequently, we have also investigated the performance of our approach with respect to a small natural-language grammar. The following is the definition of the recognizer in Haskell. This example illustrates the close correspondence between the grammar and the program code when parser combinators are used (obviating the need to give the grammar separately in this example). The recognizer `sent` recognizes sentences in a very small subset of English. `tp` stands for termphrase, `det` for determiner, and `vp` for verbphrase:

```
sent = memoize "sent"
        (tp `then2` vp `then2` tp)

tp = memoize "tp"
     (simple_tp
     `orelse2` (tp `then2` join `then2` tp))

join = memoize "join"
      (term2 "and" `orelse2` term2 "or")
```

```
simple_tp = memoize "simple_tp"
              (proper_noun
               `orelse2` det_phrase)
proper_noun = memoize "proper_noun"
               (term2 "helen
                `orelse2` term2 "john"
                `orelse2` term2 "pat")
det_phrase = memoize "det_phrase"
               (det `then2` noun)

det =  memoize  "det"
               (term2 "every"
                `orelse2` term2 "some")

noun = memoize "noun"
               (term2 "boy"
                `orelse2` term2 "girl"
                `orelse2` term2 "man"
                `orelse2` term2 "woman")
vp = memoize "vp"
     (verb
     `orelse2` (vp `then2` join `then2` vp))

verb = memoize "verb"
               (term2 "knows"
                `orelse2` term2 "respects"
                `orelse2` term2 "loves")
```

Application of `tp` to the ambiguous termphrase:

```
      ["every","boy","or","some","girl",
       "and","helen","and","john","or","pat]
```

requires 57,131 reductions, 102,477 cells, and returns the following result:

```
([3,6,8,10,12],


([("tp",[(1,11),(4,8),(7,5),(9,3),(11,1)]),
  ("simple_tp"  ,[(1,1),(4,1),(7,1),etc.]),
  ("proper_noun",[(1,1),(4,1),(7,1),etc.]),
   etc.

 [("proper_noun",[(1,[]),(4,[]),(7,[8]),
                  (9,[10]),(11,[12])]),

  ("det",        [(1,[2]),(4,[5]),(7,[]),
                           (9,[]), (11,[])]),

  ("noun",       [(2,[3]),(5,[6])]),
  ("det_phrase", [(1,[3]),(4,[6]),(7,[]),
                           (9,[]),(11,[])]),

  ("simple_tp"  ,[(1,[3]),(4,[6]),(7,[8]),
                       (9,[10]),(11,[12])]),

  ("tp",         [(1,[3,6,8,10,12]),
                  (4,[6,8,10,12]),
                  (7,[8,10,12]),
                  (9,[10,12]),(11,[12])]),

  ("join",       [(3,[4]),(6,[7]),(8,[9]),
                  (10,[11]),(12,[])])]))
```

Application of `sent` to the list of tokens corresponding to the highly-ambiguous sentence "every boy or some girl and helen and john or pat knows and respects or loves every boy or some girl and pat or john and helen" took 408,454 reductions, used 691,504 cells, and returned results in less approximately 0.5 seconds.

The prototype processor is clearly not fast. However, the combinators were not optimized, and Hugs 98 is an interpreted version of Haskell. Our approach to top-down parsing could be implemented in a more efficient programming environment.

8 COMPACT REPRESENTATION OF PARSE TREES

Reference to the example application given above shows that most of the information for reconstructing the parse trees is already available in the memotable. For example, the memotable output shows that the input contains three proper nouns at positions 7, 9 and 11, etc. Additional information could be collected during the recognition process by naming and memoizing the alternative recognizers on the right-hand sides of grammar productions. This, together with the grammar, would provide all of the information necessary to extract the potentially exponential number of parses from the memotable. The memotable is bounded by the number of recognizers, the number of indices, and the sizes of the result sets. The latter two of which depend on the length of the input. Consequently, the memotable has worst-case size $O(n^2)$ and provides a compact representation of the possibly-exponential number of parse trees which appears to be similar to that proposed by Tomita [24].

The major advantage of creating compact representation of parse trees is that syntactic agreement rules, together with semantic rules, can be used to prune out sub-trees which are shared by many possible parses.

9 CONCLUDING COMMENTS

Future work includes:

1. Extending the approach to parsers and evaluators.
2. Optimizing the combinators used in the Haskell implementation, using the techniques of Koopman and Plasmeijer [14]
3. Testing the approach on large natural-language grammars.

10 ACKNOWLEDGEMENTS

Richard Frost acknowledges the support of NSERC the Natural Sciences and Engineering Research Council of Canada.

length of input `n`	number of leftmost parses with `S ::= 's' S S | empty` $\binom{2n}{n}/(n+1)$ Note that the number of partial parses consistent with s^n is larger than this	number of reductions `mS ::= 's' mS mS | empty` `mSL ::= mSL mSL 's' | empty`		
		mS without memoization (checks all partial parses)	mS with memoization	mSL
3	5	2,781	2,834	5,990
6	132	65,049	7,081	28,366
12	20,812	out of space	23,297	206,903
24	128,990,414,734		99,469	2,005,561
48	1.313278982422e+26		424,929	17,125,991
96	huge		2,620,807	out of space
192			18,119,356	
384			134,091,390	

11 REFERENCES

1. Camarao, C., Figueiredo, L. and Oliveira, R.,H. (2003) Mimico: A Monadic Combinator Compiler Generator. *Journal of the Brazilian Computer Society* Vol 9(1).
2. Frost, R. A. and Hafiz, R. (2006) Using monads to accommodate ambiguity and left recursion with parser combinators. *Technical Report 06–007* School of Computer Science, University of Windsor, Canada.
3. Frost, R. A. (2003) Monadic memoization — Towards Correctness-Preserving Reduction of Search. *AI 2003* eds. Y. Xiang and B. Chaib-draa. LNAI 2671 66–80.
4. Frost, R. A. (1993) Guarded attribute grammars. *Software Practice and Experience.*23 (10) 1139–1156.
5. Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming* 2 (3) 323–343.
6. Johnson, M. (1995) Squibs and Discussions: Memoization in top-down parsing. *Computational Linguistics* 21 (3) 405–417.
7. Koopman, P. and Plasmeijer, R. (1999) Efficient combinator parsers. In Implementation of Functional Languages, LNCS, 1595:122 138. Springer-Verlag.
8. Koskimies, K. (1990) Lazy recursive descent parsing for modular language implementation. *Software Practice and Experience*, 20 (8) 749–772.
9. Kuno, S. (1965) The predictive analyzer and a path elimination technique. *Communications of the ACM* 8(7) 453 — 462.
10. Leermakers, R. (1993) *The Functional Treatment of Parsing*. Kluwer Academic Publishers, ISBN 0–7923–9376–7.
11. Lickman, P. (1995) Parsing With Fixed Points. *Master's Thesis*, University of Cambridge.
12. Nederhof, M. J. and Koster, C. H. A. (1993) 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.
13. Norvig, P. (1991) Techniques for automatic memoisation with applications to context-free parsing. *Computational Linguistics* 17 (1) 91 - 98.
14. 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.
15. Tomita, M. (1985) *Efficient Parsing for Natural Language.* Kluwer, Boston, MA.
16. Wadler, P. (1985) How to replace failure by a list of successes, in P. Jouannaud (ed.) *Functional Programming Languages and Computer Architectures* Lecture Notes in Computer Science 201, Springer-Verlag, Heidelberg, 113.