COMPUTER SYSTEMS LAEORATORY

STANFORD UNIVERSITY STANFORD. CA 94305-4055


# AN EFFICIENT TOP-DOWN PARSING ALGORITHM FOR GENERAL CONTEXT-FREE GRAMMARS

**Sriram Sankar**

Technical Report: **CSL-TR-93-562**

(Program Analysis and Verification Group Report No. 62)

**February 1993**

# An Efficient Top-Down Parsing Algorithm for General Context-Free Grammars

Sriram Sankar

**Technical Report:** CSL-TR-93-562
Program Analysis and Verification Group Report No. 62

February 1993

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 943054055

Abstract

This paper describes a new algorithm for top-down parsing of general context-free grammars. The algorithm does not require any changes to be made to the grammar, and can parse with respect to any grammar non-terminal as the start symbol. It is possible to generate all possible parse trees of the input string in the presence of ambiguous grammars. The algorithm reduces to recursive descent parsing on LL grammars.

This algorithm is ideal for use in software development environments which include tools such as syntax-directed editors and incremental parsers, where the language syntax is an integral part of the user-interface. General context-free grammars can describe the language syntax more intuitively than, for example, LALR( 1) grammars. This algorithm is also applicable to batch-oriented language processors, especially during the development of new languages, where frequent changes are made to the language syntax and new prototype parsers need to be developed quickly.

A prototype implementation of a parser generator that generates parsers based on this algorithm has been built. Parsing speeds of around 1000 lines per second have been achieved on a Sun SparcStation 2.

This demonstrated performance is more than adequate for syntax-directed editors and incremental parsers, and in most cases, is perfectly acceptable for batch-oriented language processors.

**Key Words and Phrases:** compilers, top-down parsers, syntax-directed editors, context-free grammars, software development environments

# Contents


1 Introduction 1

2 Terminology 3

3 The Basic Algorithm 3

4 Algorithm A 3

5 Algorithm B (Handling Left-Recursion) 8

6 Algorithm C (Handling C-Productions) 11

7 Building Parse Trees 13

8 Experimental Results 17

9 Comparison with Other Parsing Algorithms 18

10 Some Ideas on Incremental Parsing 20

11 Conclusions and Future Work 20

# 1 Introduction

A significant majority of parsers used in today's software development is based on LR parsing technology [3, 6]. This usually involves coming up with a LALR(l) grammar for the language being parsed and automatically generating a parser based on this grammar. Although the generated parsers are extremely fast, the restriction of having to use LALR(l) grammars poses problems in some situations:

- Recent software development environments contain tools such as syntax-directed editors and incremental parsers where the language syntax is an integral part of the communication between the user and these tools. The language syntax must, therefore, be expressed in as intuitive a manner as possible. Usually, such syntax is not LALR(l). For example, the published grammars of languages such as Ada and C++ are ambiguous.
- The development and maintenance of LALR( 1) grammars can be quite difficult, especially for large languages. Although a grammar is usually written only once for a particular language, this is not the case during the development of new languages, or during the modification of existing languages. Also, some recent languages allow user-definable syntax where the syntax can be modified periodically.

Relaxing the LALR( 1) restriction is essential in these situations. In this paper, we address this by presenting a new algorithm for parsing general context-free grammars. The salient features of the algorithm are discussed below:

- ***Based on recursive descent parsing.***
  The algorithm uses a top-down approach that reduces to recursive descent parsing on LL grammars. Being top-down, this algorithm has the advantage of being used easily for incremental parsing as well as being able to incorporate good error recovery. It is also quite easy to animate the parsing process in a human-understandable way. This allows the user to interact with the parser during the parse process — for example, the user may choose to delete some parse trees during parsing based on an ambiguous grammar,
- ***There is only a minimal parser generation phase.***
  The algorithm works directly off the language grammar. When the algorithm is implemented with lookahead, the algorithm **also** uses sets such as **FIRST** and **FOLLOW.** Computation of these sets is the only activity that needs to be performed during the parser generation phase. This computation takes very little time and can be performed incrementally if the grammar is changed. Hence this algorithm is easily adaptable to changing grammars.
- ***Any grammar non-terminal may be specified to be the start symbol.***
  The algorithm is capable of parsing with respect to any grammar non-terminal as the start symbol with no changes required in the grammar. This capability allows the algorithm to be easily integrated into a syntax-directed editor, and also makes the algorithm adaptable to incremental parsing when only a portion of the input string is changed.

- ***All parse trees may be generated.***
  The algorithm maintains a data-structure which is very similar to a parse tree for the string being parsed. It is quite simple to extract each individual parse tree from this data-structure. If the grammar is LL, then this data-structure is the unique parse tree of the string being parsed. If the grammar is cyclic, the number of possible parse trees may be infinite, in which case the parse tree extraction procedure may not terminate. Extraneous parse trees may be deleted during the parsing process. One way of doing this is by performing semantic checks during the parsing process. In languages such as Ada and C++, this process will usually delete all but one of the parse trees being constructed.

- ***Easy to use with syntax-directed editors.***
  Given all the properties described above, it is quite easy to integrate this algorithm into a syntax-directed editor. The user can be given the option of entering a small portion of a program corresponding to a non-terminal as text, and the parsing process can be animated easily. Our algorithm also allows the easy implementation of a completion facility — for example, while parsing to statement, the template of an if statement can be automatically generated when "if,," is entered.

- ***Complexity.***
  The time complexity of our algorithm is $O(n^3)$ for general context-free grammars, $O(n^2)$ for unambiguous grammars, and O(n) for a large class of commonly used grammars (including LR grammars), where ***n*** is the length of the input string.

- ***Empirical results.***
  A prototype implementation of a parser based on this algorithm has been built and tested on many grammars. This parser uses a lookahead of one token. We have conducted performance experiments using an intuitive (but still complex and ambiguous) Ada grammar. Parsing speeds of around 1000 lines per second have been achieved on a Sun SparcStation 2. This is approximately ***one-third*** the speed of an LALR(1)-based parser for Ada generated using Ayacc [11].

Algorithms have been developed for general context-free grammar parsing, the most well known being Earley's algorithm [4], developed in the late 60's. More recently, Tomita [12] and subsequently Rekers [5, 8] improved on Earley's algorithm — their algorithms reduce to the standard LR parsing algorithms for LR grammars. Other algorithms and general approaches have also been published (*e.g.*, [2, 7, 9, 10]).

For reasons of simplicity of presentation, the algorithm described in this paper uses no lookahead, although the version implemented uses a lookahead of one. It is quite easy to extend this algorithm to introduce lookahead by defining and using sets such as **FIRST** and **FOLLOW.** The data-structure used in the algorithm presented in this paper is similar to the shared forests described in [2] where multiple parse trees are represented as a "forest" of parse trees with common subtrees being shared. In this paper, the data-structure has been customized for top-down parsing and for easy extraction of the individual parse trees at the completion of the parsing process.

**Paper organization.** Sections 2 through 6 introduce terminology and present the algorithm through a series of refinements. Section 7 describes how our algorithm may be extended to construct

parse trees for the input string. Experimental results are presented in Section 8. Section 9 compares our algorithm with other similar algorithms. Section 10 presents some ideas on incremental parsing, and finally, Section 11 concludes this paper.

## 2 Terminology

We represent a context-free grammar as a tuple <N, $\Sigma$, ***P***, S>. N is the set of non-terminals, $\Sigma$ is the set of terminal symbols, ***P*** is the set of productions, and S is the start symbol.

We shall use lower-case letters to represent terminals and upper-case letters to represent non-terminals. We use $\pi$ (possibly subscripted) to represent productions. Strings from $(\Sigma \cup N)^*$ are represented as Greek letters (other than $\pi$).

We use $LHS(\pi)$ to refer to the left-hand side (non-terminal) of production $\pi$. Similarly, $RHS(\pi)$ refers to the right-hand side (expansion) of $\pi$. $RHS(\pi)_i$ is used to refer to the $i^{th}$ symbol of $RHS(\pi)$.

If $A \rightarrow \gamma$ is a production, $\alpha \in \Sigma^*$, and $\beta \in (\Sigma \cup N)^*$ then $\alpha\gamma\beta$ is a ***leftmost expansion*** of $\alpha A\beta$ $(\alpha A\beta \Rightarrow_{lm} \alpha\gamma\beta)$.

If there are strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ in $(\Sigma \cup N)^*$ such that $\alpha_1 \Rightarrow_{lm} \alpha_2 \Rightarrow_{lm} \ldots \Rightarrow_{lm} \alpha_n$, then $\alpha_n$ is a ***leftmost derivation*** of $\alpha_1$ $(\alpha_1 \Rightarrow^*_{lm} \alpha_n)$. If $S \Rightarrow^*_{lm} \alpha$, then $\alpha$ is a ***leftmost sentential form.*** If $\alpha\beta$ is a leftmost sentential form and $\alpha \in \Sigma^*$, then $\alpha$ is a ***leftmost prefix.***

## 3 The Basic Algorithm

In this section, we outline an algorithm that illustrates the basic ideas of our parsing strategy. We progressively refine this algorithm in the subsequent' sections, finally ending up with a polynomial time algorithm for general context-free grammar parsing.

The input string is scanned from left to right one symbol at a time. At any time, if $\alpha$ $(\in \Sigma^*)$ is the portion of the input string scanned so far, the algorithm constructs all leftmost prefixes $\alpha x$ $(x \in \Sigma)$. Each symbol represented by x in the previous sentence represents a legal input symbol that may occur immediately to the right of $\alpha$ in the input string. The set of all such symbols is called the ***frontier set.***

When the next input symbol is scanned, the algorithm determines if this symbol is in the frontier set. If this symbol is in the frontier set, parsing is continued as above after appending the new symbol to $\alpha$. Otherwise, the input string is rejected.

When the last (rightmost) input symbol x is scanned, there must be a leftmost prefix $\alpha x$ (where $\alpha x$ is the input string) that is also a leftmost sentential form. Otherwise, the input string is rejected.

## 4 Algorithm A

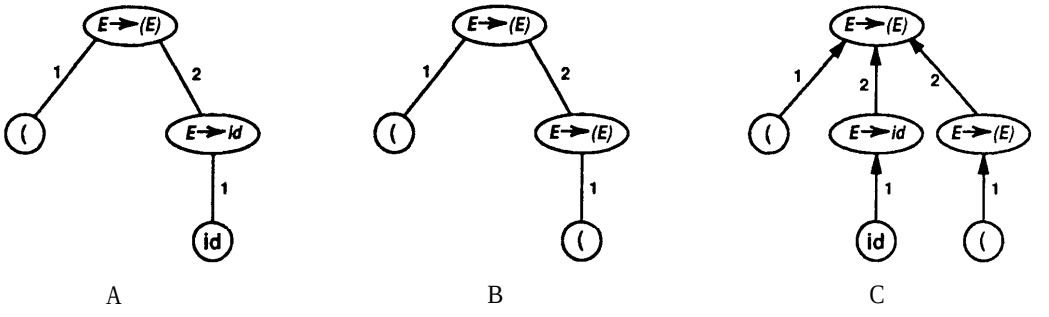
The algorithm description in the previous section left out details such as how the leftmost prefixes are determined. We now present a refinement, Algorithm A, which includes these details.

Intuitively, Algorithm A is a recursive descent parser extended to handle grammars that are not left-factored. Algorithm A does not work on grammars with left-recursion.

In Algorithm A, the leftmost prefixes are maintained along with information on how these prefixes may be derived from the start symbol S. This information is maintained as a set of trees whose roots correspond to S, and whose leaves correspond to symbols in the leftmost prefixes. We refer to these trees as ***partial parse trees.*** As each new symbol is scanned, these trees are updated to determine the new set of leftmost prefixes. The input string is accepted if at least one partial parse tree represents a legal (complete) parse tree of the input string. It is possible for multiple partial parse trees to differ in only minor aspects. The data structures used by the algorithms in this paper take advantage of this and share common portions between these trees. The following example illustrates some partial parse trees and how they are represented in our data structures.

**Example 1:**

Consider the grammar': $N$ = {E}, $\Sigma = \{id, `(', `)', `,'\}$, $\boldsymbol{P} = \{E \rightarrow id, E \rightarrow (E), E \rightarrow (E, E)\}$, and start symbol ***E.*** Figures l.A and l.B are partial parse trees that represent the leftmost prefixes "(id" and "( (" respectively. Each of these trees also indicate how the corresponding leftmost prefixes may be derived from the start symbol. Note that there are other partial parse trees not shown in this figure that represent the same leftmost prefixes, but with different derivations from the start symbol. Figure l.C illustrates how our data structure is used to represent the partial parse trees of Figures l.A and l.B by sharing the common portions of these trees. Note that we can go only from children to parents in this data structure, but this is all that is required by the algorithm. ❑




A B C

Figure 1: **Partial parse trees and their representation**

As the trees of Example 1 illustrate, each node is labeled with either *(1)* a production, or *(2)* a terminal symbol. We use L(n) to refer to the label of a node $n$. In an actual implementation, these labels can be coded as a number that represents the production/terminal number. The pointer to the parent node of a node ***n*** is maintained by ***n*** as a tuple $<p, o>$. The presence of this tuple indicates that the parent of ***n*** is ***p,*** and that the ordinal of ***n*** with respect to ***p*** is o. We use T(n) to refer to the tuple of a node ***n;*** $T(n)_p$ and ***T(n),*** refer, respectively, to the parent and ordinal components of ***T(n).***

**This grammar is a simplified version of part of Ada's expression syntax.**

Algorithm A maintains the frontier set as a set of leaf nodes of the partial parse trees. The labels (terminal symbols) on these nodes constitute the frontier set. After scanning the next input symbol, the new leftmost prefixes and the new frontier set are'obtained in two steps: ***First,*** we traverse up the partial parse trees from the old frontier nodes until we reach nodes which can have siblings to the right. The parents of such nodes are called ***anchors*** from which the partial parse trees may be updated. ***Second,*** we update the partial parse trees by adding new nodes down from the anchors until we reach leaf nodes. These leaf nodes become part of the new frontier set. These two steps are described in detail below as Algorithms A.1 and A.2 respectively. Note the correspondence to recursive descent parsing.

## Algorithm A.1 (FindNextAnchor):

Given a node ***n,*** Algorithm A.1 follows parent edges until it finds a node n' that may have a right sibling. The algorithm then returns the tuple $<T(n')_p, T(n')_o + 1>$. If such a node ***(n')*** is not found, an error return is made.

***FindNextAnchor(n:Node)*** $\rightarrow$ *Tuple*:

**Case** 1 ***(T(n)* is undefined):**
This means that ***n*** has no parent. Make an error return.

**Case *2* (if** $RHS(L(T(n)_p))_i$ **is defined for *i* = *T(n),*** + 1):
Return $<T(n)_p,$ ***T(n),*** + 1>.

**Case 3 (otherwise):**
Return $FindNextAnchor(T(n)_p)$. □

## Algorithm A.2 (ConstructLeftmostChildren):

Given a node ***n*** and an ordinal value i, Algorithm A.2 builds all possible paths from ***n*** to leaf nodes, such that the first edge in the path has ordinal i, and all subsequent edges have ordinal 1. The set of leaf nodes at the ends of these paths is returned on completion.

*ConstructLeftmostChildren(n:Node; i:Integer)* $\rightarrow$ *NodeSet*:

**Case** 1 ***(RHS(L(n));* is a non-terminal):**
For each production $\pi$ such that $LHS(\pi)$ = ***RHS(L(n));,*** create a new node n' with label $\pi$, and tuple ***<n,i>.*** For each of these nodes ***n';*** call $ConstructLeftmostChildren(n',$ ***1)*** and return the union of the node sets returned by each of these calls.

**Case 2 *(RHS(L(n));* is a terminal):**
Create a new node ***n'*** with label $RHS(L(n))_i$ and tuple ***<n,i>.*** Return ***{n'}.*** □

We now describe Algorithm A in two parts — algorithm initialization (INIT) and the scanning operation (SCAN). The frontier is maintained in the variable FRONTIER.

**INIT (algorithm initialization):**

1. For each production $\pi$ such that $LHS(\pi)$ **is** the start symbol, construct a node n and set L(n) to $\pi$. These nodes are the root nodes. ,
2. For each root ***node-n,*** call *ConstructLeftmostChildren(n,1)*, and set FRONTIER to be the union of the node sets returned by each of these calls. □ .

**SCAN (on scanning the next symbol $x$):**

1. If there are no nodes ***n*** in FRONTIER such that ***L(n)*** = x, reject the input string. Otherwise proceed to the following steps.
2. OLDFRONTIER ← FRONTIER, FRONTIER ← $\phi$.
3. For each node ***n*** in OLDFRONTIER such that ***L(n)*** = $x$, do the following:
   (a) $<n', i> \leftarrow FindNextAnchor(n)$.
   (Do not perform the next step if ***FindNextAnchor*** makes an error return.)
   (b) FRONTIER ← FRONTIER ∪ *ConstructLeftmostChildren*($n'$, ***i***). □

**Determination of a successful parse.** The input string is accepted if any call to *FindNextAnchor* during the scanning of the last input symbol results in an error return.

**Example 2:**

Assume the grammar of Example 1. The following scenario describes how Algorithm A parses the string "(id)".

**Initialization.** On initialization, Algorithm A recognizes that all productions have the start symbol, ***E,*** as their left-hand side and therefore creates a root node corresponding to each of these productions. It then applies *ConstructLeftmostChildren* to each of these nodes. The result is the set of trees shown in Figure 2.A.

**Figure conventions.** Nodes are represented as ovals 'with their labels appearing inside them. The tuple of a node is represented as a directed edge leading out of the node to its parent. This edge is labeled with its ordinal. Each node is assigned a unique number and this number appears just outside the oval corresponding to the node. These numbers are used as a convenient way to refer to nodes in the text of the paper. The shaded areas correspond to the frontier sets.

**On scanning "(".** The input symbol "(" matches the labels on the frontier nodes 5 and 6 (Figure 2.A). Therefore, ***FindNextAnchor*** is applied to both these nodes, returning the values $<2,2>$ and $<3,2>$ respectively (the first number in each of these tuples is a node number). When *ConstructLeftmostChildren* is applied on these tuples, we end up with the trees shown in Figure 2.B. Note that unnecessary nodes (1 and 4 in this case) have been omitted for clarity.

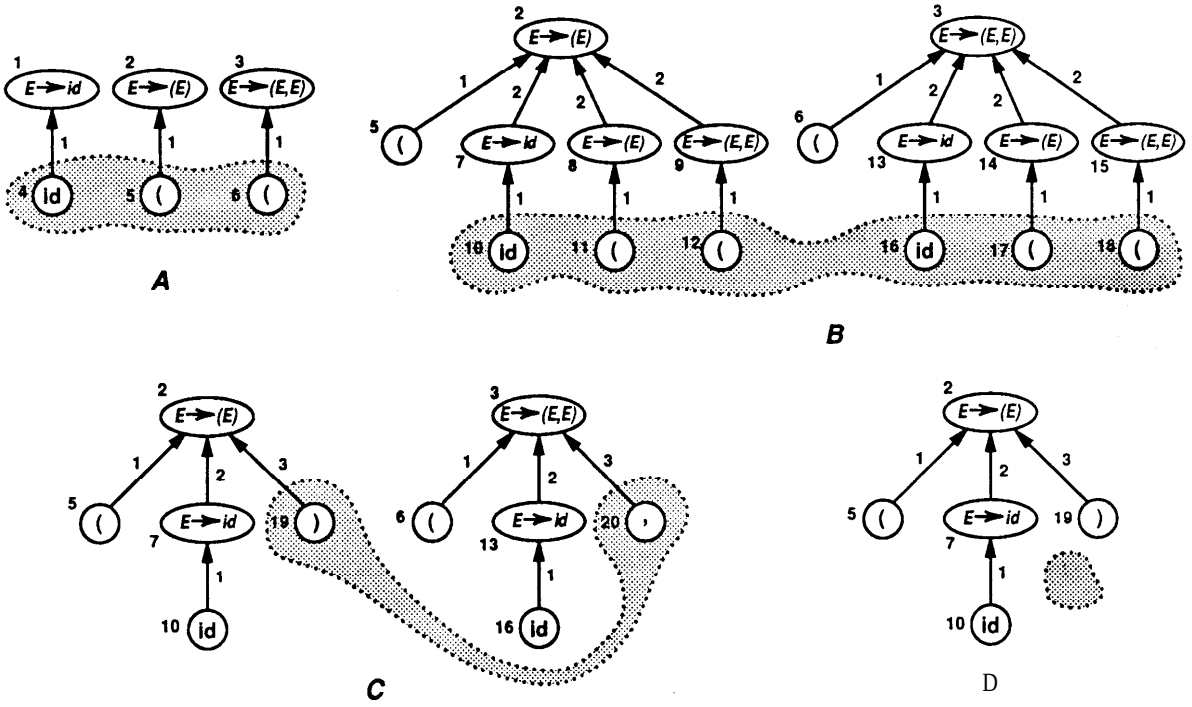



Figure 2: The parse trees generated by Algorithm A

**On scanning *"id".*** The input symbol "id" matches the labels on the frontier nodes 10 and 16 (Figure 2.B). Therefore, ***FindNextAnchor*** is applied to both these nodes, returning the values $<2,3>$ and $<3,3>$ respectively. When *ConstructLeftmostChildren* is applied on these returned values, we end up with the trees shown in Figure 2.C.

**On scanning ")".** The input symbol ")" matches the label on the frontier node 19 (Figure 2.C). Therefore, ***FindNextAnchor*** is applied to this node and results in an error return. Therefore, *ConstructLeftmostChildren* is not invoked. The resulting tree is shown in Figure 2.D. The frontier is empty. Also, since ***FindNextAnchor*** made an error return, the input string is accepted. ❑

### Notes on Algorithm A:

Algorithm A has an exponential complexity with respect to the length of the input string. In Example 2, if the first ***n*** symbols in the input string were all "("s, the frontier set would contain $3 \times 2$" nodes after scanning these ***n*** symbols.

There are two situations in which Algorithm A does not work:

1. If the grammar is left-recursive, *ConstructLeftmostChildren* will not terminate.

2. If the grammar contains e-productions, it is possible for nodes with label $\epsilon$ to be added to FRONTIER. This will cause the parser to miss some of the leftmost prefixes described in Section 3.

# 5 Algorithm B (Handling Left-Recursion)

We now refine Algorithm A **to** work properly in the presence of left-recursion. This refinement will also reduce the time complexity to polynomial time ($O(n^3)$). However, Algorithm B does not handle e-productions. The next and final refinement (Algorithm C) will handle this.

The basic idea in Algorithm B is to reuse nodes with the same label. For example, in Figure 2.A. nodes 5 and 6 have the same label. We could merge them together into just one node. Also, in Figure 2.B, the nodes 7 and 13, 8 and 14, etc. have the same label. Again we could merge each of these pairs into one node each. However, we may only merge nodes that have been created during the same invocation of INIT or SCAN. It **is** all right to merge nodes having the same label and created during the same invocation of INIT or SCAN because any operation performed on one of these nodes is always performed on all the other nodes being merged together. The resulting data structure is no longer a tree — it is a directed graph.

This approach to reusing nodes having the same label creates some overhead. First, nodes may now have multiple parents. We therefore associate a set of tuples with each node rather than just one tuple. We use $TS(n)$ to refer to the tuple set of n. Second, we need to be able to determine whether or not a node with a particular label has been created during the current invocation of INIT or SCAN (to reuse this node), and if so we need to be able to locate this node. This can be done efficiently by maintaining an array in which we keep track of these nodes. For the remainder of this paper we shall assume the existence of the following two functions:

*AlreadyCreated(l:Label)* → ***Boolean:*** Return ***true*** if a node with label $l$ has been created during the current invocation of INIT or SCAN. □

*GetNode(l:Label)* → Node: Makes sense to call this function only if *AlreadyCreated(l)*. Returns the node with label $l$ created during the current invocation of INIT or SCAN. • I

### Algorithm B.l (FindNextAnchor):

Since nodes may have more than one parent, ***FindNextAnchor*** now returns a set of tuples. Also, it is no longer feasible to make an error return because we may hit a root node even when anchors have been found by going up other paths. Therefore ***FindNextAnchor*** sets a boolean global variable SUCCESS to ***true*** to indicate hitting a root node.

***FindNextAnchor(n:Node)*** → *TupleSet*:

1. If n is a root node, set SUCCESS ← ***true.***

2. **TEMP** $\leftarrow \phi$.

3. For each tuple $<p,$ ***i>*** $\in$ ***TS( n),*** if $RHS(\ L(p))_{i+1}$ is defined, add $<p, i+1>$ to **TEMP;** otherwise, add ***FindNextAnchor(p)*** to **TEMP.**

4. Return **TEMP. Cl**

**Algorithm B.2** (ConstructLeftmostChildren):

*ConstructLeftmostChildren(n:Node; i:Integer)* $\rightarrow$ *NodeSet*:

**Case 1** ($RHS(L(n))_i$ **is a non-terminal):**
Set **TEMP** $\leftarrow \phi$. Then for each production $\pi$ such that $LHS(\pi)$ = $RHS(L(n))_i$:

**Case 1.1** ($AlreadyCreated(\pi)$)**:**
Add <***n, i***> to $TS($ ***GetNode(n***

**Case 1.2 (otherwise):**
Create a new node ***n'*** with label $\pi$, and tuple set ***{<n,*** $i>\}$. Call $ConstructLeftmostChildren(n'$, ***1)*** and add the node set returned to **TEMP.**

Return **TEMP.**

**Case 2** ($RHS(L(n))_i$ **is a terminal):**
If $AlreadyCreated($ $RHS($ ***L( n));),*** add < n, i> to $TS($ ***GetNode(*** $RHS(\ L(n))_i)$) and return the empty set; otherwise create a new node n' with label $RHS(L(n))_i$ and tuple set $\{<n,$ ***i>}.*** and return {n'}. • I

**INIT (algorithm initialization):**

Same as INIT of Algorithm A. • I

**SCAN (on scanning the next symbol** $x$**):**

1. SUCCESS $\leftarrow$ ***false.***

2. If there are no nodes ***n*** in **FRONTIER** such that ***L(n)*** = x, reject the input string. Otherwise proceed to the following steps.

3. **OLDFRONTIER** $\leftarrow$ **FRONTIER.**

4. For each node ***n*** in **OLDFRONTIER** such that ***L(n) = x,*** call $ConstructLeftmostChildren(n'$, ***i)*** for each $<n', i>\in$ ***FindNextAnchor(n).*** Set **FRONTIER** to be the union of the node sets returned by each of these calls. 0

**Determination of a successful parse.** The input string is accepted if **SUCCESS** is ***true*** after all input symbols have been scanned.

**Example 3:**

This example illustrates how Algorithm B handles grammars with left-recursion and ambiguity. Consider the grammar: $N$ = ***{E},*** $\Sigma$ = ***{id,*** '+'***}***, $P$ = ***{E → E + E, E → id},*** and start symbol $E$. Figure 3 shows the graphs generated while parsing the string ***"id + id + id".***

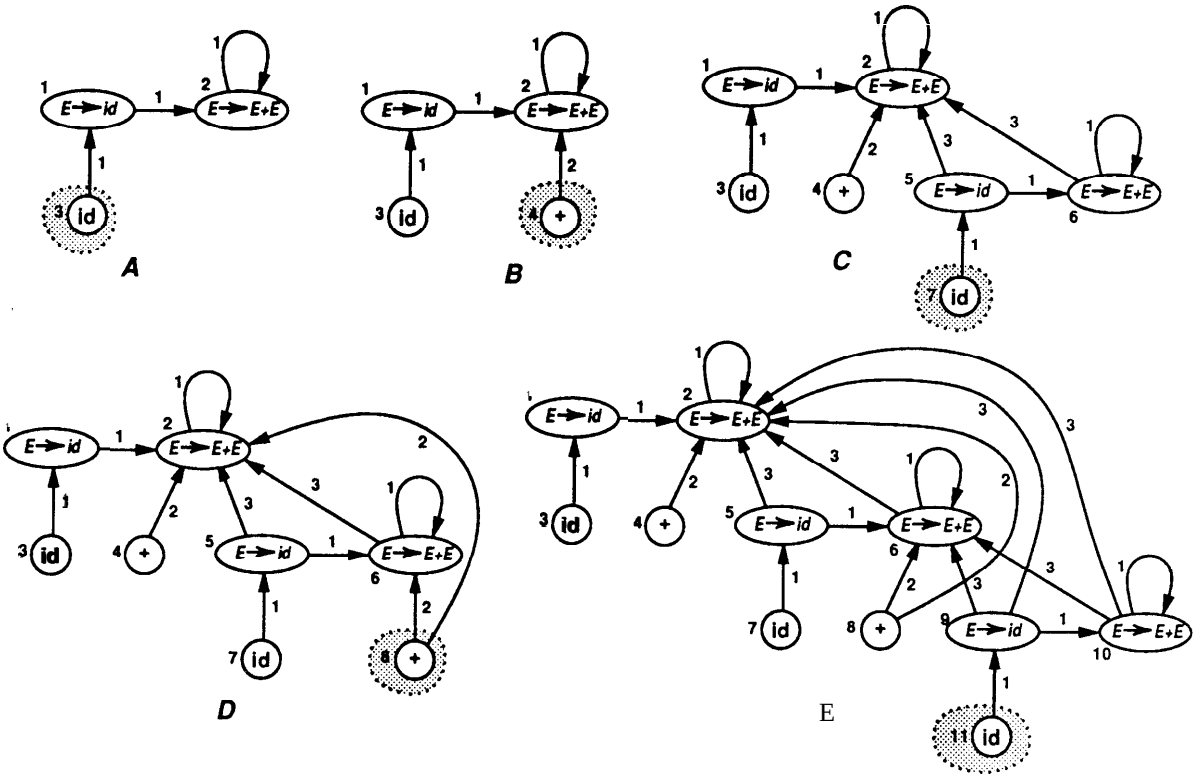



Figure 3: Parsing in the presence of ambiguity

**Initialization.** Nodes 1 and 2 are the root nodes. When ***ConstructLeftmostChiZdren*** is applied on node 1, node 3 is created and added to the frontier set. When ***ConstructLeftmostChiZdren*** is applied on node 2, it is realized that children with labels $E \rightarrow E + E$ and $E \rightarrow id$ have to be constructed for node 2. Since nodes with these labels already exist, they are reused to obtain the graph of Figure 3.A.

**On scanning "*id*".** ***FindNextAnchor*** is applied on node 3, which causes it to be applied recursively on node 1. Since node 1 is a root node, SUCCESS is set to ***true,*** and the tuple set $\{<2,2>\}$ is returned. Applying ***ConstructLeftmostChiZdren*** on this tuple set results in the graph of Figure 3.B.

**On scanning "+".** ***FindNextAnchor*** is applied on node 4 and the tuple set $\{<2,3>\}$ is returned. Applying *ConstructLeftmostChildren* on this tuple set results in the graph of Figure 3.C.

On **scanning "*id*".** ***FindNextAnchor*** is applied on node 7, which causes it to be applied recursively on node 5. Now there are two paths that ***FindNextAnchor*** can take. Taking the path to node 6 causes the tuple $<6,2>$ to be added to **TEMP.** Taking the other path causes *FindNextAnchor* to be applied recursively on node 2. Since node 2 is a root node, **SUCCESS** is set to ***true.*** ***FindNextAnchor*** then takes the self-loop back to node 2 causing the tuple $<2,2>$ to be added to **TEMP.** Applying *ConstructLeftmostChildren* on this tuple set results in the graph of Figure 3.D. Note that the two edges leaving node 8 represent two possible partial parse trees for the leftmost prefix scanned so far.

**On scanning "+".** ***FindNextAnchor*** is applied on node 8 and the tuple set $\{<6,3>, <2,3>\}$ is returned. Applying *ConstructLeftmostChildren* on this tuple set results in the graph of Figure 3.E. □

### A Modification to *FindNextAnchor*:

If $A \Rightarrow^{*}_{lm} A$ is possible for some non-terminal ***A,*** ***FindNextAnchor*** might go into an infinite loop when it encounters productions with ***A*** on the left-hand side. To prevent this, we add the following rule:

> During a particular invocation of INIT or SCAN, ***FindNextAnchor*** may not traverse a particular edge more than once.

In addition to eliminating infinite loops, this rule also makes Algorithm B more efficient by decreasing the amount of tree traversing performed by ***FindNextAnchor.*** This rule is also applied to Algorithm C below.

## 6 Algorithm C (Handling E-Productions)

We now refine Algorithm B to work properly in the presence of c-productions. The basic idea of these refinements is to continue to search for more frontier nodes when c-nodes are encountered. Algorithm C is the final version of the parsing strategy presented in this paper. Nodes may now be labeled with the symbol $\epsilon$.

### Algorithm C.1 (FindNextAnchor):

***FindNextAnchor*** is modified to continue searching for more anchors to the right of anchors from where a derivation to $\epsilon$ is possible.

***FindNextAnchor(n:Node)*** $\rightarrow$ *TupleSet*:

1. If ***n*** is a root node, set SUCCESS $\leftarrow$ ***true.***
2. TEMP $\leftarrow \phi$.
3. For each tuple $<p, i> \in$ ***TS(n):***
   (a) Let ***k*** be the largest number such that for all j (i < j < ***k),*** $RHS(L(p))_j$ is a non-terminal and $RHS(L(p))_j \Rightarrow^*_{lm} \epsilon$ is possible. For each j ***(i*** < j < ***k),*** add $<p,$ ***j***> to TEMP.
   (b) If $RHS(L(p))_k$ is defined, add $<p, k>$ to TEMP; otherwise add ***FindNextAnchor(p)*** to TEMP.
4. Return TEMP. □

## Algorithm C.2 (ConstructLeftmostChildren):

*ConstructLeftmostChildren* is modified to construct more children to the right of nodes from which derivations to $\epsilon$ are possible.

*ConstructLeftmostChildren(n:Node;* ***i:Integer)*** $\rightarrow$ *NodeSet*:

**Case** 0 ***(RHS(L(n))*** **is** $\epsilon$**):**
If $AlreadyCreated(\epsilon)$, add $<n,$ ***i***> to $TS(GetNode(\epsilon))$; otherwise create a new node ***n'*** with label $\epsilon$ and tuple <n,i>. Return the empty set.

**Case 1** ***(RHS(L(n));*** **is a non-terminal):**
Set TEMP $\leftarrow \phi$. Then for each production $\pi$ such that $LHS(\pi)$ = ***RHS(L(n));:***

**Case 1.1** ***(*** $AlreadyCreated(\pi)$**):**
Add $<n, i>$ to ***TS( GetNode(n***

**Case 1.2 (otherwise):**
Create a new node ***n'*** with label $\pi$, and tuple set $\{<n, i>\}$. Let ***k*** be the largest number such that for all j (0 < j < ***k),*** $RHS(\pi)_j$ is a non-terminal such that $RHS(\pi)_j \Rightarrow^*_{lm} \epsilon$ is possible. For each j (0 < j < ***k*** and also for j = ***k*** if $RHS(\pi)_k$ is defined), call $ConstructLeftmostChildren(n'$, j). Add the node sets returned by each of these calls to TEMP.

Return TEMP.

**Case 2** ($RHS(L(n))_i$ **is a terminal):**
If $AlreadyCreated($ ***RHS( L(*** $n))_i$), add <n, i> to ***TS( GetNode( RHS( L(n));))*** and return the empty set; otherwise create **a new** node ***n'*** with label ***RHS( L(n));*** and tuple set {<n, i>}, and return ***{n'}.*** □

**INIT (algorithm initialization):**

1. For each production $\pi$ such that $LHS(\pi)$ is the start symbol, construct a node ***n*** and set ***L(n)*** to $\pi$. These nodes are the root nodes.
2. FRONTIER $\leftarrow \phi$, SUCCESS $\leftarrow$ *false*.
3. For each root node ***n***: Let ***k*** be the largest number such that for all j (0 < ***j*** < ***k***), $RHS(L(n))_j$ is a non-terminal such that $RHS(L(n))_j \Rightarrow^*_{lm} \epsilon$ is possible. For each j (0 < j < ***k*** and also for j = ***k*** if ***RHS(L(n))k*** is defined), call *ConstructLeftmostChildren*($n$, ***j***). Add the node sets returned by each of these calls to FRONTIER. If $RHS(L(n))_k$ is not defined, set SUCCESS $\leftarrow$ *true*. □

**SCAN (on scanning the next symbol $x$):**

Same as SCAN of Algorithm B. □

**Determination of a successful parse.** The input string is accepted if SUCCESS is *true* after all input symbols have been scanned.

# 7 Building Parse Trees

Algorithm C builds a parse graph which closely resembles the parse tree of the string being parsed. In fact, when parsing with respect to a LL( ***k***) grammar with a lookahead of ***k*** symbols, the parse graph produced by Algorithm C is the unique parse tree of the sentence being parsed[2].

When the grammar is not LL, the parse graph gets complicated in two ways: *(1)* Tuple sets may contain more than one tuple. Each of these tuples needs to be considered during the generation of parse trees; *(2)* If the grammar is left-recursive, the parse graph may contain loops. We need to determine how many times each loop needs to be unraveled to produce a parse tree.

The algorithm presented in this section generates parse trees from a parse graph corresponding to a successful parse of an input string. A straightforward approach would be to scan the parse graph from the leftmost leaf to the rightmost leaf building the parse tree in the process. However, in the presence of loops in the parse graph (corresponding to left-recursion), the algorithm would have to guess how many times the loop needs to be unraveled (note that the parse tree of a left-recursive production is left-heavy). This may require the inspection of an unbounded amount of the parse graph to the right. This complicates the straightforward approach.

Quite interestingly, if the scanning order is reversed, i.e., the parse graph is scanned from the rightmost leaf to the leftmost leaf, the problems mentioned above with loops are not encountered. The only situation in which the algorithm will encounter loops in the right to left scanning process

[2] Given the properties of LL grammars, there will be no sharing of parse trees in the parse graph — hence all tuple sets will contain at most one tuple.

is when the grammar is cyclic. The algorithm can unravel such loops any number of times and still produce a valid parse tree.

The general parse tree generation process is as follows: The algorithm takes the rightmost leaf node and constructs a path to one of the root nodes (T). This path will form part of the parse tree and will be such that any portion of the parse tree to the right of this path corresponds to e-productions. Such paths will be formalized below ***as rightmost*** paths. This path will now take the role of an "anchor" onto which the rest of the parse tree to the left will be attached. The next step is to choose an "anchor point", an appropriate node (n) in the anchor path. A rightmost path is then constructed from the next leaf node, ***n'*** (to the left of the current leaf node), to n. This causes a little more of the parse tree to be constructed, and a new anchor path $(n' \rightarrow n \rightarrow r)$ is defined. This process is continued until the leftmost leaf node has been reached. It can be proved that the generated tree rooted at $r$ is a parse tree of the parsed input string. This parse tree may contain missing portions all of which can be filled up with c-expansions.

Before we can present the algorithm formally, we need to extend the parse graph data structure and Algorithm C **as** follows:

1. We form a list of all frontier nodes that match input tokens from the right to the left,. We refer to this list as *TokenList.* In Figure 3.E, *TokenList* starts with node **11,** continuing on to nodes 8, 7, 4, and 3 respectively.

2. Every node in the parse graph is assigned a ***generation*** number. This number is the index of the current token being scanned when the node is created. In Figure 3.E, nodes 1, 2, and 3 have generation 1; node 4 has generation 2; nodes 5, 6, and 7 have generation 3; node 8 has generation 4; and nodes 9, 10, and 11 have generation 5.

**Definition (rightmost paths).** The sequence $n_0 e_1 n_1 \ldots e_m n_m$ (where $n_i$'s are nodes and $e_i$'s are edges) is a path in the parse graph if every $e_i$ is in $TS(n_{i-1})$ and $(e_i)_p = n_i$. In addition, this path is a ***rightmost*** path if for each $e_i$, $RHS(L(n_i))_j \Rightarrow^*_{lm} \epsilon$ is possible for all $j > (e_i)_o$.

**Definition (Rightmost(n,NS,Ord).** ***We*** define ***Rightmost(n, NS, Ord)*** where ***n*** is a node, $NS$ is a set of nodes, and ***Ord*** is an ordinal number, to be the set of all paths from ***n*** to a node in NS such that:

- If ***Ord*** = 0, then each path is a rightmost path.
- If ***Ord*** $\neq$ 0, then each path is of the form n . . . $n_{m-1} e_m n_m$ where $n_m \in$ NS, n . . . $n_{m-1}$ is a rightmost path, and for all ***j*** $((e_m)_o < j < Ord)$, $RHS(L(n_m))_j \Rightarrow^*_{lm} \epsilon$ is possible.

Note that in the earlier informal description of the parse tree building algorithm, the very first path generated is a member of the set $Rightmost(l, RootNodes, 0)$ where $l$ is the rightmost leaf node, and $RootNodes$ is the set of all root nodes in the parse graph. All subsequent paths that are generated are members of the sets ***Rightmost(n', n, i)*** where n' and ***n*** are as defined in the earlier informal description, and i is the ordinal of the edge leading into ***n*** in the immediately previous anchor path.

We are now ready to formally present the parse tree building algorithm. For simplicity, we describe a version of this algorithm that builds one parse tree. At various points this algorithm chooses an arbitrary path from the set *Rightmost*(. . .). Each such choice results in a different set of parse trees. This algorithm can be extended in a straightforward manner to generate all parse trees by iterating over all paths in the set *Rightmost*(. . .). Some extra bookkeeping is required in this case, which may obscure the salient aspects of the algorithm — hence we do not present this version in the paper.

The tree building algorithm uses a recursive routine *BuildTree* which takes three arguments, a parse tree node, a node set, and an ordinal number. The algorithm is started off by calling this routine as *BuildTree*(*T*, *RootNodes*, 0). On return from this call, ***T*** is an entire parse tree of the parsed string except that the c-expansions have not been filled out. *BuildTree* can easily be extended to complete these parts of the tree. Again, we avoid these details in the paper.

*BuildTree(R:***in out** ***Node; NS:NodeSet;*** *Ord:Natural)*:

1. ***Leaf*** t First node in *TokenList*, *TokenList* ← Rest of *TokenList*.
2. Choose a path ***P*** from *Rightmost*(*Leaf*, ***NS, Ord).***
3. Create a path ***P'*** for the parse tree that is identical to ***P.*** If ***Ord*** = 0, set ***R*** to be the last node in ***P'. R*** will be the root of the parse tree generated by *BuildTree*. If ***Ord*** $\neq$ 0, ***R*** will already contain a node — do not create a new node for the very last node of ***P';*** instead use ***R.*** This is effectively adding a new path into an already created partial parse tree.
4. Set ***Ptr*** to point to the first node in ***P*** and repeat the following steps until *Ptr* crosses the end of ***P:***
   (a) Move ***Ptr*** along ***P*** until a node ***n*** with a generation different from that of the previous node is encountered. Let i be the ordinal of the edge through which *n* was reached.
   (b) Make the recursive call *BuildTree*($n'$, ***{n}, i),*** where n' is the node in ***P'*** corresponding to $n$. ***Cl***

**Example 4:**

This example illustrates how *BuildTree* will build parse trees from the last graph of Figure 3.

1. The first call to *BuildTree* results in determining the set *Rightmost*(11, $\{1,2\}$, 0). This set contains two paths: 11 → 9 → 6 → 2 and 11 → 9 → 2. Suppose we choose the first path. The portion of the parse tree constructed is shown in Figure 4.A. *BuildTree* now moves up the path from 11 → 9 → 6. It stops here because the generation number changes here. *BuildTree* then makes the recursive call *BuildTree*($6'$, $\{6\}$, ***3).***
2. The second call to *BuildTree* results in determining the set ***Rightmost(8,*** $\{6\}$, 3), which contains just the path 8 → 6. The parse tree is updated as shown in Figure 4.B. *BuildTree* now moves up the path to node 6 and makes the recursive call ***Build*** *Tree*( $6'$, $\{6\}$, ***2).***

3. The third ***call*** to *BuildTree* results in determining the set ***Rightmost(7,*** $\{6\}$, 2), which contains just the path 7 → 5 → 6. The parse tree is updated as shown in Figure 4.C. Since all nodes in this path have the same generation, this call to *BuildTree* terminates going back to the second ***call*** to *BuildTree* which also terminates.

4. Control is back in the first call to *BuildTree* which now moves ***Ptr*** up from node 6 to node 2, resulting in the recursive call *BuildTree*(2′, { 2}, ***3)*** (since node 2 has a generation different from that of node 6).

5. The fourth call to *BuildTree* results in determining the set ***Rightmost(4,*** $\{2\}$, 3), which contains just the path 4 → 2. The parse tree is updated as shown in Figure 4.D. *BuildTree* now makes the recursive call *BuildTree*(2′, $\{2\}$, ***2).***

6. The fifth ***call*** to *BuildTree* results in determining the set ***Rightmost(3,*** $\{2\}$, 2), which contains just the path 3 → **1** → 2. The parse tree is updated as shown in Figure **4.E. Since all nodes** in this path have the same generation, this call to *BuildTree* terminates going back to the fourth call to *BuildTree* which also terminates. This in turn returns control to the first call to *BuildTree* which also terminates.

If the other path **(11** → 9 → 2) was chosen in the first step of this example, then the parse tree in Figure 4.F would have been constructed. Note that in this case, the self-loop at node 2 is traversed to get two instances of this node (2' and 2") in the parse tree. □

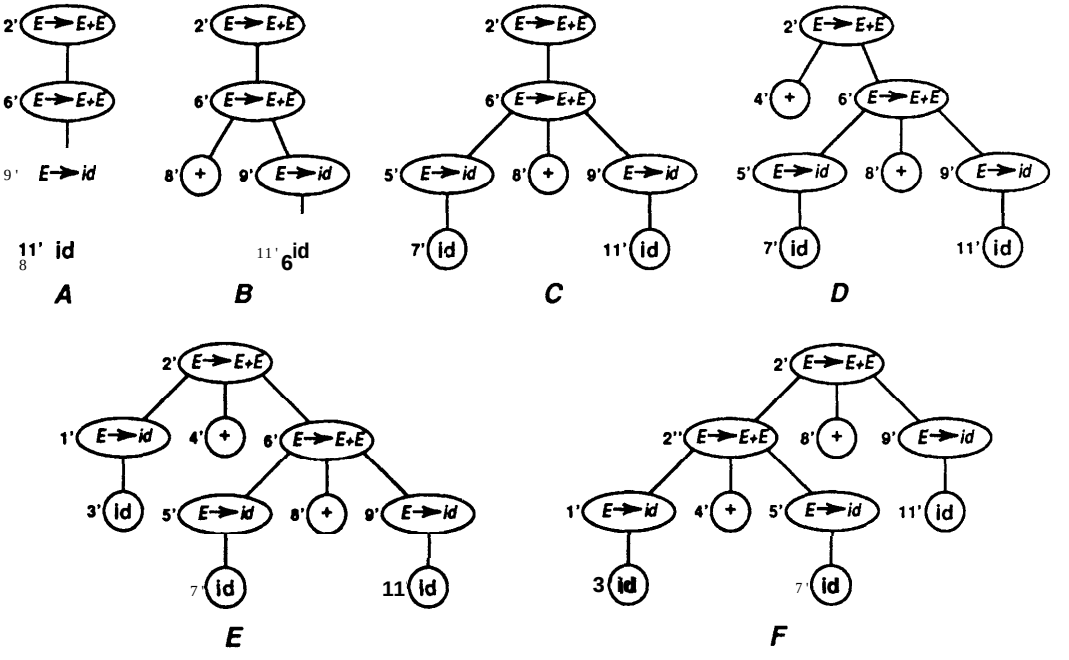



Figure **4: Building parse trees**

**Proof of Correctness Outline.** We now present a proof outline for the correctness of the tree building algorithm. There are three parts to this proof:

1. If the algorithm does terminate successfully, then the data structure it produces is indeed a parse tree for the input string.

2. Regardless of the various arbitrary choices of paths made by the algorithm, the algorithm will terminate successfully.

3. Every parse tree of the input string will be generated by the algorithm.

The notion of "successful termination" is defined as follows: When the algorithm terminates, it has reached the end of *TokenList* and not attempted to search for more entries in this list.

The data structure produced by the algorithm is indeed a tree, and given the assumption of successful termination, the leaves of this tree are the tokens from the input string. The definition of ***Rightmost*** guarantees that there are no missing children on the right of any path. Since these paths were initially constructed by *ConstructLeftmostChildren*, we are also guaranteed that there are no missing children on the left of any path either. Hence this tree must be a complete parse tree and (1) holds.

We now demonstrate that *(2)* holds. Given that the input string has parsed successfully, there has to be a rightmost path from the rightmost leaf node to a root node (the existence of such a. path is what is used to determine a successful parse). The generation number of the leaf node in this path is the index of the last token in the input string, and that of the root node is 1. The following observation together with the fact that the generation numbers of nodes in any path do not decrease as we go closer to leaf nodes guarantees that the algorithm will terminate successfully.

***If P is a path between n and a leaf node l such that there is no other node in P** with the same **generation as that of n, then there has to be a path between n and another leaf node to the** left of l **which does not intersect with P. Conversely, if there are other nodes in P with the** same generation as **that** of **n, then there cannot be any paths from n to leaf nodes to the** left of **1 that do not intersect with P.***

This observation also justifies the method used to select anchor points. In fact, the second part of this observation indicates that no other method of selecting anchor points will produce a parse tree. This demonstrates that (3) holds.

*(A **more formal and complete proof** of **correctness can be made in the final** version, if desired by **the referees and space permits.)***

# 8 Experimental Results

We have developed a prototype implementation of a parser based on Algorithm C with a lookahead of one token. The parser is written in Ada. We have also implemented a syntax-directed editor and have integrated this parser with the editor.

We have tested our parser on many grammars starting with a few simple languages during initial feasibility tests, and then on various variants of Ada grammars from a LALR( 1) grammar on the one extreme to ambiguous grammars on the other. We are also using this parser on Rapide [l], a language being designed at Stanford University. The syntax of Rapide has been changed frequently

during the language design **process,** and we have been able to keep the parser up-to-date with the changes in the grammar quite easily.

We conducted performance experiments with this parser on an Ada grammar. This grammar was nearly a verbatim copy of the Ada syntax in the Ada Language Reference Manual and is ambiguous in a few places. We compared the performance of our algorithm to that of a parser generated using Ayacc[3] [11] for a LALR(l) Ada grammar. Some of the important aspects of this experiment were:

- All attribute rules were removed from the LALR( 1) Ada grammar.
- Lexical analyzer operations were not timed.
- The entire heap space was allocated by our algorithm during initialization and was not timed.

Figure 5 represents the parsing of 50 randomly chosen Ada programs ranging from 0 to 1000 lines long on a Sun SparcStation 2 using the two parsers. The size of the program (in lines) is shown on the x-axis, and the time taken (in seconds) to parse this program is shown on the y-axis.

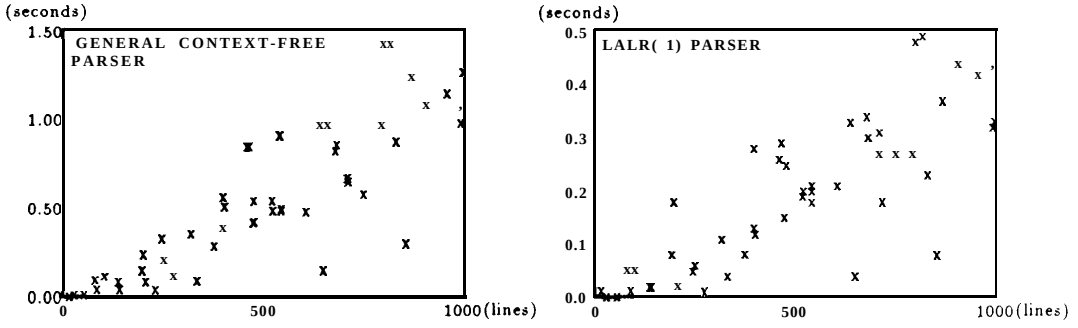



Figure 5: **Plot of execution time versus size of program**

These performance figures show that the general context-free parser parses at approximately 1000 lines per second on the Sun SparcStation 2, and runs at approximately one-third the speed of the Ayacc generated parser. This demonstrated performance is more than adequate for syntax-directed editors and incremental parsers, and in most cases, is perfectly acceptable for batch-oriented language processors.

# 9 Comparison with Other Parsing Algorithms

## Earley's Algorithm

Earley's algorithm is quite similar to LR-parsing in that the algorithm goes through a sequence of ***states,*** each of which contains a set of ***items.*** However, Earley's algorithm constructs items during the parsing process as opposed to during the generation of parsers in the LR case.

[3] Ayacc **stands for Ada-yacc and generates parsers in Ada.**

There is a strong correspondence between Earley's algorithm with no lookahead and the algorithm presented in this paper. Earley's ***predictor*** performs the same operations as *ConstructLeftmostChildren;* Earley's ***scanner*** performs the same operations as SCAN; and Earley's ***completer*** performs the same operations as *FindNextAnchor.*

The items in Earley's algorithm have the following correspondence with our parse graphs: (1) The productions correspond to the parse graph nodes; (2) The point in the production indicating how much of the production has been recognized corresponds to the ordinal of edges arriving at the node (multiple edges with different ordinals define multiple items); and (3) The pointer back to the position in the input string at which this instance of the production was created corresponds to the edges leaving the node in the parse graph (multiple edges leaving a node define multiple items). The fourth entry in an item in Earley's algorithm is the lookahead — since we are considering parsing with no lookahead, this entry is irrelevant. The closure of all nodes reachable from the frontier set along with the ordinals of the edges used to reach these nodes is identical to the set of items that Earley's algorithm would generate in the same situation.

The main differences between the two algorithms are:

1. Earley's algorithm constructs new items for each new state, while our algorithm reuses existing nodes by incrementally updating the partial parse tree/graph.
2. In the presence of lookahead, Earley's ***predictor*** has to work harder while *ConstructLeftmostChildren* has to do less work. i.e., The two algorithms start to differ in the presence of lookahead.

Given this correspondence, all the complexity results for Earley's algorithm also apply to our algorithm. Hence, the time complexity of our algorithm is $O(n^3)$ for general context-free grammars, and $O(n^2)$ for unambiguous grammars, where ***n*** is the length of the input string. With a lookahead of ***k*** symbols, our algorithm can parse LR(k) grammars in O(n) time.

***(A more formal comparison of the two algorithms can be made in the final version, if desired by the referees and space permits.)***

## Rekers' Algorithm

Rekers [5, 8] has developed a general context-free bottom-up parsing algorithm. This algorithm is a refinement of Tomita's algorithm [12] and has a parser generation phase when an LR parse table is constructed based on the grammar. Given that the grammar can be non-LR, individual parse table entries can contain multiple rules. These conflicts are handled by starting off parallel parsers to handle each of these rules. These parsers are merged when the tops of their respective stacks become identical, or they pop the stack beyond the point where they split up in the first place. Some parsers may reach reject states in which case they simply die. The input string is accepted if at least one of the parsers reaches an accept state on scanning the last token. These parsers are set up to build a forest a parse trees during the parsing process, so that at the end of the parse, all parse trees are available.

Although Rekers' algorithm is exponential in the worst case (for ambiguous grammars,), its performance is similar to our algorithm for typical grammars. Hand simulations indicate that Rekers' algorithm performs slightly better, and this is a direct consequence of having generated parse tables ahead of time. The main advantages of our algorithm over Rekers' algorithm translates to the standard advantages of LL parsers over LR parsers. Also, with more complex grammars, the performance of Rekers' algorithm may degrade since it builds all parse trees during the parsing process while our algorithm builds them later. At this later stage, we have the option of choosing not to generate all parse trees.

## 10 Some Ideas on Incremental Parsing

Although much work needs to be done on applying the algorithm presented in this paper to incremental parsing, this section presents some preliminary ideas. Given a string $\alpha\beta\gamma$ that has been parsed already, suppose we make a modification to result in the string $\alpha\delta\gamma$. How can we minimize the amount of work involved in parsing the modified string?

To use the algorithm presented in this paper, the parse graph and frontier sets of the original string needs to be retained. When the modified string is presented to the incremental algorithm:

- The parse graph of $\alpha$ remains the same. Hence the parsing process needs to start only at the beginning of $\delta$ using the original frontier set realized at the end of parsing $\alpha$.
- During parsing of 6, the parse graph of $\beta$ may be reused if the same structure needs to be recreated. For example, if a portion of $\delta$ parses to the same non-terminal as a similar portion of $\beta$, there is potential for reuse of the parse graph of $\beta$.
- If the frontier set at the end of parsing $\delta$ is the same as that realized at the end of parsing $\beta$, the incremental parsing process is done. Otherwise we have to continue parsing into $\gamma$ until the frontier sets match.
- A problem that needs to be addressed is the generation numbers attached to the nodes in the parse graph. Since the lengths of $\beta$ and $\delta$ will usually differ, some kind of renumbering scheme needs to be devised.

## 11 Conclusions and Future Work

We have presented a capability whereby general context-free grammars may be used to describe languages. This flexibility comes at the cost of a slight loss in performance. However, recent improvements in hardware technology have, to some extent, compensated for this performance loss. Parsing speeds of 1000 lines per second are quite acceptable for software being developed today.

We are currently integrating our parser generator with a syntax-directed editor generator. We are working on enhancing our algorithm with animation and completion capabilities. Further studies are required to determine how grammar attributes may be used in our framework and how our algorithm can be generalized for full incremental parsing.

## References


[1] F. Belz and D. C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In ***Proceedings of the ACM Tri-Ada Conference,*** Baltimore, December 1990. ACh4 Press.

[2] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings* ***of the 27th Annual Meeting of the Association for Computational Linguistics,*** pages 143–151, June 1989.

[3] F. DeRemer. Simple LR(k) grammars. ***Communications of the ACM,*** 14(7):453–460, July 1971.

[4] J. Earley. An efficient context-free parsing algorithm. ***Communications of the ACM,*** 13(2):94-102, February 1970.

[5] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. In *Proceedings of the* ***SIGPLAN Conference on Programming Language Design and Implementation,*** pages 179- 191, June 1989.

[6] D. E. Knuth. On the translation of languages from left to right. ***Information and Control,*** 8(6):607–639, 1965.

[7] B. Lang. ***Deterministic Techniques for*** *Efficient* ***Non-Deterministic Parsers,*** volume *14* of ***Lecture Notes in Computer Science,*** pages 255-269. Springer-Verlag, 1974.

[8] J. Rekers. ***Parser Generation for Interactive Environments.*** PhD thesis, University of Amsterdam, The Netherlands, January 1992.

[9] Y. Schabes. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars, In ***Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics,*** pages 106-113, June 1991.

[10] T. Schöbel. A new parsing strategy for context-free grammars. Technical Report 7/91, Universität Stuttgart Fakultät Informatik, May 1991.

[11] D. Tolani and D. Taback. Preliminary Ayacc user's manual. Technical Report UCI-85-10, University of California, Irvine, September 1985.

[12] M. Tomita. *Efficient* ***Parsing for Natural Languages.*** Kluwer Academic Publishers, Boston, 1986.