Modular Parsers for Natural-Language Processing (with proofs in the appendices)

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

[1]School of Computer Science, University of Windsor

[2]Department of Computer Science, University of Durham.

Abstract. Highly-modular parsers can be constructed as directly-executable top-down backtracking processors, and combined to form more-complex parsers using operators defined in the host programming language. In functional and logic programming, this approach is implemented using parser combinators and definite clause grammars respectively. Standard implementations are inefficient for ambiguous languages and cannot accommodate left-recursive grammars, and hence are limitated in their use. Memoization is known to improve efficiency, and there has been some success in accommodating left-recursion. However, there is no integrated solution to both problems that is entirely satisfactory.

This paper extends the approach to accommodate ambiguity and direct and indirect left recursion in polynomial time, and to create compact polynomial-sized representations of the potentially exponential number of parse trees which can be generated for highly-ambiguous languages.

1 Introduction

Combinator parsers and definite clause grammars are highly modular and have structures similar to the grammars defining the languages to be processed. As such, they are executable specifications with all of the associated advantages, e.g. piecewise construction and in-line coding of processors for application-dependent languages. Additional advantages are that 1) parsers can be defined to return semantic values directly instead of intermediate parse results, 2) parsers can be parameterized in order to accommodate context-sensitive languages (e.g. van Eijck [4]), 3) the syntactic and semantic processing of component parsers can be implemented to reflect differences in how subsets of the input language are to be treated, and 4) in functional programming, the type checker can be used to catch errors in parsers attributed with semantic actions.

The advantages above have contributed to the popularity of parser combinators and definite clause grammars and they have both been used extensively, especially by researchers in the A.I. community. However, standard implementations have exponential time and space complexity for ambiguous languages, and cannot handle left-recursion. As such they have limited use. For example, their use in natural-language processing is limited due to the fact that ambiguity is inherent in natural language and left-recursion is necessary in order to obtain both leftmost and rightmost parses of ambiguous input. Converting grammars to non-left-recursive form is not appropriate as it can result in loss of some parses.

Norvig's [14] method of using memoization to achieve polynomial complexity for top-down parsing has been used to improve efficiency of combinatory parsers

and definite clause grammars, and various approaches have been developed for accommodating left recursion (see section 4). This paper, however, describes the first method which accommodates ambiguity and direct and indirect left recursion in polynomial time, and which creates compact polynomial-sized representations of the potentially exponential number of parse trees.

Our method handles arbitrary context-free grammars, as can Earley's [3] parser and faster versions of it (Aycock and Horspool [1]). Our method produces compact representations of parse trees as does Tomita's [16] parser. A disadvantage of our approach is that it has $O(n^4)$ time complexity, for ambiguous grammars, compared with $O(n^3)$ for Earley-style parsers. We believe that in some applications (e.g. in experimenting with new linguistic theories) this increase in complexity is compensated for by the advantages discussed above, and also that in many applications the actual time resulting from the increase in complexity in parsing is not a major factor in the overall time required when subsequent semantic processing is taken into account.

We present our approach in the context of combinator parsing. However, the central ideas are also applicable to definite clause grammars. We begin with a motivating example, and then provide some background material. We follow that with an incremental description of the method. We then provide an outline of proofs of termination and complexity, followed by some experimental results. Complete proofs and the Haskell code of an implementation of our method are available in the appendices.

2 An Example Use of the Parser

The grammar below is from Tomita [16], `S` stands for sentence, `NP`-nounphrase, `VP`-verbphrase, `PP`-prepositional phrase, `prep`-preposition, and `det`-determiner:

```
S    ::= NP VP | S PP                  det  ::= 'a' | 't'
NP   ::= noun | det noun | NP PP       noun ::= 'i' | 'm' | 'p' | 'b'
PP   ::= prep NP                       verb ::= 's'
VP   ::= verb NP                       prep ::= 'n' | 'w'
```

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

```
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')
```

Note the close structural relationship between the parser code and the grammar. Each component parser `np`, `pp`, etc. can be tested separately. 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". We discuss this compact representation later.

```
apply p "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))]]
```

3 Top-Down Backtracking Recognition

Top-down recognizers are often 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 ro 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 `#input`, the members of which are accessed through an index `j`. 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 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.

A recognizer `term_t` for a single terminal is a function which 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. 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 (note that we use a functional pseudo code throughout, in order to make the paper accessible to a wide audience):

```
term_s  j  = {}      , if j > #input
           = {j + 1}, if the jth element of the input = 's'
           = {}      , otherwise
```

The `empty` recognizer 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`, applies `q` to `j`, and then unites the results. (We use single quotes to surround infix functions):

```
(p `orelse` q) j = (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` in the grammar is built by combining those recognizers using the parser combinator `` `then` ``. 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`.

```
(p `then` q) j = ⋃(map q (p j))
```

e.g., assuming that the input is `"ssss"`, then `(term_s `then` term_s) 1 => {3}`

The combinators above can be used to define composite mutually-recursive recognizers, e.g. consider the grammar `sS ::= 's' sS sS | empty`. The corresponding recognizer `sS` can be defined as follows:

```
sS = (term_s `then` sS then 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 were the whole input is recognized as an `sS`.

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

The method described above 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 induced by the operator `` `orelse` ``. We show later how complexity can be improved, using Norvig's memoization technique.

4 Previous Approaches to Left Recursion

Another limitation is the inability to accommodate left-recursive grammars, i.e. grammars in which a non-terminal `p` derives an expansion `p ..` headed with a `p` either directly or indirectly. Application of a recognizer for such a grammar results in infinite descent. Various solutions have been proposed [2,5,8,10,11,12,13,15], but non has been entirely satisfactory. However, our method has borrowed something from many of them. As with the exponential approach of Kuno [10], we use the length of the remaining input to curtail recursive descent. As with the exponential approaches

of Shiel [15] and Nederhof and Koster [13], we pass an additional parameter to parsers that is used to curtail recursion. However, our parameter also contains a memotable to improve complexity. As with Johnson [8] our approach integrates a technique for dealing with left recursion with memoization to achieve polynomial complexity. Our $O(n^4)$ method differs from Johnson's $O(n^3)$ approach in the technique that we use to accommodate left recursion – Johnson uses continuations. However, our algorithm returns compact representations of parse trees whereas Johnson's method does not. Johnson states, in his conclusion, that "an implementation attempt (to create a compact representation) would probably be very complicated." As such, we view our approach as a variation of Johnson's approach, which is better suited for implementation using parser combinators or definite clause grammars and which leads to a straightforward compact representation of parse trees.

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 [12] has also used a monad in his approach, but for a different purpose. Lickman accommodated left-recursion, but in exponential time. Our approach is most closely related to the recognition algorithm described in Frost and Hafiz [5]. We also use memoization and bounds to curtail left recursion. However, our algorithm uses contexts (see later) to accommodate indirect left recursion, and is a parser, rather than a recognizer, which returns compact representations of parse trees.

5 The New Algorithm

5.1 Memoization

To improve complexity, a memotable is constructed during recognition. 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 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.

One approach, suggested by Norvig [14], is to encapsulate the recognizers in a `memoize` function which performs the memotable lookup and update. This "memoization" process can be implemented in various ways depending on the programming language used. We introduce the function `memoize` to indicate that a recognizer has been memoized. This function 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. `memoize` is defined as follows, where the `update` function stores the result of recognizer application in the table:

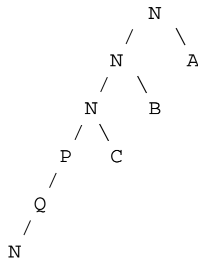
```
memoize name ri j = if lookup succeeds, return memotable result
                    else apply ri to j, update table, and return results
```

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

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

5.2 Accommodating direct left recursion

In order to accommodate left recursive productions, 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 the 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. 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 (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):



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:

```
memoize name ri j  =
 if lookup succeeds, return memotable results
 else if cij > (#input)-j+1, return {},
 else increment cij, apply ri to j, update memotable and return results
```

Recognizers may now be defined using explicit left recursion. For example:

```
mSL = memoize "mSL"  ((mSL `then` mSL `then` ms)  `orelse` empty)
 ms = memoize "ms" term_s
```

5.3 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 parse 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 were 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:

```
[(index, [(recognizer id, left-rec count)])]
```

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.

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 limits, and if so, which recognizers, at which indices, caused the curtailment. A list of `(recognizer id, index)` pairs which caused curtailment in any of the subtrees is returned with the result. The functions `` `orelse` `` and `` `then` `` 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. Whenever 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) Considering 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.

5.4 Extending recognizers to parsers

The idea is that instead of returning a list of indices representing successful end points for recognition, parsers also return, and store, the parse trees. However, in order that these trees be represented in a compact form, they are constructed with reference to other entries in the memotable, enabling the explicit sharing of common subtrees. The example in section 2 illustrates the results returned by such a parser.

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

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

The combinator `` `then` `` 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))]])etc.
```

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 same 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 verbphrase `"vp"` spanning positions 2 to 11 refers to a group of subtrees corresponding to the two parses of a nounphrase `"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))]])
```

We show later that this approach allows the potentially-exponential number of parse trees to be represented in $O(n^3)$ space as in Tomita's [14] algorithm.

6 Termination

The only source of iteration is in recursive function calls. Therefore, the proof, a complete version of which is available at the URL given in section 1, is based on the identification of a measure function which maps the arguments of recursive calls to a well-founded ascending sequence of values.

Basic recognizers such as `term_s` 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 a number, which increases by at least one for each recursive call. The fact that the number 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.

7 Complexity of Recognition and Parsing

Complete formal proofs are available at the URL given in section 1. The following is an informal overview of those proofs. 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: we begin by showing that `orelse` requires $O(n)$ operations to merge the results from two alternate recognizers, and that `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 of recognition. 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 terminal or non-terminals), the complexity of non-left recursive parsers is $O(n^3)$ and of left-recursive parsers it is $O(n^4)$. The extended 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 — see section 2 for examples.) 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 then. The analysis shows that 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 `then`. Owing to the fact that the complexity of `then` 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 compact representation of parse trees stored 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 $O(n^3)$.

8 Implementation

We have implemented our method in the pure functional programming language Haskell using a technique called "monadic memoization" [6]. The code is available at the URL given in section 1. Use of a monad allows the memotable to be systematically threaded through the parsers while hiding the details of table update and reuse. The use of a lazy language allows us to easily generate the first n-parses without having to perform any unnecessary computation (although this can be done in a non-lazy language with careful programming). Our algorithm can be readily implemented in other programming languages which support recursion and dynamic data structures.

9 Experimental Results and Concluding Comments

The complexity analysis does not tell us how well the algorithm performs in practice. Consequently, we tested a parser for the following grammar `sS ::= 'a' sS sS | empty`, and some weakly-equivalent left-recursive versions, on input of various lengths. We chose this grammar as it is highly ambiguous. According to Aho and Ullman sS generates a huge number of leftmost parses of sequences of 'a''s. For example, for 24 'a''s there are over 128 billion parses which cover all of the input, and many more partial parses. We used four parsers in our experiment (the results are given on the following page):

1) An unmemoized non-left–recursive parser:
```
s = (term 'a' `then` s `then` s) `orelse` empty
```
2) A memoized non-left–recursive parser:
```
sm = memoize "sm" ((term 'a' `then` sm `then` sm) `orelse` empty)
```
3) A memoized left–recursive parser:
```
sml = memoize "sml" ((sml `then` sml `then` term 'a') `orelse` empty)
```
4) A left–recursive parser with all parts memoized. This improves efficiency similarly to converting the grammar to Chomsky Normal Form. Note that this version overrides the associativity of `then`, and memoizes a subcomponent of the parser.

```
smml = memoize "smml" ((smml `then`
     (memoize "smml_a" (smml `then` term 'a'))) `orelse` empty)
```

In conclusion, we have extended previous work of others on modular parsers, constructed as executable specifications, to accommodate ambiguous left-recursive grammars in polynomial time and space. While not as efficient as Earley-style parsers, our approach offers several advantages which are of particular relevance to applications such as the 1) the investigation of new linguistic theories, and 2) the prototyping of natural-language interfaces to databases, search engines, and web pages, where complex and varied semantic actions need to be closely integrated with syntactic processing. Future work includes analysis w.r.t. grammar size, testing with

large grammars, and extending the approach to a fully-general attribute grammar programming environment.

input length	Number of parses, excluding partial parses	Seconds to generate and display the packed representation of all parses, including partial parses, using the Glasgow Haskell Compiler on a PC with 0.5 GB RAM			
		s	sm	sml	smml
3	5	0.05	0.02	0.07	0.08
6	132	1.22	0.15	0.20	0.18
12	20,812	out of space	0.52	0.80	0.71
24	128,990,414,734		4.24	5.84	4.28
48	1.313278982422e+26		32.65	out of space	68.21

References

1. Aycock, A. and Horspool, R. N. (2002) Practical Earley Parsing. The Computer Journal, 45(6):620-630.
2. Camarao, C., Figueiredo, L. and Oliveira, R.,H. (2003) Mimico: A Monadic Combinator Compiler Generator. *Journal of the Brazilian Computer Society* Vol 9(1).
3. Earley, J. (1970) An efficient context-free parsing algorithm. Communications of the Association for Computing Machinery, 13(2):94-102.
4. Eijck, J. van (2003) Parser combinators for extraction. In Paul Dekker and Robert van Rooy, editors, Proceedings of the Fourteenth Amsterdam Colloqium, pages 99104. ILLC, University of Amsterdam.
5. 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.
6. 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.
7. Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming* 2 (3) 323–343.
8. Johnson, M. (1995) Squibs and Discussions: Memoization in top-down parsing. *Computational Linguistics* 21 (3) 405–417.
9. Koskimies, K. (1990) Lazy recursive descent parsing for modular language implementation. *Software Practice and Experience*, 20 (8) 749–772.
10. Kuno, S. (1965) The predictive analyzer and a path elimination technique. *Communications of the ACM* 8(7) 453 — 462.
11. Leermakers, R. (1993) *The Functional Treatment of Parsing*. Kluwer Academic Publishers, ISBN 0–7923–9376–7.
12. Lickman, P. (1995) Parsing With Fixed Points. *Master's Thesis*, University of Cambridge.
13. 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.
14. Norvig, P. (1991) Techniques for automatic memoisation with applications to context-free parsing. *Computational Linguistics* 17 (1) 91 - 98.
15. 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.
16. Tomita, M. (1985) *Efficient Parsing for Natural Language.* Kluwer, Boston, MA.

Appendix 1 — Proof of Termination

Basic recognizers such as `term_s` 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`. If a recognizer has an entry in `memotable` for the index `j`, it is not reapplied and the call terminates with those results. 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-rec counter for that recognizer and that index before the recursive call is made. Therefore an appropriate measure function maps the index and the set of left–rec values to a number, which, according to the above argument, increases by at least one for each recursive call. The fact that the number 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 above analysis also applies to parsers.

Definition 1:
`input` is a finite sequence of tokens of length $\texttt{input}_{\#}$

Definition 2:
`R` is a finite set of recursively-defined memoized recognizers of size $R_{\#}$ which have been constructed by finite application of `empty`, `term`, `orelse`, and `then`. The members of `R` are denoted by $r_i, \ 1 \leq i \leq R_{\#}$.

Definition 3:
`C` is a finite set of "left-rec counters", the members of which are denoted by c_{ij} where $1 \leq i \leq R_{\#}$ and $1 \leq j \leq \texttt{input}_{\#}$. The counter c_{ij} is the left-rec counter for recognizer r_i applied to the `input` at position given by index `j`. These counters are stored in `ctable.`

Definition 4:
The measure function $||.||$ maps recognizer inputs `(ind,(ctable,mtable))` to the number = $\texttt{ind} \ + \Sigma c_{ij} \in \texttt{ctable}$

Assumption 1: The starting recognizer is applied to index `1` and the empty memotable `([],[])`.

Lemma 1:
$\forall_{\text{indexes}} j \ \ 1 \leq j \leq (\texttt{input}_{\#} + 1)$ this follows from the definition of `term` which is the only function which increments the indexes, and then only if the index $\leq \texttt{input}_{\#}$. It follows from the definition of merge that the length of the lists of results returned by recognizers $\leq$ ($\text{input}_{\#}$ + 1).

Lemma 2:
$\forall_{counters} c_{ij}$ 0 $\leq$ c_{ij} $\leq$ (input$_{\#}$-j+1) this follows from the definition of memoize which is the only function which increments the left-rec counters and then only if the counter $\leq$ input$_{\#}$-j.
Lemma 3:
$\|.\|$ has a minimum value of 1 and is bound to finite size (from definitions 1 to 4 and lemmata 1 and 2). Furthermore, $\|.\|$ can only increase increments of 1 (definitions of term and memoize) hence there it has well-founded ordering $\succ$.

Induction lemma IL R:
$\forall r_i \in$R if r_i(ind,(ctable,mtable)) returns results
(results,(ctable',mtable')
then
results = [] $\vee$
($\forall_{j \in results}$ j $\geq$ ind) & ($\Sigma c_{ij} \in$ctable'$\geq \Sigma c_{ij} \in$ctable)

That is, each index in the list of results returned by a recognizer is equal or greater than the input index, and the sum of the left-rec counters in the ctable returned by the recognizer is equal or greater than the sum of the counters in the table given as input.

Proof by induction on R$_{\#}$
Base case 1: IL {empty} (definition of empty).

Base case 2: IL {memoize "any" (term any)} (definition of term and memoize).

Induction Step:
Assume IL S, *Show* IL(S $\cup$ {r}):

Case 1: r = memoize "r" (p `then` q) for somep,q$\in$S. IL{r} follows from the definition of then and memoize.

Case2: r = memoize "r" (p `orelse` q) for some p,q$\in$S. IL{r} follows from the definition of orelse.

In practice, recognizers can be a combination of more than two recognizers constructed with `orelse` and `then`. However, from definition 2 the number of component recognizers is constant.

IL(S $\cup$ {r}) follows from IL S and IL{r} . Hence IL R. It also follows from the definitions of memoize, orelse, and then, that IL holds for the closure of R under the operation of then. i.e IL R ^{then}

Proof of termination:
$\forall$ $r_i \in$R and $\forall r_k \in R^{then}$ and any recursive expansion of r_i such that
r_i derives r_k `then` r_i'.

Suppose r_k returns (results,(ctable',mtable')) . From the induction lemma, there are two cases:

Case 1: r_k fails, results = [] , there is no recursive call of r_i' (defn. of then).

Case2: results $\neq$ []. It follows from the definition of then that: r_i(ind,(ctable,mtable)) invokes a recursive call r_i'(ind',(ctable',mtable') for every ind'$\in$results. For each of these calls, there are two subcases (induction lemma):

Sub-case a) ind'>ind & $\Sigma c_{ij} \in$ctable'$\geq \Sigma c_{ij} \in$ctable)

Sub_case b) ind'= ind & $\Sigma c_{ij} \in$ctable'$\geq \Sigma c_{ij} \in$ctable)

sub-case b) is the left-recursive case, in whichmemoize increments the left-recursion counter forr_i inctable before the recursive applicationr_i'.

In both sub-cases, it follows from the definitions of ||.|| and memoize, that, for all recursive calls r_i'

||arguments to r_i'|| $\succ$ ||arguments to r_i|| •

Extending recognizers to parsers does not involve any additional recursive computations and consequently, the above proof of termination also applies to parsers.

Appendix 2 — Complexity of recognizers

This appendix contains a proof of complexity of recognizers based on the new algorithm.

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-rec counters: From lemma 1 (in the proof of termination) 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 $O(n)$. These two operators could be made O(1) by use of arrays. In either case this does not affect overall complexity.

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 $O(1)$, 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 rq j'` has been computed. Then, application of a memoized recognizer (r_p `then` r_q) 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.
The cost of an application of a recognizer to an 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.
Although recognizers can be built from a combination of more than two recognizers, using multiple applications of orelse and then, the number of component recognizers is is determined by the grammar and is independent of n. Due to the fact that each application of orelse is $O(n)$ and each application of then is $O(n^2)$ irrespective of how many times they are used in building a composite recognizer, the number of component recognizers has no effect on complexity.

In the worst case, each recognizer $r_i \in R$ is applied to each of the n indices at most once. It follows that the total cost is $O(n^3)$.

Proof of $O(n^4)$ complexity for recognizers with direct left recursion.
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 (the left-rec count assures this). It follows that the total cost is $O(n^4)$.

Proof of polynomial complexity for recognizers which use contexts to accommodate indirect left recursion.
In the worst case, where every non-terminal is involved in the path of indirect recursion for every other non-terminal, each recognizer $r_i \in R$ could be applied to each of the n indices n * nt times, where nt is the number of non-terminals. This worst case situation would only occur if, for all recognizers, their results were curtained by the most constraining context first, and subsequent curtailments were progressively less constrained. However, in many grammars, indirect left recursion only involves a few other non-terminals. Also, in many cases less-constrained results would be computed before more-constrained contexts were met, thereby allowing early reuse of results. In any case, the complexity remains $O(n^4)$ with respect to the length of the input.

Appendix 3 — Complexity of Parsers and Size of the Compact Representation

The only difference between parsers and recognizers is that parsers construct and store parts of parse trees rather than end points of recognition. In this appendix, we extend the complexity analysis of recognizers to that of parsers. We begin by defining a "parse tuple" as 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 one of the possibly-many parse tree returned by p for the sequence of tokens from s to e. (Note that this corresponds to entry in the compact representation. An example parse tuple, from the example in section 2 of the paper is:

```
("s", (1,8),[Branch [SubNode ("np",(1,2)), SubNode ("vp",(2,8))]])
```

Parsers corresponding to grammars in Chomsky Normal Form (CNF) will return, in the worst case, for each start/end point pair (s, e) , $(((e - s) + 1)) * nt^2)$ parse tuples, where nt 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/end point pair. Note that each of these parse tuples 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.

In order to convert basic recognizers to parsers, we modify them so that they return leaves containing the terminal together with the end point. This has no effect on complexity.

In order to combine parsers, we modify `orelse` and `then` to manipulate parse tuples rather than endpoints. Note that whole parse trees are not manipulated, only parse tuple

The operator `orelse` now merges sets of parse tuples rather than endpoints. For each of the $O(n)$ endpoints, orelse has to merge two sets of $O(n)$ parse tuples. This raises the complexity of `orelse` to $O(n^2)$

Fortunately, there is no effect on the complexity of `then` due to the fact that the sequencing of two parsers `p then q` only requires that `q` be applied to the $O(n)$ endpoints returned by `p` and, for each of the $O(n)$ successful applications of `q`, the creation of the parse tree. This requires only the construction of a root node combined with a reference to the group of parse trees returned by `p` and the group of parse trees returned by `q`. The complexity of `then` remains $O(n^2)$. Owing to the fact that the complexity of `then` had the highest exponent in the process of applying 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.

It follows that for grammars in Chomsky Normal Form the complexity of non-left recursive parsers is $O(n^3)$ and of left-recursive parsers it is $O(n^4)$.

The compact representation of parse trees that is stored 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 the `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. If the parser corresponds to a CNF grammar, each of these sub-entries contains $O(n)$ parse tuples as discussed above. It follows that the size of the compact representation, in this case, is $O(n^3)$

Appendix 4 — Haskell code

The following is the complete code which implements the new algorithm. If the paper is accepted, we shall provide a more carefully commented version of this code on a web page with reference to it in the paper.

```
import List

data Tree a = Leaf a
            | Branch [Tree a]
            | SubNode (NodeName, (Start,End))
              deriving (Eq,Ord,Show)


type NodeName = String
type Start = Int
type End   = Int

{--
    mtable is list of (parse-name, (start-position,
                              (Context, List-of-results)):rest):rests
    Context  is a pair ([reasons-for-curtailment], [left-rec-context])
```

```
    Result is a pair ((start-position, end-position),
                              [one-level-depth n-ary parse-trees])
--}

type mtable = [(String,[(Int,(Context,[Result]))])]
type Result    = ((Start, End),[Tree String])
type State     = mtable
type StateM t = State -> (t, State)
type Context  = ([String],[(Int,[(String, Int)])])

-- state-monad definition
unitS :: t -> StateM t
unitS x = f where f t = (x,t)

bindS :: StateM t1 -> (t1 -> StateM t2) -> StateM t2
m `bindS` k = f
              where f x = (b,z)
                          where (b,z) = k a y
                                        where (a,y) = m x


-- `orelse` combinator
(p `orelse` q) inp cc = p inp cc  `bindS` f
                         where
                         f (l1,m) = q inp cc  `bindS` g
           where
           g (l2,n)  = unitS ((union (fst l1) (fst l2),[]) ,(m ++ n))

-- Note that, we need to unite the 'reasons-for-curtailment'


-- `then` combinator
-- cc is 'current left-rec context'
-- l is of type 'Context'

(p `thenS` q) inp cc = p inp cc `bindS` f
                        where
                        f (l,m) = apply_to_all q m l cc



apply_to_all q [] l cc     = unitS ((fst l,[]),[])
apply_to_all q (r:rs) l cc = (q `add_P` (r,cc,l)) `bindS` f
                            where
                            f (l1,m)  = ((apply_to_all q rs l cc) `bindS` h)
                                        where
                                        h (l2,n) = unitS ((union (fst l1)
                                                    (fst l2),[]),( m ++  n))

-- like `orelse`, reasons for curtailments are united
q `add_P` (rp,cc,l) = (q (pickEnd rp) cc) `bindS` f
                    where
                    f (l1,m) = unitS ((union (fst l) (fst l1),[]),(addP
m rp))

-- selecting the end-position
pickEnd ((s,e),t) = e
addP0 m rp | m == [] = []
           | otherwise = addP m rp
```

```
addP [] ((s1,e1),t1)                     = []
addP (((s2,e2),t2):restQ) ((s1,e1),t1)  = ((s1,e2),
                             addToBranch ((s2,e2),t2)  ((s1,e1),t1))
                                           : addP restQ ((s1,e1),t1)


-- n-ary branching
-- There could be 9 cases.
-- If there already exists a branch, we just add the new candidate to
the end .
-- If two candidates are two branches, we append them.
-- Otherwise, we form a new branch with non-branch candidates.

addToBranch ((st2,en2),((SubNode (name2,(s2,e2))):ts2)) ((st1,en1),
                                       ((SubNode (name1,(s1,e1))):ts1))
       = [Branch [(SubNode
(name1,(st1,en1))),(SubNode (name2,(st2,en2)))]]
addToBranch ((st2,en2),((Branch t2):ts2)) ((st1,en1),((Branch t1):ts1))
       = [Branch (t1++t2)]
addToBranch ((st2,en2),((Branch t2):ts)) ((st1,en1),((SubNode
(name1,(s1,e1))):ts1))
       = [Branch ((SubNode (name1,(st1,en1))):t2)]
addToBranch ((st2,en2),((SubNode (name2,(s2,e2))):ts2))
((st1,en1),((Branch t1):ts))
       = [Branch (t1++[(SubNode
(name2,(st2,en2)))])]
addToBranch ((st2,en2),((SubNode (name2,(s2,e2))):ts2)) ((st1,en1),[Leaf
x])
       = [Branch [(SubNode (("Leaf "++x)
,(st1,en1))),(SubNode (name2,(st2,en2)))]]
addToBranch ((st2,en2),[Leaf x]) ((st1,en1),((SubNode
(name1,(s1,e1))):ts1))
        = [Branch [(SubNode
(name1,(st1,en1))),(SubNode (("Leaf "++x),(st2,en2)))]]
addToBranch ((st2,en2),((Branch t2):ts)) ((st1,en1),[Leaf x])
       = [Branch ((SubNode (("Leaf
"++x),(st1,en1))):t2)]
addToBranch ((st2,en2),[Leaf x]) ((st1,en1),((Branch t1):ts))
       = [Branch (t1++[(SubNode (("Leaf
"++x),(st2,en2)))])]
addToBranch ((st2,en2),[Leaf x2]) ((st1,en1),[Leaf x1])
                   = [Branch [(SubNode (("Leaf "++x1),(st1,en1))),
                              (SubNode (("Leaf "++x2),(st2,en2)))]]




-- empty and term combinators
empty x l = unitS (([],[]),[((x,x), [Leaf "empty"])])

term c r l |r - 1 == length input     = unitS (([],[]),[])
           |input !! (r - 1)  == c    = unitS (([],[]),[((r,r+1),[Leaf
[c]])])
    |otherwise               = unitS (([],[]),[])


----Memoize----
-- left-rec count check is done within the second element of 'context'
--  during recursive-descent only and not saved in mtable. If the parse
--  is curtailed, its name is returned as the 'reason'
```

```
memoize name f inp context mtable
       | table_res /= []           = ((fst1 table_res,(addNode name inp
                                             (snd1 table_res))), mtable)

       | (funccount (snd context)) > ((length input) - (inp-1) )
                                   = ((([name],[]),[]), mtable)
       | table_res == []           = (( l1 ,(addNode name inp
newRes)),udtTab)


            where
            fst1 [(a,b)] = a
            snd1 [(a,b)] = b

     udtTab      = (udt ((l1,newRes),mft) name inp)

-- we select only those parsers and counts in the left-rec-context
-- that have a match in the set of parsers that have caused curtailment
-- in the subtrees below the current node.
l1 = makeContext (fst l) (findContext (snd context))
          where
        findContext []                          = []
        findContext ((st,rest):sr) | st == inp = [(st,rest)]
                                   | otherwise = findContext sr

          table_res   = lookupT name inp (snd context) mtable


          funccount []                 = 0
          funccount ((key,funcp):rest) | key == inp = findf funcp
                                  | otherwise  = funccount rest

           where
          findf  []           = 0
          findf  ((tk,fc):rx) | tk == name = fc
                                        | otherwise  = findf rx


        ((l,newRes),mft) = ((fst res,packAmb $ sort (snd
res)),newtable)

    where

       (res, newtable)  = f inp ([],(incContext (snd
context) name inp)) mtable




makeContext [] [(st,((n,c):ncs))]          = ([],[])
makeContext (r:rs) []                      = ((r:rs),[])
makeContext [] []                          = ([],[])
makeContext (r:rs) [(st,((n,c):ncs))]
                   = ((r:rs), [(st,makeContext_ (r:rs) ((n,c):ncs))])

makeContext_ [] ((n,c):ncs)      = []
makeContext_ (r:rs) ((n,c):ncs)  = makeContext__ r ((n,c):ncs)
                                       ++ makeContext_ rs ((n,c):ncs)
```

```
makeContext__ r [] = []
makeContext__ r ((n,c):ncs) | r == n    = (n,c): makeContext__ r ncs
                            | otherwise = makeContext__ r ncs


incContext [] name inp  = [(inp,[(name,1)])]
incContext ((st,((n,c):nc)):sn) name inp
      | st == inp = ((st, (addNT ((n,c):nc)) name inp ) :sn)
      | otherwise = ((st,((n,c):nc)): incContext sn name inp
)


addNT []  name inp                     = [(name,1)]
addNT ((n,c):nc) name inp  | n == name = ((n,(c + 1)):nc)
                           | otherwise = ((n,c):addNT nc name inp)



addNode name inp []         = []
addNode name inp (((s,e),t):rs) = ((s,e),[SubNode (name,(s,e))])
                                                 :addNode name inp rs


-- local-ambiguity packing for compact forest

packAmb []                                                = []
packAmb [((s1,e1),t1)]                                    =
[((s1,e1),t1)]
packAmb [((s1,e1),t1),((s2,e2),t2)]  | (s1,e1) == (s2,e2) = [((s2,e2),
t1++t2)]
                                     | otherwise          =
[((s1,e1),t1),((s2,e2),t2)]


packAmb (((s1,e1),t1):((s2,e2),t2):xs)
        | (s1,e1) == (s2,e2) = packAmb (((s2,e2), t1++t2):xs)
        | otherwise          = ((s1,e1),t1):packAmb (((s2,e2),t2):xs)

----Look Up----
-- saved left-rec-context  is compared with the current
left-rec-context.
-- The result is only reused if, for each parser and index, the left-rec
-- count in the saved left-rec context is less than or equal to
-- the left-rec count in the current left-rec context.

lookupT name inp context mtable
                    | res_in_table == [] = []
                    | otherwise  = checkUsability inp context (lookupRes
                                                 (res_in_table !! 0)
inp)

      where
      res_in_table = [pairs|(n,pairs) <- mtable,n ==
name]



lookupRes [] inp                       = []
lookupRes ((i,res):rs) inp | i == inp  = [res]
                           | otherwise = lookupRes rs inp
```

```
checkUsability inp context [] = []
checkUsability inp context [((re,sc),res)]
                     | re == []  =  [((re,sc),res)]
                     | otherwise =  checkUsability_ (findInp inp context)
                                            (findInp inp sc) [((re,sc),res)]


findInp inp []                     = []
findInp inp ((s,c):sc) | s == inp  = c
                       | otherwise = findInp inp sc

checkUsability_ [] [] [(sc,res)]            = [(sc,res)]
checkUsability_ ((n,cs):ccs) [] [(sc,res)]   = []
checkUsability_ [] ((n1,cs1):scs) [(sc,res)] = [(sc,res)]
checkUsability_ ((n,cs):ccs) ((n1,cs1):scs) [(sc,res)]
                | and (memCheck ((n,cs):ccs) ((n1,cs1):scs)) = [(sc,res)]
                | otherwise = []

memCheck [] ((n1,cs1):scs) = []
memCheck ((n,cs):ccs) ((n1,cs1):scs) = condCheck (n,cs) ((n1,cs1):scs)
                                          ++ memCheck ccs ((n1,cs1):scs)

condCheck (n,cs) ((n1,cs1):scs)
                  | (notElemCheck (n,cs) ((n1,cs1):scs)) == [] = []

                  | eqOrGreater (n,cs) ((n1,cs1):scs)          = []

                  | otherwise                                  = [False]



notElemCheck (n,cs) []                      = []

notElemCheck (n,cs) ((n1,cs1):scs) | n /=n1 = notElemCheck (n,cs) scs

                                   | otherwise = [False]

eqOrGreater  (n,cs) []    = True

eqOrGreater  (n,cs) ((n1,cs1):scs) | n == n1 && cs1 <= cs = True

                                   | n == n1 && cs1 > cs  = False

                                   | otherwise            = eqOrGreater (n,cs) scs

----Update----



udt (res, mtable) name inp
               = update mtable name inp res

update [] name inp res                  = [(name,[(inp, res)])]
update ((key, pairs):rest) name inp res
               | key == name = (key,my_merge inp  res pairs):rest
               | otherwise   = ((key, pairs): update rest name inp res)


my_merge inp  res [] = [(inp, res)]
my_merge inp  res ((i, es):rest)
```

```
|inp == i  = (i, res):rest
|otherwise = (i, es): my_merge inp res rest
```