IMPLEMENTATION OF A SLR(1) PARSING ALGORITHM

By

JOSEPH LEE GRAY
Bachelor of Arts
California State University at Long Beach
Long Beach, California
1971

Submitted to the Faculty of the Graduate College of the Oklahoma State University in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE May, 1973
IMPLEMENTATION OF A SLR(1)

PARSING ALGORITHM

Thesis Approved:

[Signatures]

Thesis Adviser

[Signatures]

DD Fisher

[Signatures]

Dean of the Graduate College
PREFACE

This thesis is a description of the SLR(1) parsing algorithm. The advantage of using SLR(1) techniques in syntax analyzers is the generality and efficiency over other parsing schemes. The description is designed to appeal to the reader's academic as well as implementation interests.

Thanks are due to Dr. Donald Fisher and Dr. George Hedrick for their suggestions for improvement of this thesis and especially to my major adviser, Dr. James Van Doren, who, above everything else, asked me questions that made me think.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. CONTEXT-FREE GRAMMARS</td>
<td>4</td>
</tr>
<tr>
<td>Definitions</td>
<td>4</td>
</tr>
<tr>
<td>Parsing</td>
<td>10</td>
</tr>
<tr>
<td>Relations and Closures of Relations</td>
<td>15</td>
</tr>
<tr>
<td>Practical Restrictions on CF Grammars</td>
<td>23</td>
</tr>
<tr>
<td>III. LEFT TO RIGHT TRANSLATION OF LANGUAGES</td>
<td>26</td>
</tr>
<tr>
<td>The LR(k) Method</td>
<td>26</td>
</tr>
<tr>
<td>The SLR(1) Method</td>
<td>35</td>
</tr>
<tr>
<td>Comparison of Table Construction Methods</td>
<td>45</td>
</tr>
<tr>
<td>IV. CONCLUSION</td>
<td>51</td>
</tr>
<tr>
<td>A SELECTED BIBLIOGRAPHY</td>
<td>53</td>
</tr>
<tr>
<td>APPENDIX A - LIST OF SYMBOLS</td>
<td>56</td>
</tr>
<tr>
<td>APPENDIX B - USER'S GUIDE</td>
<td>58</td>
</tr>
<tr>
<td>APPENDIX C - PROGRAM LISTING</td>
<td>66</td>
</tr>
<tr>
<td>APPENDIX D - LOGIC BLOCK DIAGRAM</td>
<td>80</td>
</tr>
</tbody>
</table>
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Configuration Sets - LR(1) Method on G₂</td>
<td>30</td>
</tr>
<tr>
<td>II. LR(1) Configuration Sets for G₃</td>
<td>31</td>
</tr>
<tr>
<td>III. LR(1) Table for G₃</td>
<td>34</td>
</tr>
<tr>
<td>IV. The &quot;Overlay&quot; Modification of Table III</td>
<td>37</td>
</tr>
<tr>
<td>V. LR(0) Configuration Sets for G₁</td>
<td>38</td>
</tr>
<tr>
<td>VI. SLR(1) Table for G₁</td>
<td>39</td>
</tr>
<tr>
<td>VII. SLR(1) Configuration Sets for G₃</td>
<td>46</td>
</tr>
<tr>
<td>VIII. SLR(1) Configuration Sets for G₄</td>
<td>48</td>
</tr>
<tr>
<td>IX. SLR(1) Table for G₄</td>
<td>49</td>
</tr>
</tbody>
</table>


**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Syntax Tree for $S^+?i+i?$</td>
<td>8</td>
</tr>
<tr>
<td>2. Syntax Tree for $S^+?i_{0}^{1}+i_{1}^{2}+i_{2}^{3}+i_{4}^{4}*(i_{5}^{5}+i_{6}^{6})?$</td>
<td>8</td>
</tr>
<tr>
<td>3. Boolean Matrix Representation of $H(G_1)$</td>
<td>17</td>
</tr>
<tr>
<td>4. Graph Representation of $H(G_1)$</td>
<td>18</td>
</tr>
<tr>
<td>5. Boolean Matrix Representation of $H^+(G_1)$</td>
<td>19</td>
</tr>
<tr>
<td>6. Graph Representation of $H^+(G_1)$</td>
<td>20</td>
</tr>
<tr>
<td>7. Parsing $?affc?$ Using Table III</td>
<td>34</td>
</tr>
<tr>
<td>8. Parsing $?i+i+i?$ Using Table VI</td>
<td>41</td>
</tr>
<tr>
<td>9. Parsing $?abc?$ Using Table IX</td>
<td>49</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

This thesis is a presentation of a reasonably general method for parsing and gaining conceptual insight into languages described by context-free (CF) grammars. Included are the definition of a CF grammar, a development of some of the characteristics of a CF grammar, and the definition and construction of a general parsing scheme for a significant subset of CF languages. The purpose is to show how to develop certain conceptual characteristics of any particular CF language and at the same time mechanically construct a table-driven syntax analyzer for that grammar by using the method for table construction contained herein. The former is particularly valuable for languages with which the reader is not intimately familiar.

The main area of applicability is in writing translators for computer programming languages. In particular, the parsing method applies to a large subset of CF languages written in Backus-Naur Form (BNF) in which most of the commonly used programming languages can be described approximately. Syntax analyzers are only part of the compiling process and are usually intertwined with other parts (semantic routines, scanners, code generators, etc.); however, this paper isolates the syntax analyzer for the purpose of examination.

A useful side effect of the table construction method is that an understanding of the grammar and the language may be obtained even if
the complete table cannot be generated for a particular grammar. Hence, this thesis will serve as a useful guide for studying programming languages for which no compiler is available if the user can express the grammar in BNF.

There has always been a decision between whether to program in a low-level language such as assembler or machine language, which is difficult, machine dependent, and fast in terms of translation time, or in a high-level language such as FORTRAN, which is easier to do, easier to train personnel for, and machine independent, but slower in translation time and perhaps not applicable to a particular problem. At this time, the consensus seems to be that the high-level languages are more desirable; therefore, one goal of the computer scientist is to correct the deficiencies. The solution is to write several high-level languages for different areas of applicability and to write efficient translators for them. Out of this goal have come translator writing systems (TWS) of which one part is the syntax analyzer. Writing a syntax analyzer for a TWS should be done in such a way that the analyzer can be used for a large class of grammars (e.g., a large subset of CF grammars), and it must work efficiently. It is with this goal in mind that this project was undertaken.

The basis for the method of parser construction presented in this thesis was developed by Knuth (10); and the first widely publicized, efficient implementation of the method was developed by DeRemer (3,4, 5). An analysis of both methods (table construction and parser construction) and certain optimizations on the table construction method have been developed by Aho and Ullman (1,2). The implementation presented here has similarities to all of the above plus some of the
author's own innovations.

In particular, DeRemer (4) has demonstrated that the technique is superior or equivalent in efficiency to other parsing methods such as operator precedence, simple precedence, bounded context, or McKeeman's mixed strategy precedence (MSP) (11) and also more general in its acceptance of languages.
CHAPTER II

CONTEXT-FREE GRAMMARS

Definitions

In general, a context-free grammar is a set of rules specifying a language. The language, L, is some subset of the set of all finite strings of symbols from an alphabet, A. That is, (possibly) not all strings of elements of L's alphabet are in L. The purpose of the grammar is to specify which strings can legitimately occur in L. Although the alphabet, A, is finite, the set of strings of A, denoted by A*, may be countably infinite. However, depending on the grammar, L may or may not be infinite. A second purpose of the grammar is to give a finite representation of L, even though L may be infinite.

To specify a grammar, there is a need for a set of symbols that is disjoint from the alphabet so that the grammar may be written in such a way that the rules of the grammar are not confused with strings in L. To accomplish this, a set of metasymbols, usually referred to as non-terminal symbols and characterized by the property that they do not appear in the alphabet, is used. The metasymbols represent the syntactic categories of the grammar.

The union of the alphabet and the metasymbols is referred to as the vocabulary, V, of the grammar; and the set of all strings of symbols from the vocabulary is denoted by V*.
Colons, commas, periods, and semicolons are punctuation symbols in the production rules defined below. They are not in the vocabulary. A comma means "is followed by"; a semicolon means "or" (exclusive); a colon means "may be rewritten as"; and a period is an end delimiter.

There are many variations in punctuation. Often the commas are replaced by blanks, the semicolons by vertical bars, the colons by either arrows or double colons followed by equals, and the periods by either blanks or semicolons.

Finally, the grammar is specified by a set of rules (also called rewriting rules or productions) of the form \( U_i : u_i \), where \( U_i \) is a metasymbol and \( u_i \in \mathcal{V}^* \). The set \( \{U\} \) has the property that exactly one element, say \( U_g \), appears only on the left of a colon and never on the right. The \( U_g \) is called the goal symbol (also distinguished symbol). This definition is overrestrictive but serves the purpose of this thesis. \( U_g \) is called the left-hand-side (LHS), and \( u_i \) is called the right-hand-side (RHS).

Formally, a grammar, \( G \), is defined as a quadruple \((\mathcal{V}_T, \mathcal{V}_N, P, S)\) where \( \mathcal{V}_T \) is the set of terminal symbols, \( \mathcal{V}_N \) is the set of non-terminal symbols, \( P \) is the set of productions, and \( S \) is the goal symbol.

As an example, the grammar, \( G_1 \), is specified by:

1. \( S : ?, E, ? \).
2. \( E : E, +, T \).
3. \( T \).
4. \( T : P, **, T \).
5. \( P \).
6. \( P : (, E, ) \).
7. \( i \).
Here, $V_T = \{?, +, **, (, )\}$, $V_N = \{S, E, T, P\}$. $S$ is the goal symbol, and $P$ is given.

The reader may ask how to represent one of the punctuation symbols in a production rule if it is actually in the alphabet; possible answers are to use some other symbol or to enclose the symbols of the alphabet within some other symbol not in the alphabet. By definition of the action of the semicolon, $E: E, +, T; T$ is equivalent to the two rules $E: E, +, T$ and $E: T$. The punctuation used (13) also allows the use of multi-character symbols.

Since a production means that the LHS can be rewritten as the RHS, applications of the production rules result in the following:

<table>
<thead>
<tr>
<th>PRESENT STRING</th>
<th>APPLIED RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $S$</td>
<td></td>
</tr>
<tr>
<td>(2) $?E?$</td>
<td>1</td>
</tr>
<tr>
<td>(3) $?E+T?$</td>
<td>2</td>
</tr>
<tr>
<td>(4) $?T+T?$</td>
<td>3</td>
</tr>
<tr>
<td>(5) $?P+T?$</td>
<td>5</td>
</tr>
<tr>
<td>(6) $?i+T?$</td>
<td>7</td>
</tr>
<tr>
<td>(7) $?i+P?$</td>
<td>5</td>
</tr>
<tr>
<td>(8) $?i+i?$</td>
<td>7</td>
</tr>
</tbody>
</table>

The final result, line #8, is a terminal string, that is, a string of terminal symbols. Each line is a direct derivative (6) of the previous line. Or, more formally, $X$ is a direct derivative of $W$ (written $W+X$) by application of the rule $U: u$ if there are (possibly empty) strings $x$ and $y$ such that $W = xuuy$ and $X = xuy$. The transitive closure of $+$, denoted by $+^*$, defines $X$ as a derivative of $W$ if there exist strings $W_0, W_1, \ldots, W_i$ such that $W = W_0 + W_1, W_1 + W_2, \ldots, W_{i-1} + W_i = X$. Line #8 is a derivative of line #2, for example. All derivatives of the goal symbol are called sentential forms. Sentences, the elements of the language, are
sentential forms consisting of terminal symbols only. More formally then, a language is defined as the set of sentences, that is, the strings of terminal symbols derivable from the goal symbol.

Since the grammar specifies the language, it now should be possible to tell what strings are valid in $L(G_1)$, the language generated by $G_1$. According to rule #1, legitimate strings are enclosed by question marks. Rules #2-3 describe an $E$ as a sequence of $T$'s separated by $+$'s. For example, $E+E + E + T + T+E + T + T + T + T$ specifies that an $E$ can be the sum of four $T$'s. Because $E$ appears in its own definition, the length of the string that can be produced is arbitrary. In this case, it is left recursion. ($E$ appears as the leftmost symbol of one of the RHS alternatives defining $E$.) If the rule were written $E: T, +, E$, then it would indicate right recursion. If there were a rule such as $E: T, E, T$, it would indicate embedded recursion. Rules #4-5 are similar in that they define a $T$ to be an arbitrarily long sequence of $P$'s separated by $**$'s. Finally, rules #6-7 define a $P$ to be either a parenthesized $E$ or an $i$. Recursion is a mechanism by which the finite grammar can describe an infinite language. For example, in $L(G_1)$, any arbitrarily long sequence of $i$'s separated by $+$'s is a legitimate sentence.

A conventional way to describe pictorially the derivation of $?i+i?$ presented earlier is given in Figure 1 and is called a syntax tree. Syntax trees are useful in that they reveal something about the structure of the grammar. For example, the question of precedence of operators and whether a particular operator is left associative or right associative is easily seen in a syntax tree of the string in
question. The string \( i_0 + i_1 + i_2 + i_3 + i_4 ** (i_5 + i_6) \) and its syntax tree are presented below in Figure 2. (The subscripts are only to facilitate correspondence of the string with the tree.

![Syntax Tree for \( S+*?i+i? \)](image1)

**Figure 1. Syntax Tree for \( S+*?i+i? \)**

![Syntax Tree for \( S+*?i_0 i_1 + i_2 + i_3 + i_4 ** (i_5 + i_6) \)](image2)

**Figure 2. Syntax Tree for \( S+*?i_0 i_1 + i_2 + i_3 + i_4 ** (i_5 + i_6) \)**
If the tree is traversed in postorder (9), it is clear that parenthesized expressions have precedence (i.e., they are encountered first in a postorder traversal) over **, which has precedence over +. Also, + is left associative while ** is right associative.

G₁ specifies FORTRAN-like arithmetic expressions. The associativity (grouping), right or left, is determined by the recursion, right or left. For some syntactic units, the grouping is unimportant; for example, a COMMENT is usually defined as any string of symbols of the alphabet with particular delimiters (e.g., /* */ in PL/1), and the grouping of the symbols is usually unimportant. However, the grouping is of utmost importance in syntactic units such as arithmetic expressions. Examination of G₁ and syntax trees for different sentences of L(G₁) reveals the 1 to 1 correspondence of left recursion with left associativity and right recursion with right associativity.

The reader may well ask, "Is the syntax tree for a particular string unique?" Or perhaps more importantly, "Are the members of a set of syntax trees for a given string equivalent?" This is all part of a larger question, namely, "Is the grammar ambiguous?" A grammar is said to be ambiguous if the language produced by the grammar is ambiguous. Formally, a grammar is unambiguous if there does not exist more than one canonical derivation sequence for any sentence in the language. A thorough discussion of grammar ambiguity is beyond the scope of this thesis; suffice it to say that, for the purpose of this thesis, if a given sentence has two or more different syntax trees, then the grammar is ambiguous. In particular, the method presented in this thesis fails if the grammar is ambiguous. However, if the method fails, it is not necessarily true that the grammar is ambigu-
Parsuing

Due to the complexity and depth of most modern high-level programming languages, there is a need to produce syntax analyzers mechanically to minimize costs of translator implementation, to maintain some degree of uniformity across different machines, and to facilitate changes and extensions to the language.

How is a string of L analyzed? What exists at this point is a set of rules for generating sentences of L(G). For a small finite language, one method is to generate all possible sentences and save them and then, to check any input string for validity, simply do a look-up. However, even for G₁, this method is not feasible if for no other reason than the recursion allows arbitrarily long sentences.

There are two general methods of analyzing (also called recognizing or parsing) elements of a language. The first, and possibly easiest to understand, is the top-down method. It is essentially a goal-oriented method; that is, predictions are made as to what the sentence is (hopefully the goal symbol), and then attempts are made to verify the prediction by determining if all of one of the RHS alternatives are present. Of course, to detect this presence leads to further predictions for any part of the alternative which is a non-terminal symbol. Essentially what is done is to "draw" the syntax tree from top to bottom (root to leaves). In parsing the sentence \( ?i+? \), the first prediction is that the sentence is an \( S \). But before it can be said that it is an \( S \), the RHS must be verified, that is, an \( E \) enclosed in question marks. The first question mark is
found in the string. Now an E must be found; that is, the presence of one of the RHS alternatives for E must be verified. If recognition of some alternative is attempted and failure results, then it is necessary to "backup" and try a different alternative; if all alternatives have been tried, then the string is not a sentence. Continuing with this example, a try is made to find an E; but, from the earlier discussion, an E is a sequence of T's separated by +'. Therefore, a T must be found; but a T is one or more P's separated by **'; therefore, a P must be found, and is found since the next input symbol is i, which completes a RHS alternative for P. Since there is no **, the longest T is found since P is a RHS alternative. The + is now detected and the next T in a manner similar to the first and, therefore, an E has been found and, with the closing question mark, an S; hence, the string is a sentence in L(G₁). Referring back to Figure 1, what has been done is to work down the tree, from left to right. Left recursion can cause problems in top-down parsing. For example, in the above discussion, left recursion was avoided by saying that an E was one or more T's separated by +'; however, that conclusion was only reached after some analysis of the grammar. If the problem had been attacked blindly, an E would have been predicted, then a move made to the alternative E, +, T and an E promptly predicted; and an endless loop would be entered.

The second commonly used parsing method is the bottom-up method. With bottom-up parsing, the syntax tree is not "drawn" but rather assumed to exist; and the method proceeds to verify this assumed tree. Again, working with G₁, the sentence ?i+i?, and Figure 1, a
phrase of the sentence is defined to be the set of end nodes of some subtree of the syntax tree. That is, a phrase is a derivation of some non-terminal symbol. The set of phrases of Figure 1 is \{i, i+i, ?i+i?\}. The handle is defined to be the leftmost phrase which contains no phrases other than itself. That is, the handle is the leftmost set of end nodes forming a complete branch, which is to say it is the direct derivation of the leftmost, bottom-most, non-terminal symbol node in the tree. Hence, in the example, i is the handle.

The following algorithm, given in (6), reflects the general philosophy of bottom-up parsing:

1. Let \( s = s_0 \) be a string to be analyzed. For \( i = 0, 1, \ldots, n \) until \( s_n = S \) has been produced, do the following steps.
2. Find the handle of \( s_i \).
3. Replace the handle of \( s_i \) by the name of its father in the syntax tree.
4. Prune the handle from the tree.

The sequence \( s_n + s_{n-1} + \ldots + s_0 \) is now a derivation of \( s_0 \). The following demonstrates the algorithm applied on \( s = s_0 = ?i+i? \).

<table>
<thead>
<tr>
<th>PRESENT STRING</th>
<th>HANDLE</th>
<th>STRING AFTER STEP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ?i+i?</td>
<td>i</td>
<td>?P+i?</td>
</tr>
<tr>
<td>(2) ?P+i?</td>
<td>P</td>
<td>?T+i?</td>
</tr>
<tr>
<td>(3) ?T+i?</td>
<td>T</td>
<td>?E+i?</td>
</tr>
<tr>
<td>(4) ?E+i?</td>
<td>i</td>
<td>?E+P?</td>
</tr>
<tr>
<td>(8) S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The steps in the "present string" column are followed back-
wards, the derivation \( S \rightarrow^* \gamma \) results. In fact, a rightmost derivation sequence exists in that each step is of the form \( PAB \rightarrow PcB \) where \( B \) is a terminal string, \( c \) is a terminal symbol, and \( PcV^* \); that is, a production whose LHS is the rightmost non-terminal symbol of the sentential form is used. In this paper, the rightmost derivation is used as the canonical derivation. A canonical parse is the reverse of a canonical derivation.

All parsing methods have both good and bad characteristics. Some are easy to implement but inefficient while others are complex but efficient. Perhaps it is the lack of a "best" method that has led to the variety of methods (6). In general, there are two problems with which all syntax analyzers must deal.

First, the problem of backtracking must be dealt with. In both bottom-up and top-down parsing, a choice must be made as to which alternative of a production should be used in the next step of the parse. Input symbols are then picked up to try to fulfill that alternative. If the parsing scheme picks the wrong alternative, then it must back up and try another. One way of alleviating this problem, at least somewhat, is with look-ahead. That is, the parser scans ahead in the input string to gain a clue as to which alternative to attempt to match. Some of the questions raised by look-ahead are whether only to look ahead or to look back at what has been processed or both and how far to look. As a preview, the method presented later has implicit unrestricted look-back and one symbol look-ahead.

The second problem area for syntax analyzers is error recovery. That is, if and when an error is detected, what course of action
should the analyzer take. "ERROR IN ABOVE PROGRAM" is not a very informative diagnostic message. On the other extreme, an analyzer which could correct every error would have the intelligence to write programs itself. Error recovery and error correction are not treated to any degree of sophistication in this thesis.

One of the principal characteristics about a large class of context-free languages for which parsing methods in this thesis apply is that the syntax analyzers for them can be formalized as deterministic push down automata (DPDA) (6). By push down, it is meant that, if the DPDA were modelled by a computer program, then that program would use a stack. That is, a history of the previously travelled path is recorded (remembered). The nature of this DPDA, which consists of a finite number of states, a push down mechanism, and state transitions, is to input the symbols of a string and to make state transitions according to what symbol is read and the present state. In effect, a DPDA "remembers" the previous symbols (at least the ones it needs) by the path of state transitions to reach the present state. The goal is to reach a unique state, the final state, at the same time the input string is depleted. A language is deterministic if every sentence of the language is accepted by a DPDA. That is, every sentence causes the DPDA to reach the final state at the same time the input string becomes depleted.

Knuth's original work (the LR(k) method) is equivalent to a DPDA in its acceptance of languages. The author's implementation is somewhat less general in that a restricted form of Knuth's method is used, resulting in a parser which accepts a large subset
of the languages acceptable to a DPDA.

Relations and Closures of Relations

In the previous discussion of look-ahead and look-back, it was implied that they were methods for deciding which RHS alternative to use in the next step of a parse. This is equivalent to saying that the handle can be uniquely determined. Usually, when there is look-ahead, what action to take is determined not only by what the scanned input symbol is but also by how much of a handle has been recognized. In particular, the rightmost symbol (top of the stack) of the partially recognized handle is of interest. That is, the relation between the two symbols determines the action. The need for knowing particular relations between symbols of a grammar has led to a number of important properties and algorithms.

To begin with, it is necessary to review the definition and properties of a binary relation and describe the notation. For sets A and B, the Cartesian product of A and B is defined to be $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$. A binary relation, R, defined on $A \times B$, is defined to be a subset of $A \times B$ such that the relation holds between the first and second elements of the ordered pairs. The possibilities $A = B, A \subseteq B, B \subseteq A, A \cap B \neq \emptyset$ or $A \cap B = \emptyset$ exist. There are four notations used in this paper to describe R defined on $A \times B$.

Notation #1

$$R = \{(a, b) \mid a \in A, b \in B, \text{ and } a R b\}$$

Notation #2

$$R(a) = \{b \mid a \in A, b \in B, \text{ and } a R b\}$$
Notation #3

The relation can be defined by a matrix whose entries are either 0 (false) or 1 (true), that is, a Boolean matrix. Correspond the rows with elements of A and the columns with elements of B. If \( a \sim R b \), and \( a \) corresponds to row \( i \), and \( b \) corresponds to column \( j \), then the \( ij \)th entry is 1. If \( a \not\sim R b \), then the \( ij \)th entry is 0.

Notation #4

The relation can be defined by a directed graph such that nodes \( a \) and \( b \) are connected by an arc if and only if \( a \sim R b \). That is, for \( a \in A \), \( b \in B \), and \( a \sim R b \), there exists an arc from node \( a \) to node \( b \).

The properties of a relation, \( R \), defined on \( A \times B \), can be stated symbolically as:

**Reflexive.** \( a \sim R a \) for every \( a \in A \) and every \( a \in B \)

**Symmetry.** \( a \sim R b \) if and only if \( b \sim R a \)

**Transitivity.** \( a \sim R b \) and \( b \sim R c \) if and only if \( a \sim R c \)

for \( a \in A \), \( b \in A \cap B \), \( c \in B \)

If all three properties exist for \( R \), then \( R \) is said to be an equivalence relation; for example, the relation of equality of positive integers (here \( A = B \)) is an equivalence relation.

In the following, \( i \), \( j \), and \( k \) are positive integers:

**Reflexive.** \( i = i \)

**Symmetry.** \( i = j \) if and only if \( j = i \)

**Transitivity.** \( i = j \) and \( j = k \) if and only if \( i = k \)

The relation, \( H \), defined on \( V \) of \( G_1 \) by \( H = \{ (a,b) \mid A \in V_{N}, b \in V, C \in V^*, \text{and } A: b, C \in P \} \), exists between all LHS's and the first (head) symbol of their RHS alternatives. The pairs of \( G_1 \) for which \( H \) holds are \{ (S,?) \}, \{ (E,E) \}, \{ (E,T) \}, \{ (T,P) \}, \{ (P,(),(),i) \}. It is more con-
venient to represent the relation with a Boolean matrix whose rows and columns correspond to \( V \). For \( H(G_1) \), Figure 3 applies. Also, for reasons of visual clarity, it is convenient to represent a relation as a directed graph where nodes related to each other are connected. For \( H(G_1) \), Figure 4 applies. In terms of the directed graph, the Boolean matrix is the adjacency matrix. In Figure 4, an \( E \) eventually leads to a \( ( \). Some way to represent this in a single step rather than three is desirable. That is to say, a relation like \( H \), but which is transitive, is desired so that all possible head symbols of strings that are derivatives of a given non-terminal symbol can be discerned. If \( H \) were transitive (which it is not), then an application of the transitivity would give \( E H T \rightarrow T H P \rightarrow E H P \), and \( E H P \rightarrow P H \). But \( (E,P) \) and \( (E,() \) are not in \( H \) since \( P \) is not the first symbol of a RHS alternative of a production for which \( E \) is the LHS and likewise for \( ( \). Therefore, it is necessary to define a new relation, \( H^+ \), the transitive closure of \( H \). However before defining \( H^+ \), the properties of the transitive closure of a relation need to be developed.

\[
\begin{array}{ccccccccc}
S & E & T & P & ? & + & ** & ( & ) & 1 \\
\hline
S & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
E & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
T & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
P & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

Figure 3. Boolean Matrix Representation of \( H(G_1) \)
The product of two relations, say $R$ on $A \times B$ and $P$ on $C \times D$, is defined by a relation $RP$ if and only if there exists an $e \in B \cap C$ such that $c R e \land e P d$ is true. If $P$ is a product relation, say $QT$, such that $e QT d$ so that there does exist an $f$ such that $e Q f \land f T d$ is true, then, for the relation $RP$, which is actually $PQT$, it is true that $c R e \land e Q f \land f T d$. But $\land$ is associative and hence $R(QT) = (RQ)T$. A theorem (7) that will be used extensively hereafter states that the Boolean matrix representation of a product relation can be computed by the product of the Boolean matrices for
the original relation. Using the definition of product, the powers of a relation, $R$, are defined by $R^n = R \cdot R^{n-1}$ where $n > 0$ and $R^1 = R$ and the **transitive closure** of $R$ by $a \xrightarrow{R^+} b$ if and only if there exists $a \xrightarrow{R^*} c$ such that $a \xrightarrow{R^n} c$ for some $n > 0$. If the **identity relation** is denoted by $R^0$, that is, $a \xrightarrow{R^0} b$ if and only if $a = b$, then the **reflexive transitive closure**, $R^*$, can be defined as $a \xrightarrow{R^*} b$ if and only if $a \xrightarrow{R^n} b$ for $n \geq 0$. For the transitive closure, if each power of $R$ is considered as a separate relation, then $R^+ = (R^1 \cup R^2 \cup R^3 \cup \ldots \cup R^n)$ where $n$ is the number of elements in the set on which the relation is defined. This is proven by Gries in (7). It should be clear without proof that $R^+$ is itself a transitive relation. The transitive closure of $H(G_1)$ is defined by $H^+(A) = \{ b \in V | A \xrightarrow{R^*} b \}$ where $C \in V^*$. $H^+(G_1)$ can be represented by the Boolean matrix in Figure 5.

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>T</th>
<th>P</th>
<th>?</th>
<th>+</th>
<th>**</th>
<th>( )</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>?</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>**</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.** Boolean Matrix Representation of $H^+(G_1)$
Translating Figure 5 into a graph, Figure 6 results:

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

Figure 5. (Continued)

Translating Figure 5 into a graph, Figure 6 results:

\[ S \rightarrow ? \]
\[ \begin{array}{ccc}
E & T & P \\
\end{array} \]

Figure 6. Graph Representation of \( H^+(G_1) \)

\( H^+(G_1) \), the reflexive transitive closure of \( H \), would differ from \( H^+(G_1) \) by having an arc from each node into itself.

There are two subtle but very important ideas that are used here and need to be brought to the surface. The first is that, when forming the Boolean matrix \( H^+ \), a twist on matrix algebra is used. To actually perform RR, the rules of matrix multiplication are used, with "and" replacing "times" and "or" replacing "plus." This correspondence is clear when the Boolean matrix is represented with 1 for "true" and 0 for "false." That is, for ordinary matrix multiplication
\((AB = C)\), the \(ij^{th}\) element of \(C\) is defined by

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}; \]

but, for Boolean matrix multiplication, the \(ij^{th}\) element of \(C\) is defined by

\[ c_{ij} = (a_{i1} \land b_{1j}) \lor (a_{i2} \land b_{2j}) \lor ... \lor (a_{in} \land b_{nj}) \]

where \(A\) and \(B\) are square Boolean matrices of rank \(n\). Rewriting the definition of \(R^+\) as \(R^+ = R^n + R^{n-1} + \ldots + R^1\), it is seen that the computation of \(R^+\) has similarities of evaluating a matrix polynomial with all coefficients equal to the identity matrix. Clearly, in a mechanical computation, some efficient method for the calculation of \(R^+\) is needed, perhaps a method similar to the nested multiplication method of evaluating polynomials. Such a method does exist and is known as the Warshall algorithm. The second point is how to relate the powers of a relation to the grammar. \(H^+(G_1)\) is used as an example. Clearly, \(H^1(G_1)\) is the application of one production, that is, \(H(G_1)\). But \(H^1 \lor H^2\) is the application of one or two productions. For the graph of Figure 4, this in effect is connecting the paths of length 2, for example, the arc \(T \rightarrow i\). Likewise, for higher powers, \(H^1 \lor H^2 \ldots \lor H^i\) in effect connects arcs of length 1, 2, \(\ldots, i\). Of course, this is with respect to the original graph. With respect to the present updated graph at each step, paths of length 2 are always connected.

Warshall (14) developed an algorithm for computation of the closure of an \(n \times n\) Boolean matrix that is superior to other methods (e.g., nested multiplication). For example, Warshall claims that,
while the computation of closure matrices for other methods goes up with \( n^3 \), his method goes up slightly faster than \( n^2 \).

Normally, the Warshall algorithm calls for \( n \) iterations; however, from a practical point of view, the user can, under certain restrictions, reduce the number of iterations in the original algorithm and still produce the desired closure matrix. For \( G_1 \), there are 10 rows in the Boolean matrix representation of \( H(G_1) \). If the original algorithm were used, 10 iterations would be made, one for each row. However, there are only seven production rules so that at most seven iterations are needed. There is only one node for each non-terminal symbol; hence, the longest possible path has length equal to the number of non-terminal symbols. But it is also true that three of the production rules of \( G_1 \) have the same LHS, and only one of the rules with a common LHS can apply at any step. Hence, only four iterations are needed. The point is that usually a restriction (resulting in greater efficiency) can be imposed on the Warshall algorithm, depending on the relation being closed.

As stated earlier, for \( G_1 \), four iterations are needed and the Warshall algorithm makes one iteration for each row of the Boolean matrix. Since the Boolean matrices of concern represent a relation (i.e., a set of ordered pairs), the rows may be swapped in any manner provided similar swaps are made with the columns. Again recalling that the relation \( H \) is defined on \( V_N \times V \), it should be clear that it is desirable and correct to arrange the Boolean matrix representation of \( H \) so that the non-terminal symbols occupy contiguous rows and that the Warshall algorithm need only iterate on those rows. (Figure 3 is arranged this way.) If closure of \( H(G_1) \) is thought of in terms of
Boolean matrix multiplication, the reader will see that, at every step (i.e., every power of H), the rows labelled with terminal symbols remain all zeroes. So it must also be with iterations of the Warshall algorithm.

A symbolic statement of the algorithm may be found in (14); however, the major goal of this thesis is to present concepts and methods that are actually used in an implementation and, therefore, a PL/1 program segment is used to describe the working algorithm.

Let M be a bit matrix representing a relation defined on A x B whose rows correspond to the elements of A and whose columns correspond to elements of B. It is necessary that A ⊆ B and that, if row i corresponds to x ∈ B. (An example of such an M is the first four rows and all columns of Figure 3.) The PL/1 program segment follows.

```pl1
DO K=LBOUND(M,1) TO HBOUND(M,1); /* FOR ALL ROWS */
DO I=LBOUND(M,1) TO HBOUND(M,1); /* FOR ALL ROWS */
  IF M(I,K) THEN /* IF A K TH COLUMN ENTRY IS TRUE */
    DO J=LBOUND(M,2) TO HBOUND(M,2); /* FOR ALL COLUMNS */
      IF M(K,J) THEN M(I,J)=1'B;
      END;
  END;
END;
END;
```

Practical Restrictions on CF Grammars

Gries (7) discusses some practical restrictions on CF grammars so that mechanically generated parsers can be applied more efficiently to the languages generated by CF grammars. Some methods require more restrictions than others. The LR(k) method, to be presented later, requires fewer restrictions than any other known method for which efficient parsers can be mechanically produced (3).
Restriction #1

A production of the form $A : A$ clearly makes a grammar ambiguous, serves no useful purpose, and can easily be detected either mechanically or by visual inspection. In this thesis, it is assumed no such production is present.

Restriction #2

Every non-terminal symbol must appear in some sentential form, that is, $S \rightarrow^*xAy$ for every $A \in V_N$ and $x, y \in V^*$. This condition can be mechanically detected by constructing the relation WITHIN, denoted by $W$, and defined by $W(A) = \{ B \mid B$ is a non-terminal symbol that appears in a production whose LHS is $A \}$, then computing $W^+$. For any "0" in the goal symbol row, except the goal symbol column, the symbol represented by that column is not "within" the goal symbol and therefore violates the restriction.

Restriction #3

Every non-terminal symbol must be able to derive a terminal string. Gries (7) presents an algorithm for detecting this condition, which basically consists of "marking" any production whose RHS is composed of only terminal symbols or "marked" non-terminal symbols. Several passes over the productions are usually needed; and the algorithm stops when, during a previous pass, no LHS was "marked." When the algorithm stops, any unmarked production cannot derive a terminal string and therefore contributes nothing to the language specified by the grammar.

Restriction #4

No production is of the form $A : \epsilon$, that is, no RHS is empty. Again this restriction is easily detected by visual inspection. In
this thesis, it is assumed no such production is present.

**Restriction #5**

No duplicate RHS's are present in the grammar. Duplicate RHS's cause most bottom-up methods to fail but do not necessarily affect the method presented in this thesis. However, as a general rule of thumb, grammars with duplicate RHS tend to cause the table construction method to fail to produce a complete table.

In the author's implementation, Restrictions #1 and #4 must be detected visually, but #2, #3, and #5 are mechanically detected. However, only warnings are issued since, if these restrictions are violated, they do not necessarily cause the method presented in this thesis to fail but do make it less efficient.

In this chapter, elementary topics have been investigated. For a theoretical basis for these concepts, the reader is referred to (8) and, for an application-oriented reference, to (7).
CHAPTER III

LEFT TO RIGHT TRANSLATION OF LANGUAGES

The LR(k) Method

The reader may well ask which is better, top-down or bottom-up parsing. There are advantages in both. What is sought is a completely language-independent (assuming a CF grammar) recognizer that is efficient and combines the most desirable aspects of both top-down and bottom-up methods. This is precisely what is embodied in Knuth's (10) LR(k) method, which can be described generally as a parsing method that scans sentences from left to right, using no more than k symbol look-ahead to determine whether to input the next symbol or make a reduction. LR(k) grammars (grammars that produce languages which can be parsed with LR(k) methods) are the largest known class of CF grammars for which deterministic (i.e., no backtracking), left-to-right, bottom-up parsers can be mechanically generated. In fact, this class of grammars is capable of describing virtually all of the commonly used programming languages (3). Another way of describing a deterministic language is to say that the handle can always be uniquely determined. That is, the parser never picks the "wrong" RHS alternative.

The LR(k) method, given a CF grammar, produces a table which is used by a language-independent parsing algorithm to parse sentences of the language generated by the grammar. In general, Knuth's original
LR(k) method produces tables too large for practical use. A closely related method known as SLR(k) (3) (simple LR(k)), which results in more practical parsers, is the method of principal concern here. However, for reasons of completeness, the LR(k) method is treated briefly.

If a is a right sentential form, that is, a is a rightmost derivation of the goal symbol, then $\text{FIRST}_k^k (a)$ is defined to be the first k terminal symbols derivable from a. That is, $\text{FIRST}_k^k (a) = \{ w \in V_T^* \mid a \Rightarrow^* wx, x \in V_T^* \text{ and either } w \text{ is } k \text{ symbols long or } w \text{ is less than } k \text{ symbols long and } x = \emptyset \}$. If $a \in V_T^*$, then $\text{FIRST}_k^k (a)$ is the first k symbols of a. Every right sentential form contains a handle. An informal definition of an LR(k) grammar, given in (1), is that a grammar is LR(k) if the handle, h, of a right sentential form, bha, is unique and the production that derived the handle is uniquely determined by examining bh and $\text{FIRST}_k^k (a)$.

Development of an algorithm which does this examining for all right sentential forms follows. In actual practice, this consists of constructing the aforementioned table, which tells the parsing algorithm whether to stack the incoming symbol or make a reduction. A reduction consists of popping a RHS from the stack and replacing it with the corresponding LHS. This parsing action is the reason for stating earlier that the LR(k) method of parsing corresponds to a DPDA. The row of the table that is used in the decision corresponds to a DPDA state, the "push down" to the stack; and the method is deterministic as described above. An LR(k) table is actually two tables in one (1). The table is considered to be a pair of functions $(p, g)$ such that:

(1) $p$, the parsing action function, maps the look-ahead strings (length k or less) into stack, error, or
reduce \textit{i}, where \textit{i} is a production number.

(2) \textit{g}, the goto function, maps \textit{V} to the states (rows of the table).

The process ends when the \textbf{final state} (a particular row of the table) is entered. The problem of entering the final state with unex-pended suffix does not exist since special delimiters are placed before and after the text to be processed. Also, there is a \textbf{start state} in which to start the processing. The parsing algorithm is the same for both the LR(k) and the SLR(k) methods. Actually, the tables are quite similar for both methods also, but it is in the construction of the table where the methods differ.

For an LR(1) grammar, that is, \( k = 1 \), only one symbol look-ahead is allowed. It has been proven (10) that any LR(k) grammar can be rewritten in an equivalent form as an LR(1) grammar. Here, FIRST \((A)^c \subseteq \textit{H}^+(A)\), that is, it contains the terminal symbol elements.

The LR(1) table is constructed by first constructing the configuration sets. There is a 1 to 1 correspondence between these configuration sets and rows of the table. Each configuration set is composed of items; each item is of the form \((A^+a,b,u)\) where \(A^+ab\) is a production (represents a direct derivation); the "." marks the dividing point in a partially recognized handle; and \(u\) is a valid next input symbol if the item is recognized. There are two important actions used to construct the configuration sets.

\textbf{CLOSEURE} - A set begins with items specified by expansion. The first set begins with \((S^+,?E?,\emptyset)\). If \((A^+a,b,c,u)\) is in the set, then \((B^+,d,v)\) is added to the set for productions \(B: d\) for any \(d \in \textit{V}^*\) and \(v \in \text{FIRST}(cu)\). Here, \(a,c \in \textit{V}^*\) and \(B \in \textit{V}_N\). What is being done is
to find an item with the dot to the left of a non-terminal, then to enter all productions for which that non-terminal is a LHS. FIRST (cu) indicates what terminal symbol can follow the non-terminal symbol in the sentential form. Duplicate entries are never made. If FIRST (cu) has two elements, say \( v_1 \) and \( v_2 \), then two set entries are required; however, the SLR(k) method only has one set entry since FIRST is not considered when forming the configuration sets. This is the essential difference in the LR(k) and SLR(k) methods of construction.

**EXPANSION** - Once a set is closed, it may be used to form a new set. That is, the algorithm finds all items in \( A \) with an \( X \) to the right of the dot \( (X \in V) \). Then the new set, \( A' \), is initialized to these items with the dot moved to the right of the \( X \) such that \( A' \) is a set of items \((B+aX.b,u)\) and \((B+aXb,u)\) is in the set \( A \). Each item can be used only once for expansion. If the sets are numbered from 1 to \( n \), then, if \( A = A_i \) and \( A' = A_j \), the entry at row \( i \), column \( X \) (i.e., the column corresponding to \( X \)), is set to \( j \). If \( A' = A'' \), then \( A' \) is not added to the set of configuration sets; but the table is set as if it were unique.

G\(_2\) is specified by:

1. \( S: E \).
2. \( E: A, A \).
3. \( A: a, A; \)
4. \( b. \)

\((B+a.b,c_1), \ldots, (B+a.b,c_m)\) is denoted by \((B+a.b,c_1/c_2/\ldots/c_m)\).

The results of computation of the configuration sets for \( G_3 \) are shown in Table I.
TABLE I

CONFIGURATION SETS - LR(1) METHOD ON $G_2$

<table>
<thead>
<tr>
<th>SET</th>
<th>NAME</th>
<th>NO.</th>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td></td>
<td>1.</td>
<td>S+.E,∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.</td>
<td>E+.AA,∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.</td>
<td>A+.aA,a/b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.</td>
<td>A+.b,a/b</td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td>1.</td>
<td>S+E.,∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.</td>
<td>E+A.A,∅</td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td>1.</td>
<td>A+.aA,∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.</td>
<td>A+.aA,a/b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.</td>
<td>A+.b,∅</td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td>1.</td>
<td>A+a.A,a/b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.</td>
<td>A+.aA,a/b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.</td>
<td>A+.b,a/b</td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td>1.</td>
<td>A+b.,a/b</td>
</tr>
<tr>
<td>A5</td>
<td></td>
<td>1.</td>
<td>E+AA.,∅</td>
</tr>
<tr>
<td>A6</td>
<td></td>
<td>1.</td>
<td>A+a.A,∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.</td>
<td>A+.aA,∅</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.</td>
<td>A+.b,∅</td>
</tr>
<tr>
<td>A7</td>
<td></td>
<td>1.</td>
<td>A+b.,∅</td>
</tr>
<tr>
<td>A8</td>
<td></td>
<td>1.</td>
<td>A+aA.,a/b</td>
</tr>
<tr>
<td>A9</td>
<td></td>
<td>1.</td>
<td>A+aA.,∅</td>
</tr>
</tbody>
</table>

$G_3$ is specified by:

1. S: ?, E, ?.
2. E: a, A, b;
3. a, B, c;
4. d, A, c;
5. d, B, b.
6. A: f, A;
7. f.
8. B: f, B;
9. f.
The results of computation of the configuration sets for $G_3$ are shown in Table II.

### TABLE II

**LR(1) Configuration Sets for $G_3$**

<table>
<thead>
<tr>
<th>SET</th>
<th>NAME</th>
<th>NO.</th>
<th>ITEMS</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>S+.?E?,Ø</td>
<td>1.</td>
<td>S+.?E?,Ø</td>
<td>from A0.1</td>
</tr>
<tr>
<td>A1</td>
<td>E+.aAb,?</td>
<td>2.</td>
<td>E+.aAb,?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E+.aBc,?</td>
<td>3.</td>
<td>E+.aBc,?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E+.dAc,?</td>
<td>4.</td>
<td>E+.dAc,?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E+.dBb,?</td>
<td>5.</td>
<td>E+.dBb,?</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>S+.?E?,Ø</td>
<td>1.</td>
<td>S+.?E?,Ø</td>
<td>from A1.1</td>
</tr>
<tr>
<td>A3</td>
<td>E+a,Ab,?</td>
<td>1.</td>
<td>E+a,Ab,?</td>
<td>from A1.2</td>
</tr>
<tr>
<td></td>
<td>E+a,Bc,?</td>
<td>2.</td>
<td>E+a,Bc,?</td>
<td>from A1.3</td>
</tr>
<tr>
<td></td>
<td>A+.fA,b</td>
<td>3.</td>
<td>A+.fA,b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+.f,b</td>
<td>4.</td>
<td>A+.f,b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B+.fB,c</td>
<td>5.</td>
<td>B+.fB,c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B+.f,c</td>
<td>6.</td>
<td>B+.f,c</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>E+d.Ac,?</td>
<td>1.</td>
<td>E+d.Ac,?</td>
<td>from A1.4</td>
</tr>
<tr>
<td></td>
<td>E+d.Bb,?</td>
<td>2.</td>
<td>E+d.Bb,?</td>
<td>from A1.5</td>
</tr>
<tr>
<td></td>
<td>A+.fA,c</td>
<td>3.</td>
<td>A+.fA,c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+.f,c</td>
<td>4.</td>
<td>A+.f,c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B+.fB,b</td>
<td>5.</td>
<td>B+.fB,b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B+.f,b</td>
<td>6.</td>
<td>B+.f,b</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>S+.?E?,Ø</td>
<td>1.</td>
<td>S+.?E?,Ø</td>
<td>&quot;final&quot; set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>from A2.1</td>
</tr>
<tr>
<td>A6</td>
<td>E+aA.b,?</td>
<td>1.</td>
<td>E+aA.b,?</td>
<td>from A2.1</td>
</tr>
<tr>
<td>A7</td>
<td>E+aB.c,?</td>
<td>1.</td>
<td>E+aB.c,?</td>
<td>from A2.2</td>
</tr>
<tr>
<td>A8</td>
<td>A+.fA,b</td>
<td>1.</td>
<td>A+.fA,b</td>
<td>from A3.1</td>
</tr>
<tr>
<td></td>
<td>A+.f,.b</td>
<td>2.</td>
<td>A+.f,.b</td>
<td>from A3.2</td>
</tr>
<tr>
<td></td>
<td>B+.fB,c</td>
<td>3.</td>
<td>B+.fB,c</td>
<td>from A3.3</td>
</tr>
<tr>
<td></td>
<td>B+.f,c</td>
<td>4.</td>
<td>B+.f,c</td>
<td>from A3.4</td>
</tr>
<tr>
<td></td>
<td>A+.fA,b</td>
<td>5.</td>
<td>A+.fA,b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A+.f,b</td>
<td>6.</td>
<td>A+.f,b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B+.fB,c</td>
<td>7.</td>
<td>B+.fB,c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B+.f,c</td>
<td>8.</td>
<td>B+.f,c</td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td>E+dA.c,?</td>
<td>1.</td>
<td>E+dA.c,?</td>
<td>from A4.1</td>
</tr>
<tr>
<td>A10</td>
<td>E+dB.b,?</td>
<td>1.</td>
<td>E+dB.b,?</td>
<td>from A4.2</td>
</tr>
<tr>
<td>A11</td>
<td>A+.fA,c</td>
<td>1.</td>
<td>A+.fA,c</td>
<td>&quot;All is not a duplicate of A8&quot;</td>
</tr>
</tbody>
</table>
### TABLE II (Continued)

<table>
<thead>
<tr>
<th>SET</th>
<th>NAME</th>
<th>NO.</th>
<th>ITEMS</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>A+f.,c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>B+f.B,b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>B+f.,b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>A+.fA,c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>A+.f,c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>B+.fB,b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>B+.f,b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A12</td>
<td>1.</td>
<td></td>
<td>E+aAb.,?</td>
<td>from A6.1</td>
</tr>
<tr>
<td>A13</td>
<td>1.</td>
<td></td>
<td>E+aBc.,?</td>
<td>from A7.1</td>
</tr>
<tr>
<td>A14</td>
<td>1.</td>
<td></td>
<td>A+fA.,b</td>
<td>from A8.1</td>
</tr>
<tr>
<td>A15</td>
<td>1.</td>
<td></td>
<td>B+fB.,c</td>
<td>from A9.1</td>
</tr>
<tr>
<td>A16</td>
<td>1.</td>
<td></td>
<td>E+dAc.,?</td>
<td>from A9.1</td>
</tr>
<tr>
<td>A17</td>
<td>1.</td>
<td></td>
<td>E+dBb.,?</td>
<td>from A10.1</td>
</tr>
<tr>
<td>A18</td>
<td>1.</td>
<td></td>
<td>A+fA.,c</td>
<td>from A11.1</td>
</tr>
<tr>
<td>A19</td>
<td>1.</td>
<td></td>
<td>B+fB.,b</td>
<td>from A11.3</td>
</tr>
</tbody>
</table>

The reader who is interested in understanding the structure of a grammar using LR(k) techniques should pay particular attention to computation of the configuration sets. For any given item, the dot delimits how much of a handle has been formed. Closure shows what the next input symbol can be. Although the same item may appear in more than one set, the history of how that set was entered is contained in the entries created by expansion.

Table III contains the LR(1) table for G3. The table is computed from the configuration sets by the following algorithm (2):

1. If (B+b.,u) is in A and B is not the goal symbol, then $p(u) = i$ where $i$ is the number of the production $B: b$.

2. If (B+a.b,u) is in A and $b \neq \emptyset$, then $p(v) =$ (for stack)
for all $v \in \text{FIRST}(bu)$, that is, for all terminal symbols that can legitimately follow $a$ in this state.

(3) If $(S \rightarrow \varepsilon, \varnothing)$ is in $A$, then $p(\varnothing) = \text{accept}$.

(4) $p(u) = \text{error}$ (blank entry) otherwise.

(5) $g(X)$ entries are as mentioned earlier.

(6) If more than one entry is attempted for any table position, then the grammar is not LR(k) for the $k$ used in constructing the configuration sets.

The parsing algorithm is quite simple once the table is generated. Also, the parsing algorithm is general in that it applies to a restricted form of the LR(k) method, the SLR(1) method. The table entry is selected by letting STACKTOP (i.e., the top of the stack) select the row and the next input symbol select the column. When the table entry is "stack," the next input symbol is stacked along with the table entry which is a state name. When the table entry is reduce (i.e., a production number), $N$ symbols are popped from the stack where $N$ is two times the length of the RHS of the production used in the reduction, and the LHS of the production is pushed onto the stack along with the table entry selected by the STACKTOP row and LHS column. This table entry is always a state name. (This creates the effect of pushing the LHS into the unexpended suffix and then reading it.)

The symbols in the stack concatenated with the unexpended suffix at any step yield a right sentential form. Working from bottom to top, this results in $S \rightarrow \varepsilon ? E ? + ? a B c ? + ? a B c ? + ? a f f c ?$, which is indeed the right-most derivation sequence for $? a f f c ?$. 
TABLE III

LR(1) TABLE FOR G₃

<table>
<thead>
<tr>
<th>STATE</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>Ø</th>
<th>S</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>?</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>S</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>S</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STACK</th>
<th>SUFFIX</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?affc?</td>
<td>initial condition, read ?</td>
</tr>
<tr>
<td>0?1</td>
<td>affc?</td>
<td>read a</td>
</tr>
<tr>
<td>0?1a3</td>
<td>ffc?</td>
<td>read f</td>
</tr>
<tr>
<td>0?1a3f8</td>
<td>fc?</td>
<td>read f</td>
</tr>
<tr>
<td>0?1a3f8f8</td>
<td>c?</td>
<td>reduce B: f.</td>
</tr>
<tr>
<td>0?1a3f8B15</td>
<td>c?</td>
<td>reduce B: f, B.</td>
</tr>
<tr>
<td>0?1a3B7</td>
<td>c?</td>
<td>read c</td>
</tr>
<tr>
<td>0?1a3B7c13</td>
<td>?</td>
<td>reduce E: a, B, c.</td>
</tr>
<tr>
<td>0?1E2</td>
<td>?</td>
<td>read ?</td>
</tr>
<tr>
<td>0?1E2?5</td>
<td>Ø</td>
<td>accept</td>
</tr>
</tbody>
</table>

Figure 7. Parsing ?affc? Using Table III
The SLR(1) Method

Knuth's original article (10) introducing LR(k) grammars is considered a classic because of its theoretical soundness and generality. However, attempts at practical implementation have suggested changes that result in somewhat less generality but substantially greater practicality.

DeRemer proposed (3) and implemented (5) an LR(k)-like method which he called SLR(k) for simple-LR(k). Basically, it consists of constructing LR(k) configuration sets for k = 0; that is, the method assumes (at least at configuration set construction time) that the grammar is LR(0). Whereas Knuth's original method uses k symbol look-ahead while constructing the configuration sets, DeRemer doesn't make use of k symbol look-ahead until table construction time and then only if necessary.

The SLR(1) method is stated initially in terms of the LR(1) method. The FOLLOW function, F, is defined by $F(A) = \{a \mid S \rightarrow^* bAc \text{ and } a \in \text{FIRST}(c) \text{ where } A \in V_N, a \in V_T, b \in V^*, \text{ and } c \in V_T^*\}$. That is, $F(A)$ is the set of terminal symbols which may follow $A$ in any right sentential form. The following algorithm constructs the SLR(1) table (2):

1. Construct the LR(0) configuration sets of items.
2. Replace each item of the form $(A \rightarrow b, \emptyset)$, $b \in V^*$, in each set by $(A \rightarrow b, a)$ for all $a \in F(A)$.
3. Construct the LR(1) tables from the altered sets of items with the function $g$ determined as though dealing with LR(0) sets of items.
It is possible to have a conflict, that is, more than one entry for a table position for the SLR(1) method when one does not exist for the LR(1) method, which occurs when an attempt to perform the SLR(1) method on $G_3$ is made.

The author has implemented changes in the SLR(1) method which make the implementation more efficient. First, the stack and accept entries are deleted, and the numbers are negated in the $p$ portion of the LR(1) table. Secondly, the modified $p$ portion is "overlaid" with the $g$ portion. Here, positive entries must be considered as not only transitions to a different state (row) but also as signals for stacking; and the row corresponding to the final state must be identified so that a transition to it can be detected. But these are minor points. Also, if it is always agreed to surround the single RHS alternative of the goal symbol with special delimiters, the $\emptyset$ column is completely eliminated since the only possible entries are reduction entries and accept; however, there are no reduction entries in the $\emptyset$ column except for the number of the production $S: ?, E, ?.$, but this is detected by detecting a transition to the final state. Also, the final state row and goal symbol column is deleted since there are no entries in either. The effect of this "overlaying" is an approximate 33 percent saving on the size of the table. Table IV shows the effect of "overlaying" Table III.

This change is now incorporated, and the LR(0) sets of items for $G_1$ are constructed. But first, some notation should be reviewed. Earlier it was seen that a particular set was initialized via expansion of some other set. These items in the initialized set are called the basis entries. The other entries of a set, that is, those added via
closure of the basis entries, are called closure entries. It should be noted that all basis entries never have the dot all the way to the left whereas closure entries always have the dot all the way to the left.

The reader is advised that the author's construction of the configuration sets is not identical to DeRemer's (4) in order; however, it is identical in content. For example, the author initializes the first state to be the final state so that its position is known regardless of the grammar being processed.

TABLE IV
THE "OVERLAY" MODIFICATION OF TABLE III

<table>
<thead>
<tr>
<th>STATE</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>?</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>15</td>
<td>-7</td>
<td>-9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>18</td>
<td>19</td>
<td>-9</td>
<td>-7</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-8</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The SLR(1) configuration set computation and table construction for $G_1$ are demonstrated in Tables V and VI.

TABLE V
LR(0) CONFIGURATION SETS FOR $G_1$

<table>
<thead>
<tr>
<th>SET NO.</th>
<th>ITEMS</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$S \rightarrow ?E.?$</td>
<td>final state</td>
</tr>
<tr>
<td>2.</td>
<td>$S \rightarrow ?E?$</td>
<td>initial state</td>
</tr>
<tr>
<td>3.</td>
<td>$S \rightarrow ?,E?$</td>
<td>from 2</td>
</tr>
<tr>
<td></td>
<td>$E^+,E^+T$</td>
<td>closure entries for</td>
</tr>
<tr>
<td></td>
<td>$E^+,T$</td>
<td>the single basis</td>
</tr>
<tr>
<td></td>
<td>$T^+,P^\ast T$</td>
<td>entry; closure</td>
</tr>
<tr>
<td></td>
<td>$T^+,P$</td>
<td>ceases when dot is</td>
</tr>
<tr>
<td></td>
<td>$P^+.i$</td>
<td>left of terminal</td>
</tr>
<tr>
<td></td>
<td>$P^+.i(E)$</td>
<td>symbols</td>
</tr>
<tr>
<td>4.</td>
<td>$S \rightarrow ?E,?$</td>
<td>from 3; expansion</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow E,+.T$</td>
<td>gives final state</td>
</tr>
<tr>
<td></td>
<td>from 3</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>$E \rightarrow T,$</td>
<td>from 3 or 8; no</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow T,$</td>
<td>expansion here</td>
</tr>
<tr>
<td>6.</td>
<td>$T \rightarrow P^\ast T$</td>
<td>from 3 or 8</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow P$</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>$P^+.i$</td>
<td>from 3 or 8</td>
</tr>
<tr>
<td>8.</td>
<td>$P^+.i(E)$</td>
<td>from 3 or 8</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow E^+,T$</td>
<td>indirect recursion</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow E^+,T$</td>
<td>lengthens the set of</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow T^+,P^\ast T$</td>
<td>configuration sets</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow T^+,P$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P^+.i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P^+.i(E)$</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>$E \rightarrow E^+,T$</td>
<td>from 4</td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow T^+,P^\ast T$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow T^+,P$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P^+.i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P^+.i(E)$</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>$T \rightarrow T^+,P^\ast T$</td>
<td>from 6</td>
</tr>
<tr>
<td></td>
<td>$P^+.i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$P^+.i(E)$</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>$P^+(E)$</td>
<td>from 8</td>
</tr>
</tbody>
</table>
TABLE V (Continued)

<table>
<thead>
<tr>
<th>SET NO.</th>
<th>ITEMS</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.</td>
<td>E→E+T</td>
<td>from 8</td>
</tr>
<tr>
<td>13.</td>
<td>E→E+T</td>
<td>from 9</td>
</tr>
<tr>
<td>14.</td>
<td>T→P**T</td>
<td>from 10</td>
</tr>
<tr>
<td></td>
<td>P→(E)</td>
<td>from 11</td>
</tr>
</tbody>
</table>

TABLE VI

SLR(1) TABLE FOR G₁

<table>
<thead>
<tr>
<th>STATE</th>
<th>S</th>
<th>E</th>
<th>T</th>
<th>P</th>
<th>?</th>
<th>+</th>
<th>**</th>
<th>1</th>
<th>( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>-5</td>
<td>-5</td>
<td>10</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>13</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td>-2</td>
<td></td>
<td></td>
<td>-2</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-4</td>
<td>-4</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-7</td>
<td>-7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is now shown how to understand at least part of the structure of L(G₁) by using Tables V and VI. Set #2 shows that an S is an E surrounded by '?'s and that '?' must be the first input symbol. The dot represents the state of the parse. That is, the symbols to the
left of the dot have been recognized (in the stack in the parsing algorithm); and those to the right have not been recognized.

Set #2 has no reduction (no item with the dot to the right), hence a state transition to state (row) #3 is made. (See row #2 of Table VI.) Set #3 (i.e., the basis entries) shows that this set was entered after reading (stacking) a ?, and the next symbol must be an E. The closure entries show the possibilities of what an E can be; that is, since the basis entry in the present sentential form is a derivation, the closure entries show what sentential form can possibly exist after one or more direct derivations of the basis entry. This is similar to a top-down parse of every possible sentence. For all closure entries, it is necessary to read (because of the dot position) and make a state transition.

From previous discussion, it is known that an E is a series of T's separated by +'s. This can be deduced from Tables V and VI. Starting at set #3, which is one time the dot appears to the left of an E, it is seen that the closure entries define an E to be several different configurations. In particular, E+.E+T and E+.T show that, in order to have an E, a reduction on one or the other must be made. E+.T will certainly pop the stack and require a return to set #3 with an E as the next symbol if the next input symbol is + (see row #5 of Table VI), after which a transfer to set #4 and a try to build a longer E will be made.

To see this more clearly, ?i+i+i? is now parsed by using Table VI and using the same parsing technique presented earlier.
In the actual implementation, only states are stacked since, if the symbol is needed for any reason, it can be deduced because each canonical derivation sequence is unique and the stack and table together maintain a history of the parse.

The reader is encouraged to visually correspond the parse with the configuration sets. Perhaps the greatest asset of the SLR(1) method is that any set of productions for a CF grammar can be input, and the user will be provided with the sets and tables which can help lead to an understanding of the language generated by the grammar. And, at the same time, the user is provided with a syntax analyzer with which he can experiment with sentences for purposes of establishing validity.

So far, everything said about SLR(1), at least with respect to
G₁, also applies to LR(0). What is the difference between the two methods? In an actual LR(0) table, rather than enter the reductions only under symbols in the FOLLOW set, they would be entered under every terminal symbol. For example, row #5 in Table VI would have a -3 under **, i, and ( also. It appears DeRemer (4) would do likewise in most cases with his SLR(1) method. This could cause reductions to be made after an error condition is detected; in fact, this is a characteristic of the SLR(k) method.

Clearly, the above action will not work for state (row) #6 in Table VI. This would be an example of a conflict. In SLR(1) table construction, there are two kinds of conflicts. DeRemer (4) uses the term inadequate state for a state with conflicts. An inadequate state is one with either both a reduction entry and a transition entry or two different reduction entries. A table with no inadequate states is a table for an LR(0) grammar (4). A state with only a reduction entry is a reduce state. A state with only transitions is a read state. An inadequate state is said to be solvable if the one symbol look-ahead set (FOLLOW function) indicates which action to take for a given next symbol. An unsolvable inadequate state is one where, with one symbol look-ahead, which action to take still cannot be determined.

State #6 is the only inadequate state for G₁, and it is solvable. By inspecting set #6, it is seen that both a reduction and a transition are present. Of course, the problem is caused by the right grouping of ** and the need to look ahead in the input string to see if the longest T has been found, which is a series of P's separated by **'s. The action of the parsing algorithm on right recursion is to
stack up all of the P's separated by **'s and then reduce from right to left. Two FOLLOW sets need to be computed. That is, FOLLOW (T) needs to be computed since it must be known what can legitimately be the input symbol if the reduction is made. But FOLLOW(P) is not computed for the entry T+P,**T since, by definition, the one symbol look-ahead set for a transition entry is FIRST (symbol to right of dot, FOLLOW (LHS)), which in this case is FIRST (**, FOLLOW (P)). Therefore, the FOLLOW element can be deleted since in a transition entry there is always a symbol to the right of the dot; and this symbol is either a terminal or a non-terminal, X, for which the terminal symbols in H+(X) are selected.

In state #6, the one symbol look-ahead set for T+P,**T is {**}. For FOLLOW(T), the productions are inspected to see what terminal symbols can follow T in a sentential form. From production #3 or #2, it is seen that what can follow an E can also follow a T; therefore, FOLLOW (T) = {+,),?}. Hence, G1 is SLR(1) since the only inadequate state has disjoint one symbol look-ahead sets. This, in essence, is the definition of a SLR(1) grammar (4). A disjoint set implies that, by looking one symbol ahead in the input string, it can be determined which entry of the inadequate state to employ. In state #6 of Table VI, FOLLOW (T) input symbols cause a reduction; and ** causes a transition.

The FOLLOW function can be computed two ways. One way is directly from the productions. The method first computes the relation, F, defined by F(A) = {b | there exists a production C: a,A,B,c. where c,aεV*, AεVN, BεVT and b = B or BεVN and bεVT and bεH+(B)}. Here, any one of c or a may not be present.
Now, if F is represented as a Boolean matrix, then closure of F results in FOLLOW, each row corresponding to $A \in V_N$ and the "true" columns representing the elements of FOLLOW ($A$). For an operator grammar (6), $H^+(G)$ is not needed since every $A \in V_N$ is followed by a terminal symbol or is the last symbol of a RHS.

The second way to compute FOLLOW is developed by DeRemer as a theorem. The proof is found in (4). This method (used in the author's implementation) uses the function $g$ part of the table and $T^*(G)$, the reflexive transitive closure of the inverse of the tail symbol matrix, $T$, defined by $T(A) = \{B \in V_N \mid B^{+\star}aA \text{ where } A \in V_N, a \in V^*\}$. That is, the only concern is with tail symbols that are non-terminals.

An algorithm for computing FOLLOW follows:

1. Compute $T^*(A)$ as above.
2. Start with an empty set, L.
3. For each transition under a symbol in $T^*(A)$ to some state $N$, add to L each symbol $s \in V_T$ such that there is a transition under $s$ from $N$.
4. The resulting set is FOLLOW.

Since FOLLOW is computed for every $A \in V_N$ in the author's implementation, an algorithm is presented for this also. $T, T^*$ are the denotations for the Boolean matrix representation for the relations $T, T^*$, respectively.

1. Compute $T^*$ for every $A \in V_N$; initialize FOLLOW to "false."
2. For each column, $C_1$, of $T^*$; for each row, $R_1$, of $T^*$; if $T(R_1, C_1)$ is true, then for each row, $R_2$, of the table; if TABLE ($R_2, R_1$) is not zero, then for each terminal symbol column, $C_2$; if TABLE (TABLE ($R_2, R_1$), $C_2$) is not zero, then FOLLOW ($C_1, C_2$) + "true."
This algorithm is similar to the Warshall algorithm. The reflexive transitive closure of $T$ is needed as shown in the following discussion. To compute FOLLOW $(P)$, the $p$th column of $T^*$ must have a "true" in it. But this is so only if $P$ is a tail symbol of some $A \in V_N$, which does not occur unless it is assumed the production $A: P$ is present during construction of $T^*$ for some $A \in V_N$. But it is also true that the $p$th row must have a "true" in it, that is, $P$ must have an $A \in V_N$ as a tail symbol since $T^*$ is only computed for non-terminals. The solution is to use a reflexive transitive closure, that is, all productions of the form $A: A$ are assumed to be present only during computation of FOLLOW.

The author's implementation differs from DeRemer's original SLR(1) method in that every state is considered to be inadequate. It is not clear whether DeRemer computes FOLLOW for every $A \in V_N$, but it appears that he does not. The remaining question is what differences exist among LR(1), DeRemer's SLR(1), and the author's SLR(1).

Comparison of Table Construction Methods

It should be clear from Table VI that, if reduction entries are made for all terminal symbol columns, reductions can be made after an error condition is detected. For example, if $ii?i$ is parsed using Table VI and row #7 has -6 under all terminal symbols, it is necessary to reduce the first $i$ to $P$ and, in fact, $P$ to $T$ and $T$ to $E$ before an error is detected; however, by using FOLLOW, the error is detected before the first reduction. It is desirable to detect errors at the earliest possible time; however, it is inherent in DeRemer's method (3) that reductions can take place after an error condition is
detected, and it is also inherent (although not as extensively) in the author's implementation. However, neither will read another input symbol once an error is detected. In Knuth's original method (10), neither reductions nor reading can occur after an error is detected. The reason for this is that Knuth keeps track of what the next input symbol can legitimately be for each entry in every set, but the SLR(1) method assumes that if one symbol may follow another in any sentential form then it may follow it in every sentential form.

Computation of the SLR(1) table for G₃, which was shown to be LR(1), but is not SLR(1), follows. (In fact, it is not SLR(k) for any k.)

<table>
<thead>
<tr>
<th>TABLE VII</th>
<th>SLR(1) CONFIGURATION SETS FOR G₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET NO.</td>
<td>ITEMS</td>
</tr>
<tr>
<td>1</td>
<td>S+?E?.</td>
</tr>
<tr>
<td>2</td>
<td>S+?.E?</td>
</tr>
<tr>
<td>3</td>
<td>S+?.E?</td>
</tr>
<tr>
<td></td>
<td>E+.aAb</td>
</tr>
<tr>
<td></td>
<td>E+.aBc</td>
</tr>
<tr>
<td></td>
<td>E+.dAc</td>
</tr>
<tr>
<td></td>
<td>E+.dBb</td>
</tr>
<tr>
<td>4</td>
<td>S+?E.?</td>
</tr>
<tr>
<td>5</td>
<td>E+a.Ab</td>
</tr>
<tr>
<td></td>
<td>E+a.Bc</td>
</tr>
<tr>
<td></td>
<td>A+.fA</td>
</tr>
<tr>
<td></td>
<td>A+.f</td>
</tr>
<tr>
<td></td>
<td>B+.fB</td>
</tr>
<tr>
<td></td>
<td>B+.f</td>
</tr>
</tbody>
</table>
## TABLE VII (Continued)

<table>
<thead>
<tr>
<th>SET NO.</th>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>E+d.Ac</td>
</tr>
<tr>
<td></td>
<td>E+d.Bb</td>
</tr>
<tr>
<td></td>
<td>A+.f.A</td>
</tr>
<tr>
<td></td>
<td>A+.f</td>
</tr>
<tr>
<td></td>
<td>B+.f.B</td>
</tr>
<tr>
<td></td>
<td>B+.f</td>
</tr>
<tr>
<td>7</td>
<td>E+aA.b</td>
</tr>
<tr>
<td>8</td>
<td>E+aB.c</td>
</tr>
<tr>
<td>9</td>
<td>A+f.A</td>
</tr>
<tr>
<td></td>
<td>A+f</td>
</tr>
<tr>
<td></td>
<td>B+f.B</td>
</tr>
<tr>
<td></td>
<td>B+f</td>
</tr>
<tr>
<td></td>
<td>A+.f.A</td>
</tr>
<tr>
<td></td>
<td>A+.f</td>
</tr>
<tr>
<td></td>
<td>B+.f.B</td>
</tr>
<tr>
<td></td>
<td>B+.f</td>
</tr>
<tr>
<td>10</td>
<td>E+dA.c</td>
</tr>
<tr>
<td>11</td>
<td>E+dB.b</td>
</tr>
<tr>
<td>12</td>
<td>E+aAb.</td>
</tr>
<tr>
<td>13</td>
<td>E+aBc.</td>
</tr>
<tr>
<td>14</td>
<td>A+fA.</td>
</tr>
<tr>
<td>15</td>
<td>B+fB.</td>
</tr>
<tr>
<td>16</td>
<td>E+dAc.</td>
</tr>
<tr>
<td>17</td>
<td>E+dBb.</td>
</tr>
</tbody>
</table>

Comparing the LR(1) and SLR(1) tables for $G_3$, it is seen that Table VII is much shorter than Table II. Also, in Table II, there is a note pointing out the difference between $A_8$ and $A_{11}$. These two sets combine into one set in Table VII, namely set #9; and it is because of this combining that $G_3$ is not SLR(1). In particular, $b$ and $c$ are both in FOLLOW (A) and FOLLOW (B) and, hence, if the next input symbol is $b$ or $c$, it is not known which reduction to make.
A grammar has been given that is not SLR(k) \( (G_3) \), and also a grammar has been given that is SLR(1) \( (G_1) \). For completeness, a grammar that is SLR(2) is now presented. \( G_4 \) is specified by:

1. \( S: ?, E, ? \).
2. \( G: A \).
3. \( C, B \).
4. \( A, b, c \).
5. \( A: a \).
6. \( B: b \).
7. \( C: A \).

<table>
<thead>
<tr>
<th>SET NO.</th>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( S+?G.? ).</td>
</tr>
<tr>
<td>2</td>
<td>( S+.,?G? ).</td>
</tr>
<tr>
<td>3</td>
<td>( S+?,G? ).</td>
</tr>
<tr>
<td></td>
<td>( G+.A ).</td>
</tr>
<tr>
<td></td>
<td>( G+.CB ).</td>
</tr>
<tr>
<td></td>
<td>( G+.Abc ).</td>
</tr>
<tr>
<td></td>
<td>( A+.a ).</td>
</tr>
<tr>
<td></td>
<td>( c+.A ).</td>
</tr>
<tr>
<td>4</td>
<td>( S+?G.? ).</td>
</tr>
<tr>
<td>5</td>
<td>( G+.A ).</td>
</tr>
<tr>
<td></td>
<td>( G+.A.bc ).</td>
</tr>
<tr>
<td></td>
<td>( C+.A ).</td>
</tr>
<tr>
<td>7</td>
<td>( A+.a ).</td>
</tr>
<tr>
<td>8</td>
<td>( G+.Ab.c ).</td>
</tr>
<tr>
<td>9</td>
<td>( G+.cB ).</td>
</tr>
<tr>
<td>10</td>
<td>( B+.b ).</td>
</tr>
<tr>
<td>11</td>
<td>( G+.Abc ).</td>
</tr>
</tbody>
</table>
TABLE IX

SLR(1) TABLE FOR G₄

<table>
<thead>
<tr>
<th>STATE</th>
<th>G</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>?</th>
<th>b</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td>-7/8</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The double entry in row #5 of Table IX indicates that state #5 is unsolvably inadequate since b is in FOLLOW(G) and is to the right of the dot in the transition entry. The set of sentences comprising L(G₄) is {?a?, ?ab?, ?abc?}. Figure 9 shows an attempted parse of ?abc?.

STACK  

UNEXPENDED  

SUFFIX  

NOTES  

2     |   |   |   |   | 3 |   |   |   | initial condition
2?3   |   | abc?|   |   | 3=T(2,?)
2?3a7 |   | bc? |   |   | 7=T(3,a)
2?3A5 |   | bc? |   |   | -5=T(7,b) and 5=T(3,A)

Figure 9. Parsing ?abc? Using Table IX
NOTE: At this point, $T(5,b)$ pertains, but the SLR(1) method has not provided enough information to decide whether to reduce $A$ to a $C$ or read the $b$. If the parser could look ahead one more symbol (i.e., two symbol look-ahead) and see the $c$, then it is clear that $b$ should be read. If the sentence had been $ab?$, then the "pick" would be to reduce rather than read.

```
2?3A5b8   c?   pick 8=T(5,b)
2?3A5b8c1l ?  1l=T(8,c)
2?3G4     ?  -4=T(1l,?) and 4=T(3, G)
2?3G4?1   ?  final state
```

Figure 9. (Continued)
CHAPTER IV

CONCLUSION

This thesis consists of two major parts. The first presents many of the topics covered in a beginning course in formal language theory, but in a way that is meant to appeal to the reader's intuition. A secondary purpose is to get the reader thinking about CF grammars in a way pertinent to the second major part. No single reference covers all of the presented points. Rather, most references tend to cover specific points in a more detailed manner.

The second part presents Knuth's LR(k) method of syntax analysis and, in particular, the SLR(1) method. The result of the full description and numerous examples is twofold. The first provides an efficient language-independent syntax analyzer, which may be used in the development of, for example, a compiler. Parsers for a subset of ALGOL 68, ALGOL 60, and BASIC have been produced with satisfactory results. The second provides a tool by which the input of any context-free grammar yields information which demonstrates the structure of the grammar and the language generated by the grammar. It cannot be overemphasized how useful the configuration sets are in helping to understand a language structure simply by inputting a set of BNF rules. This is especially true in grammars with indirect recursion since visual observation of the production rules yields little insight into the nature of the language.
In conclusion, LR(k) methods are the newest and most general of the methods used for syntax analysis of languages produced by CF grammars. They are shown to be superior to most methods and are more general than any known method for which efficient parsers can be mechanically produced.
A SELECTED BIBLIOGRAPHY


APPENDIX A

LIST OF SYMBOLS
## APPENDIX A

### LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>MEANING</th>
<th>PAGE OF FIRST OCCURREENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>context-free</td>
<td>1</td>
</tr>
<tr>
<td>TWS</td>
<td>translator writing systems</td>
<td>2</td>
</tr>
<tr>
<td>V</td>
<td>vocabulary of a grammar</td>
<td>4</td>
</tr>
<tr>
<td>V*</td>
<td>all strings of elements of V</td>
<td>4</td>
</tr>
<tr>
<td>,</td>
<td>is followed by</td>
<td>5</td>
</tr>
<tr>
<td>;</td>
<td>exclusive &quot;or&quot;</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>may be rewritten as</td>
<td>5</td>
</tr>
<tr>
<td>.</td>
<td>delimiter</td>
<td>5</td>
</tr>
<tr>
<td>ε</td>
<td>set inclusion</td>
<td>5</td>
</tr>
<tr>
<td>LHS</td>
<td>left hand side</td>
<td>5</td>
</tr>
<tr>
<td>RHS</td>
<td>right hand side</td>
<td>5</td>
</tr>
<tr>
<td>V_T</td>
<td>the terminal symbols of V</td>
<td>5</td>
</tr>
<tr>
<td>V_N</td>
<td>the non-terminal symbols of V</td>
<td>5</td>
</tr>
<tr>
<td>{}</td>
<td>set delimiters</td>
<td>6</td>
</tr>
<tr>
<td>+</td>
<td>a direct derivation</td>
<td>6</td>
</tr>
<tr>
<td>+*</td>
<td>a derivation (closure of +)</td>
<td>6</td>
</tr>
<tr>
<td>DPDA</td>
<td>deterministic push down automata</td>
<td>14</td>
</tr>
<tr>
<td>A × B</td>
<td>the Cartesian product of A and B</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>is a subset of</td>
<td>15</td>
</tr>
<tr>
<td>∩</td>
<td>intersection</td>
<td>15</td>
</tr>
<tr>
<td>a R b</td>
<td>a is related to b</td>
<td>15</td>
</tr>
<tr>
<td>∧</td>
<td>logical and</td>
<td>16</td>
</tr>
<tr>
<td>⇒</td>
<td>implies</td>
<td>17</td>
</tr>
<tr>
<td>∪</td>
<td>union</td>
<td>18</td>
</tr>
<tr>
<td>∨</td>
<td>logical or</td>
<td>20</td>
</tr>
<tr>
<td>Σ</td>
<td>summation</td>
<td>20</td>
</tr>
</tbody>
</table>
APPENDIX B

USER'S GUIDE

Input/Output

To use the routine, the user must be familiar with the input and output of the routine. The input comes in on two different files, PARMIN for parameters and PRODIN for the productions. There are 11 input parameters, each an integer in a 4-byte field, left justified on an 80-byte record.

<table>
<thead>
<tr>
<th>PARAMETER NUMBER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>number of productions</td>
</tr>
<tr>
<td>2</td>
<td>maximum number of symbols in any production</td>
</tr>
<tr>
<td>3</td>
<td>maximum number of characters in any symbol or at least number of characters to make every symbol unique</td>
</tr>
<tr>
<td>4</td>
<td>maximum number of unique symbols in the grammar</td>
</tr>
<tr>
<td>5</td>
<td>number of items in all configuration sets combined</td>
</tr>
<tr>
<td>6</td>
<td>number of configuration sets</td>
</tr>
<tr>
<td>7</td>
<td>maximum number of basis entries for any configuration set</td>
</tr>
<tr>
<td>8</td>
<td>= 1 to activate the DEBUG facility</td>
</tr>
<tr>
<td>9</td>
<td>= 1 to count and list solvable inadequate states</td>
</tr>
<tr>
<td>PARAMETER NUMBER</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>- 1 for full printed output</td>
</tr>
<tr>
<td>11</td>
<td>- 1 for punched output in a form to be read by the parsing routine</td>
</tr>
</tbody>
</table>

There are defaults for 0 input parameters 4, 5, 6, and 7; however, these defaults represent only a guess based on the grammar. After an initial run, output statistics allow the user to set these parameters accurately for future runs, if needed.

For the production rules, the format is the LHS (left-hand-side) immediately followed by a colon, followed by one or more blanks, then the RHS (right-hand-side) parts each followed by a comma and one or more blanks. The rightmost part of an alternative is followed by a semicolon and one or more blanks if it is not the last alternative; otherwise, it is followed by a period and one or more blanks. Column 72 must be blank; but, other than the listed restrictions, the format is free form. The first LHS is considered to be the user's "pseudo" goal symbol. That is, it is a goal symbol which may occur in a RHS. All productions with a common LHS must be grouped consecutively. This format allows the productions to be sequenced without affecting the routine.

The reason for using two different input files is that many times the user may wish to store the productions on secondary storage because of their length but, because of the need to change parameters from run to run, it is better for them to be on cards.

The routine is serially reusable, and multiple grammars may be input to the routine. To do this, the user simply places the param-
eter records (one for each grammar) in order in file PARMIN and separates each set of productions with a delimiter card that has a period in the first byte and blanks thereafter. Input of a grammar terminates on end-of-file or a delimiter record for file PRODIN, and the routine terminates on end-of-file for file PARMIN.

The output consists of several of the internal tables. The output of each section of the routine is clearly delimited by labelling. First, a copy of the productions is output followed by statistics on the grammar enabling the user to respecify some of the input parameters in order to reduce the memory requirement of the routine. Next, the encoded form of the productions is output. During input, each symbol is encoded to its position in the symbol table. Next, two mapping arrays are output along with the symbols. The "TO" column maps the symbols to the columns of the SLR(1) table, and the "FROM" column maps the columns of the SLR(1) table to the symbols. If DEBUG is enabled, the next output is messages (perhaps none) reflecting violated restrictions on the grammar. Statistics on the configuration sets are then output. Each of these statistics was put in by the user as a parameter; however, there is no way to really know what these parameters should be until after the routine has run at least once. Once the routine runs for a grammar, these output statistics will allow the user to set the parameters more accurately. All parameters should be set as small as possible since storage is allocated per the parameters. Next, the LR(0) configuration sets are output in a similar format to that presented in the body of this thesis. Also output is the dot position ("2" is all the way to the left), the upper bound of the set (all sets are in a single vector),
and the number of basis entries. Finally, the full SLR(1) table is output along with the column-to-symbol relationships and results of the inadequate state counter.

Restrictions

There are no restrictions on the input except the format and size of the host machine. This can be a factor for small-to-medium machines. For example, ALGOL 60 takes approximately 200K bytes to execute. A possible remedy for this is to store the data structures on scratch files; however, this would greatly increase execution time since the structures are not processed in any set manner. That is, processing is highly dependent on the grammar. Also, since the SLR(1) table is quite sparse, a sparse matrix technique such as found in (9) might be employed to some advantage.

Job Control Language Required

The following JCL is required if the source deck is input:

```
//JOB NAME JOB (XXXXX,YYY-YY-YYYY,5), 'NAME'
//STEP1 EXEC PLLPCG
//PL1L.SYSIN DD *
    --SOURCE DECK--
//GO.PARMIN DD *
    --PARAMETER CARDS--
//GO.PRODIN DD *
    --PRODUCTIONS--
//GO.PRINT DD SYSOUT=A
```
The routine is presently stored in load module form and may be executed with the following JCL.

//JOB NAME JOB (XXXXX,YYY-YY-YYYY,5), 'NAME'
//STEP1 EXEC PGM=SLR1
//STPLIB DD DSN=OSU.ACT11098.PROG,DISP=SHR
//PARMIN DD *
--PARAMETER CARDS--
//PRODIN DD *
--PRODUCTIONS--
//PRINT DD SYSOUT=A
//PUNCH DD SYSOUT=B,DCB=BLKSIZE=80

Suggested Modifications

In addition to the different storage techniques mentioned earlier, there are other modifications the user may want to make. For example, in the present version, SUBSCRIPTRANGE, STRINGRANGE, and SIZE are enabled for the whole routine; however, the author believes that only the input section needs such checks and that the other sections contain the logic to take care of these conditions. The reader familiar with the PL/1 compiler will recognize the savings in both compile and execution time that could be realized by turning off these condition checks. However, for small grammars, the difference in execution time is almost negligible because of the overall speed. For example, $G_1$ executes in two seconds.
The user may also want to output running statistics on the configuration sets since, if the parameters are too small, the program fails with only a brief diagnostic whereupon the user must increase the parameters and retry the grammar. For grammars with a high degree of recursion such as ALGOL 68, the problem of setting the parameters large enough and still staying within the machine storage limits can be quite frustrating. The following table may help to serve as a guide.

<table>
<thead>
<tr>
<th>GRAMMAR</th>
<th>G₁</th>
<th>ALGOL 60</th>
<th>ALGOL 68 (subset)</th>
<th>BASIC (simple precedence form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of productions</td>
<td>7</td>
<td>181</td>
<td>159</td>
<td>99</td>
</tr>
<tr>
<td>Number of parts</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Number of symbols</td>
<td>10</td>
<td>141</td>
<td>99</td>
<td>102</td>
</tr>
<tr>
<td>Number of characters</td>
<td>4</td>
<td>31</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Number of sets</td>
<td>15</td>
<td>304</td>
<td>310</td>
<td>174</td>
</tr>
<tr>
<td>Combined length of sets</td>
<td>50</td>
<td>2191</td>
<td>5592</td>
<td>957</td>
</tr>
<tr>
<td>Number of basis entries</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Reduction queue</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
If the user wants a stripped-down, super-fast version, he may also completely remove the debug section without affecting the routine. Also, he may want to output the results of the input section onto secondary storage so that, if the routine fails later because of input parameters, he may bypass the input section (with the exception of parameter input) on subsequent runs. Also, he may choose to write the output to secondary storage instead of punched cards since, for example, the BASIC grammar produces approximately 900 cards. Of course, one must realize that more output is produced than is actually needed (for example, the MAPFROM array); but, if meaningful diagnostics are to be produced by the parser, all of the output is necessary.

An alternative to punching or writing out tables would be to actually produce the parser program (minus the scanner, of course). The parser is only a skeleton whose DECLARE statements could be filled in with the proper data with the INITIAL attribute, which the routine could easily do.

If the routine is to be used to produce a parser for the language generated by the input grammar, the user may want to precede all terminal symbols in the grammar with some special symbol, for example, the double quote, because the symbol table method used is a balanced binary tree method (12) and such a prefix on the terminal symbols will tend to cause all of them to be placed in the same subtree, slightly decreasing the average look-up time. It should be pointed out that only the terminal symbols along with the symbol's position need be output to the parser if the parser's scanner uses some other look-up technique (e.g., linear search); however, this is not recommended.
APPENDIX C

PROGRAM LISTING
THE WORK HEREIN IS PARTIAL FULFILLMENT OF THE MASTER'S PROJECT
REQUIRED FOR THE MASTER OF SCIENCE DEGREE IN COMPUTER SCIENCE.

PROJECT ADVISOR: DR. J. VAN DOREN

REFERENCES:
1. COMPILEtl CONSTRUCTION - GRIES
2. SIMPLE LARIK GRAMMARS - DE REMER CALM 14 P 453-460 JULY 1971
3. PRACTICAL TRANSLATORS FOR LARIK LANGUAGES - DE REMER PH.D. THESIS
4. SIMPLE LARIK GRAMMARS - DEFINITION AND IMPLEMENTATION - DE REMER
5. THE CARE AND FEEDING OF LARIK GRAMMARS - AND JULLIAN PROC.
6. A TECHNIQUE FOR SPEEDING UP LARIK PARSERS - AND AND JULLIAN ACM
7. ON THE TRANSLATION OF LANGUAGES FROM LEFT TO RIGHT - KNUH
8. A THEOREM ON BOOLEAN MATRICES - WARSHALL JACM 11-12 1962
9. AN ALGORITHM FOR MAINTAINING DYNAMIC AVL TREES - VAN DOREN AND GRAY

THE ROUTINE CONSISTS OF 3 BASIC SECTIONS, THE THIRD BEING DIVIDED INTO
2 SUBSECTIONS, EACH OF THE FIVE CONTAINED IN A BEGIN-END BLOCK. ALSO
2 INTERNAL PROCEDURES ARE EMPLOYED. A SCHEMATIC DIAGRAM OF THE BLOCK
STRUCTURE FOLLOWS.

SLRl: PROC
REUSABLE:
BEGIN THE_WHOLE_THING:
BEGIN READER_SECTION:
BEGIN DEBUG_SECTION:
BEGIN TABLE_GENERATE_SECTION:
BEGIN LRO_GENERATE:
BEGIN SLRl_GENERATE:
BEGIN WARSHAL:
BEGIN BSLR:
BEGIN END THE_WHOLE_THING
GO TO REUSABLE
ERROR:
ERROR MESSAGE OUTPUT
GO TO REUSABLE
END REUSABLE

© 1972 PAGE 2

INPUT:
FROM FILE PARX THE FOLLOWING PARAMETERS ARE READ IN 11 FIELDS OF 4:
1. N = NUMBER OF PRODUCTIONS TO BE INPUT
END SLRl
THE RATIONALE FOR THE HEAVY USE OF BLOCK STRUCTURING IS TO REDUCE THE
INHERENT NEED FOR LARGE AMOUNTS OF STORAGE BY TAKING FULL ADVANTAGE OF
THE DYNAMIC STORAGE CAPABILITIES OF THE SOURCE LANGUAGE. For SMALLER
MOST MACHINES, SCRATCH FILES RESULTING IN SLOWER EXECUTION TIME WOULD
BE NEEDED FOR LARGE GRAMMARS.

SECTION DESCRIPTION:
REUSABLE: THE ALL INCLUSIVE REUSABLE BLOCK IS PRESENT ONLY TO ALLOW
MULTIPLE GRAMMAR INPUT; THAT IS, THE PROGRAM IS SERIALLY REUSABLE.
WHILE THING: PARAMETERS SETTING CERTAIN LIMITS ON THE GRAMMAR AND
TABLES ARE INPUT OUTSIDE THE BlocK AND USED WITHIN THE BLOCK FOR
DYNAMIC DECLARATION PURPOSES.
READER: THIS SECTION INPUTS AND ENCODES THE PRODUCTIONS, BUILDING A
SYMBOL TABLE USING BSRDI, AND BUILDS CERTAIN MAPPING ARRAYS FOR DATA
STRUCTURES USED.
DEBUG: THE EXECUTION OF THIS SECTION IS USER CONTROLLED AND PERFORMS
CERTAIN CHECKS ON THE GRAMMAR.
TABLE GENERATE: CONTAINS ONLY DECLARATIONS NEEDED FOR THE FOLLOWING 2
SECTIONS.
LRO GENERATE: THIS SECTION FIRST GENERATES THE CONFIGURATION SETS
AS IF THE GRAMMAR IS LR(0); THEN THE TRANSITION ENTRIES ARE PLACED
IN THE SLRl TABLE. THE FILLING IN OF REDUCTION ENTRIES IS
POSTPONED UNTIL THE FOLLOWING SECTION.
SLRl GENERATE: THIS SECTION GENERATES THE COMPLETE SLRl PARSING
TABLE AND IF USER SELECTS COUNTS AND LISTS INADEQUATE STATES AND
PUNCHES THE TABLE, SYMBOL TABLE, AND OTHER STATISTICS NEEDED BY THE
PARSER.

PROCEDURE DESCRIPTION:
BLlSLR: THE INSERT SECTION OF A BINARY TREE SYMBOL TABLE
IMPLEMENTATION C.F. REFERENCE.
WARSHAL: A PROCEDURE TO PERFORM THE WARSHALL ALGORITHM ON AN INPUT BIT
MATRIX C.F. REFERENCE.
THE UUT ARE

ACCUF

~

S

OTHERWISE, IMEDIATELY

10.

THI

S IS

LAST

THE

COFJR

E<IU

PRODUCTED WIIH A CO"IMU ...

I

IF THE

VCTR

Ii

NON-TEKMINALS,

mapTO.

IT IS

THE DOT POSITION OF EACH

A COMMENT.

THAT

IS, IT IS A GOAL

SYMBOL.

GOAL SYMBOL.

IF THE

EXISTS AT THE END OF THE

LSTM, UJTPUT HAT! STC.

THE Ri:UCTION LST.

The IH, UJTPUT LST.

THE

RHT-HANO-

PUN.

THE

PU.

A

I

P

I

THE LHS IS CONSIDERED TO BE THE USER'S "PRZDIUM"

SERED SYMOL.

THAT IS, IT IS A GOAL SYMBOL WHICH MAY OCCUR IN A LHS.

ALL PRODUCTIONS WITH A COMMON LHS MUST BE WRAPPED CONSECUTIVITY.

FUR

MULTIPLE Grammar INPUT, EACH INPUT IS DELIMITED BY A CARY WITH A

PERIOD IN COLUMN 1. NOTICE THIS ALLS THE PRODUCTIONS TO 0.

SEQUENCED WITHOUT AFFECTING THE ROUTINE.

OUTPUTS

ALL SIGNIFICANT INTERNAL TABLES AND STATISTICS ARE PAINTED AND

LABELLED IF THE PRINT PARAMETER IS ENABLED. ALSO ALL DATA NEEDED BY

THE PARSER IS PRINTED IF 0 SELECTED BY THE USE.

THE PARSER IS

ENCLOSED AS A COMMENT. NOT BY ALTERING THE DCR STATEMENT FOR

PUNCH, THE OUTPUT COULD BE ROUTED TO A DATA SET OR SECONDARY STORAGE.

TECHNICAL ESSAYS SUGGESTED, THE PUNCHED OUTPUT FOR THE BASIC

GRAMMAR IS APPARELATELY 900 CARDS.

MAJOR DATA STRUCTURES

MANY ARRAYS AND VECTORS ARE USED. NO SORTING IS DONE. THE INPUT

PRODUCTIONS ARE NOT STORED; HOWEVER, THEIR ENCODED FORM IS IN CORE.

THE SYMBOL SET IS LÌNEAR POSITION IN THE BINARY TREE-

STRUCTURED SYMBOL TABLE BUILT BY ASTSLK. THE INPUT SYMBOLS ARE SAVED

AND SENT TO THE PARSER FOR ERROR MESSAGE CAPABILITIES AND IN THE CASE OF

ERROR TERMINAL SYMBOLS FOR BLANKING PURPOSES. THREE MAPING ARRAYS ARE

MAINTAINED. MAPD HAS AN ENTRY FOR EACH SYMBOL SUCH THAT BY APPLYING

MAPO TO THE CODED SYMBOL A UNIQUE COLUMN OR ROW OF AN ARRAY IS

OBTAINED SUCH THAT THE NON-TERMINALS ARE GROUPED IN POSITIONS 1 TO

NUMBER OF NON-TERMINALS, AND THE TERMINALS ARE GROUPED IN POSITIONS

NUMBER OF NON-TERMINALS # 1 TO NUMBER OF SYMBOLS. MAPDij IS THE

I nVERSE OF MAPD. INDEX I ESTH. DIJ. DURING INPUT SUCH THAT ENDEX APPLIED TO

MAPD TO APPLIED TO A CODED NON-TERMINAL YIELDS THE FIRST

I RANK TERMINAL SYMBOL IN THE COLUMN IN WHICH THE SYMBOL IS THE

LEFT-HAND-

SIDE. TREE IS THE SYMBOL TABLE MAINTAINED BY ASTSLK AND IS DOCUMENTED.

ELSEWHERE F.G.F. REFERENCE. TABLE IS THE LEXED THEN SLR LEXED TABLES. EACH

I RANK DEFINES A SET AND THE COLUMNS CORRESPOND TO THE SYMBOLS (ARRANGED

SET IS A VECTOR THAT HOLDS ALL CONFIGURATION SETS. SLIM HOLD THE

LAST POSITION IN SET FOR EACH CODE, AND BASIS HOLDS THE LAST POSITION

IN THE SET BASE POSITION OF EACH SET. OUT-POSITION IS AN ARRAY

WHICH HOLDS THE DOT POSITION OF EACH BASIS ENTRY OF EACH SET (AN ENTITY

UP TO THE DOT IS TO THE LEFT OF THE COLUMN PARALLEL TO SET THAT IS SET TO 1 IF THE CORRESPONDING SET ELEMENT

EITHER CANCER OR HAS BEEN USED IN EXPANSION.

PROGRAM LOGIC

THE INPUT-EXECUTE SECTION IS STRAIGHT-FORWARD, AND THE USER WILL MAKE

NO TROUBLE DETECTING THE LOGIC BY FOLLOWING THE SOURCE CODE. IF OCCURS

IS SELECTED, THEN THE EXEC. SECTION IS ENTERED. THE DEBUG SECTION CAN

BE ALIETED WITHOUT AFFECTING THE PROGRAM. IT IS THE DUTY OF THE IMPLEMENTATION OF SOME OF THE GRAMMER checks OF GRIES. THE HEART OF

THE PROGRAM IS THE TABLE GENERATE SECTION. IN THE LHS SECTION, THE

FIRST SET IS INITIALIZED TO PRODUCTION I WITH THE DOT TO THE RIGHT.

THIS IS THE FINAL STATE, THE SECOND SET IS INITIALIZED TO THE FIRST

PRODUCTION (ALL SET ENTRIES ARE PRODUCTION NUMBERS) WITH THE DOT TO

THE LEFT. THE SET IS NOW CLOSED. THIS CONSISTS OF ENTERING INTO THE

SET ALL PRODUCTIONS (SUKEEP THIS SYMBOL TO THE RIGHT OF THE DOT.

THES ENTRIES ARE KNOWN AS CLOSURE ENTRIES. THE DOT IS ASSUMED TO BE

TO THE LEFT IN ALL CLOSURE SET ENTRIES. EACH OF THE CLOSURE ENTRIES

MUST ALSO BE CLOSURE. THE CLOSURE CONTINUES UNTIL THE SYMBOL TO THE RIGHT OF THE OUT OF ALL UNCLOSED CLOSURE ENTRIES IS A TERMINAL OR A CLOSURE.

FULL D Cut EXPANSION IS USED TO INITIATE A NEW SET. THE "CANDIDATE" FOR EXPANSION IS THE FIRST SET ENTRY WHERE MARKER BIT IS 0. FOR WHICHEVER SET IT IS IN, ALL OF THOSE SETS

ENTRIES WITII THE SAME SYMBOL TO THE RIGHT OF THE DOT ARE MARKED AND THEN USED TO FORM THE BASIS ENTRIES (THE DOT MOVES RIGHT)

POSITION) OF A NEW SET PROVIDING SUCH ACTION WOULD NOT CAUSE

DUPlication OF AN EXISTING SET. IF BOTH CLOSURE AND EXPANSION, A

DUPLICATE IS NOT THE SAME SET ELEMENT BUT ALSO THE SAME DOT

POSITION. IF, WHEN EXPANDING THE MOVEMENT OF THE DOT IS TO THE

RIGHT, THEN THIS IS A SET (TRUE STATES WITH A REDUCTION ASSOCIATE

WITH IT. THE SET ELEMENT, A PRODUCTION NUMBER, IS NEGATED AND ENTERS INTO REDUCEI3 PROVIDING THERE IS NO PREVIOUS ENTRY IN REDUCEI3. IF 

THERE IS THEN, REDUCEI3 IS SET TO THE NUMBER OF SUCH ENTRIES AND THE

ENTRIES THEMSELVES ARE STORED IN A QUEUE (MULT, 5<br>QUESTIONS). ANY ENTRIES

WHOSE THE DOT THE RIGHT ARE MARKED TAKEN OFF EXPANSION LIST) SINCE

IF THE DOT IS TO THE LEFT, THEY CANNOT BE USED FOR EXPANSION SINCE

THE DOT CANNOT BE MOVED FURTHER TO THE RIGHT. KEEP IN MIND THAT THE

DOT CAN BE A FIXED BASIS ENTRIES IS IN THE ARRAY UJTPUT-POSITION ALONE

THE DOT POSITION OF CLOSURE ENTRIES IS ASSUMED TO BE 2 ITU THE LEFTS.

THE ACTION OF CLOSING THEN EXPANDING CONTINUE UNTIL ALL ENTRIES ARE

MARKED. THE SECOND SET, WHICH IS THE NUMBER OF THE NEXT ENTRY IS THE

CERTAIN SYMBOL TO THE RIGHT OF THE DOT WHILE WITHIN A CERTAIN SET

ENTERED INTO. THAT IS, TABLES11 J K WHERE J IS THE SET THE

POSITION OF THE DOT (0 IS DEFINED AS THE STATE OF THE MAPD SUCH

K IS THE N E W SET GENERATED. A SIMILAR ENTRY IS MADE IF R IS

THE SET WHICH WOULD BE DUPLICATED BY A PARTICULAR EXPANSION. THE

SLR GENERATE SECTION COMPUTES THE FOLLOW FUNCTION PER BE READS.

THEM, THEN THE PROCEED TO ENTER THE REDUCTIONS (PREVIOUSLY STORED IN

A INPUT FROM LHS TO HFS INTO ALL COLUMNS REPRESENTING SYMOL.

IF IN FULLCOLUMN WHERE A IS THE LHS OF THE PRODUCTION INVOLVED IN THE

REDUCTIONS OF A PARTICULAR SET I SET UP TABLE. AFTER SUCH ENTRY.

THE ROW NOW CONTAINS BOTH POSSIBLE STATE TRANSITIONS AND REDUCTIONS.

HENCE A STATE (INADEQUATE STATES ARE THOSE WITH MORE THAN ONE REACTION OR A REDUCTION AND A STATE TRANSITION UNDER THE SAME SYMBOL, IF

MORE THAN ONE ENTRY IS ATTEMPTED IN ANY TABLE POSITION, THEN THE GRAMMAR IS

NOT SLR).

*** THE FOLLOWING IS A SAMPLE PARSE WHICH USES THE SLREY TABLES ***
keywords: procedure (iter., flag, pos, trellis)

procedure astshg is the search section of bst c/p, reference:
parameters:
  item - key for retrieval, insertion or deletion
  flag - status code for attempted function
  pos - linear index of node inserted or retrieved
  tree - structure containing binary search tree,
  available space list and node count
/
  declare
    (flag, pos) field bin (31,0);
  item char (*);
  1 trellis
    2 node (* char (*)
    2 ll (*) fixed bin;
    2 rl (*) fixed bin;
    2 tag (*) hit (*) alline ed,
  2 avail fixed bin (31,0);
  2 count fixed bin (31,0);

search:
begin
  /*
    search for node with key value contained in item,
  */
    declare curr fixed bin (31,0);
    currll(0,1);
    dc while (curr == 01)
      if item = node(curr) then
        /* return success */
        do
        flag +
        pos = curr
        return;
      end
      if item < node(curr) then currll(curr)
      else currll(curr)
      end
      /* return failure */
      pos +;
      flag -;
      return;
    end search;
  end pel:rrs
  end main
end parser.

end of section

variables description fall section:
bad - working variable - 0 for any symbol not "within" the user's
pseudo global symbol
basis - a vector holding the position of extent of each bases set
in the vector holding all configuration sets
buf - input buffer for productions
buf+ - vector overlaid on buf
canonical - a production number in some set to be used for possible
        expansion
config_set_limit = input parameter, dimension of vector that holds
        all configuration sets
count_inadequate_states = input parameter, 1 if user wants action
variable represents
  dot_production = input parameter, 1 if user wants action
  variable represents
  dot_position = a matrix holding the dot positions of basis sets
  element = the first transition in the row of table being scanned
  end - a vector such that element (any symbol) is the first
  production number of which symbol is the lhs
  err - error switch
  fence = the "fence" of a binary search
  follow = a bit matrix of nonterminals vs nonterminals (see above)
  input_position - input parameter,
  limit = limit of input production symbol
  marty = a vector such that marty (any symbol) maps to
  columns of a matrix such that the nonterminals are grouped
  as are the terminals
  mapfork = the inverse of martyr
  marker - a bit vector whose 1 th entry is 1 if the 1 th
            configuration set element cannot be used for expansion (ur
            has been used)
  master_fatal = error switch, trail if unable to generate skill table
  multi_reduce = a vector used to hold reductions for a given set if
               more than 1
  name = an input production symbol
  nchars = counter of basis elements
  no_bases = input parameter, maximum number of basis elements for any
            set
  no_chars = input parameter, maximum number of characters in any input
            production symbol
  nilinad = inadequate state counter
  nilnon = nonterminals counter
  nilpos = input parameter, maximum number of input productions
  nilprods = input parameter, maximum number of configuration sets
  nilsym = input parameter, maximum number of input symbols
  nilterm = number of terminal symbols
  nparts = parts counter
  nsets = configuration sets counter
  pailable = input file size parameters
  place = the first production of a group with the same lhs
  print = output print file
  printl = input file for productions (blocksize = 401
  punch = output punch file
  prod = an array of encoded productions - the code for a symbol is
        its position in the symbol table
  pt = pointer to unrecognized portion of buf
  reduce = true as soon as a reduction is detected in present state
  reduce - reduce a vector that holds the negative reduction. if any, for
            a state-if more than one then it holds how many and they are
            stored in subtract
  set = the vector holding all configuration sets
  set_limit = the "top" of set
  sig = true if any production becomes "marked" during last pass
  slims = a vector holding the extent in set of each configuration
        set
DECLARE
OSTINT ENTRY.
OSTSLR ENTRY (CHARACTER(NO_CHARS),).
AMARR ENTRY.
1 TRAIL.
2 NODE (NO_SYMS) CHARACTER (NO_CHARS) INITIAL (+ 1).
2 LL (NO_SYMS) FIXED BINARY (13,0).
2 RL (NO_SYMS) FIXED BINARY (13,0).
2 TAG (NO_SYMS) BIT (0) ALIGNED.
2 AXST FIXED AXSTY (13,0,0).
2 COUNT FIXED AXSTY (13,0).
ENDX (NO_PRODS+2*NO_PARTS) FIXED BINARY (13,0). INITIAL (1).
ENDEX (NO_SYMS+1) FIXED BINARY (13,0) INITIAL (4).
NAME (NO_SYMS+1) FIXED BINARY (13,0) INITIAL (1).
NO_TERM FIXED AXSTY (13,0) INITIAL (1).
/* THIS IS THE INPUT SECTION. */
READERS SECTION:
BEGIN:
DECLARE
PARDIN FILE INPUT RECORD:
BUF (100) CHARACTER (1) DEFINED BUF.
NAME CHARACTER (100) VARYING.
(1,1) FIXED BINARY (13,0) INITIAL (1).
NO_SYMS (NO_SYMBOLS) FIXED BINARY (13,0) INITIAL (1).
 Ital, POS, PT, LNAMEx FIXED BINARY (13,0).
ENDFILE (PARDIN) GO TO ENDFILE.
ENFILE NAME SIGNAL ERROR.
ENFILE NAME SNAP SIGNAL ERROR.
END FILE (PARDIN) ERR.
/* BEGIN OUTPUT FOR INPUT-ENDCODE SECTION...*/

/* INITIALIZE THE BRICKS AS IS SYMBOL TABLE AND INSERT GENERATED
GOAL SYMBOL AND DELIMITERS.*/
/* PUT FILE (PRINT) SKIP EDIT (INPUT PRODUCTIONS) (1,1) */
/* = GOAL = */
/* USER'S GOAL SYMBOL = */
/* 2 (COL).1, SKIP IT.
CALL ODTBINT (TREE).
CALL ODTSLR (OUTL,FLAGS) PRINT TREE.
GETCARD:
READ FILE (PARDIN) INTO BUF.
/* CHECK FOR "NEW" ALLLINES MULTIPLE GRAMMAR INPUT. */
IF (BUF) = 1 THEN GO TO ENDPARDIN.
/* CARD MUST END WITH NON-BLOCK TO PREVENT STRINGANAGE. */
/* SUBSTR(B) != 1 */
NEXTSYM:
DO = TPT Y 1 WHILE (OUTL) = 1.
END
NAME = SUBSTR(B, PT, 1). (SUBSTR(B, PT, 1))
LNAME = LENGTH(NAME)
UNNO534
UNNO535
UNNO536
UNNO537
UNNO538
UNNO539
UNNO540
UNNO541
UNNO542
UNNO543
UNNO544
UNNO545
UNNO546
UNNO547
UNNO548
UNNO549
UNNO550
UNNO551
UNNO552
UNNO553
UNNO554
UNNO555
UNNO556
UNNO557
UNNO558
UNNO559
UNNO560
UNNO561
UNNO562
UNNO563
UNNO564
UNNO565
UNNO566
UNNO567
UNNO568
UNNO569
UNNO570
UNNO571
UNNO572
UNNO573
UNNO574
UNNO575
UNNO576
UNNO577
UNNO578
UNNO579
UNNO580
UNNO581
UNNO582
UNNO583
UNNO584
UNNO585
UNNO586
UNNO587
UNNO588
UNNO589
UNNO590
UNNO591
UNNO592
UNNO593
UNNO594
UNNO595
UNNO596
UNNO597
UNNO598
UNNO599
UNNO600
UNNO601
UNNO602
UNNO603
UNNO604
UNNO605
UNNO606
UNNO607
UNNO608
UNNO609
UNNO610
UNNO611
UNNO612
UNNO613
UNNO614
UNNO615
UNNO616
UNNO617
UNNO618
UNNO619
UNNO620
UNNO621
UNNO622
UNNO623
UNNO624
UNNO625
UNNO626
UNNO627
UNNO628
UNNO629
UNNO630
UNNO631
UNNO632
UNNO633
UNNO634
UNNO635
UNNO636
UNNO637
UNNO638
UNNO639
UNNO640
UNNO641
UNNO642
UNNO643
UNNO644
UNNO645
UNNO646
UNNO647
UNNO648
UNNO649
UNNO650
UNNO651
UNNO652
UNNO653
UNNO654
UNNO655
UNNO656
UNNO657
UNNO658
UNNO659
UNNO660
UNNO661
UNNO662
UNNO663
UNNO664
UNNO665
UNNO666
UNNO667
UNNO668
UNNO669
UNNO670
UNNO671
UNNO672
UNNO673
UNNO674
UNNO675
UNNO676
UNNO677
UNNO678
UNNO679
UNNO680
UNNO681
UNNO682
UNNO683
UNNO684
UNNO685
UNNO686
UNNO687
UNNO688
UNNO689
UNNO690
UNNO691
UNNO692
UNNO693
UNNO694
UNNO695
UNNO696
UNNO697
UNNO698
UNNO699
UNNO700
UNNO701
UNNO702
UNNO703
UNNO704
UNNO705
UNNO706
UNNO707
UNNO708
UNNO709
UNNO710
UNNO711
UNNO712
UNNO713
UNNO714
UNNO715
UNNO716
UNNO717
UNNO718
UNNO719
UNNO720
UNNO721
UNNO722
UNNO723
UNNO724
UNNO725
UNNO726
UNNO727
UNNO728
UNNO729
UNNO730
UNNO731
UNNO732
UNNO733
UNNO734
UNNO735
UNNO736
UNNO737
UNNO738
UNNO739
UNNO740
UNNO741
UNNO742
UNNO743
UNNO744
UNNO745
UNNO746
UNNO747
UNNO748
UNNO749
UNNO750
UNNO751
UNNO752
UNNO753
UNNO754
UNNO755
UNNO756
UNNO757
UNNO758
UNNO759
UNNO760
UNNO761
UNNO762
UNNO763
UNNO764
UNNO765
UNNO766
UNNO767
UNNO768
UNNO769
UNNO770
UNNO771
UNNO772
UNNO773
UNNO774
UNNO775
UNNO776
UNNO777
UNNO778
UNNO779
UNNO780
UNNO781
UNNO782
UNNO783
UNNO784
UNNO785
UNNO786
UNNO787
UNNO788
UNNO789
UNNO790
UNNO791
UNNO792
UNNO793
UNNO794
UNNO795
UNNO796
UNNO797
UNNO798
UNNO799
UNNO800
UNNO801
UNNO802
UNNO803
UNNO804
UNNO805
UNNO806
UNNO807
UNNO808
UNNO809
UNNO810
UNNO811
UNNO812
UNNO813
UNNO814
UNNO815
UNNO816
UNNO817
UNNO818
UNNO819
UNNO820
UNNO821
UNNO822
UNNO823
IF LNAME < 3 THEN  DO  
  IF LNAME = 0 THEN GO TO GETCARD; 
  ELSE GO TO ERRORS; 
END; 
LNAME = MAX(LNAME, LNAME-2); 
"* INSERT IF NOT PRESENT ELSE EFFECTIVELY A SEARCH */ 
CALL HSTSLR (SUBSTR(NAME, LNAME-2), NAMEPUSH); 
IF SUBSTR(NAME, LNAME-1) = "*" THEN 
  DO; 
  SWITCH = NEXTSYM; 
  ENTER; 
  J = I + I; 
  NAMEPUSH = NAMEPUSH + I; 
  PUT FILE (PRINT) EDIT (NAME1); 
  GO TO SWITCH; 
END; 
IF SUBSTR(NAME, LNAME-1) = "*" THEN 
  DO; 
  IF NAMEPUSH = NAMEPUSH + I; 
  J = I + I; 
  NAMEPUSH = NAMEPUSH + I; 
  PUT FILE (PRINT) EDIT (INDEX_PUSH); 
  IF NAMEPUSH = NAMEPUSH + I THEN 
    /* SET MAP ARRAYS FOR NON-TERMINALS */ 
    DO; 
      NO_NON = NO_NON + 11; 
      ENDINDEX_NON = I + I; 
      NAMEPUSH = NAMEPUSH + I; 
      MAPTOKNON = NO_NON; 
      MAPNAME_NON = NO_NON; 
    END; 
    GO TO NEXTSYM; 
END; 
/* OPTIONALLY COULD SET SWITCH TO GETCARD IF IT IS KNOWN THAT EACH 
INPUT LHS STARTS A NEW CARD. */ 
/* */
SWITCH = NEXTSYM; 
GO TO ENTER; 
END; 
IF SUBSTR(NAME, LNAME-1) = "*" THEN 
  DO; 
  SWITCH = SEMI; 
  GO TO ENTER; 
SEND; 
/* */ 
/* OUTPUT STATISTICS ON INPUT GRAMMAR. */ 
/* PUT FILE (PRINT) SKIP EDIT 
"* USER REQUESTED ACTUALLY REEED*", (34) */ 
/* */
WITHIN MAPFROM([11]), MAPTO([11]) := [11];
END;
END;
/* CLOSE LOOP VIA MARSHALL ALGORITHM. */
CALL MARSHALL (INPUT);
/* ANY ZERO IN USER'S GOAL RUN (COL 3 FORWARD MEANS SAME SYMBOL IS
"WITHIN" THE USER'S GOAL. */
DO J := 2 TO NO_PRODS;
IF PRODUCT[I] = PRODUCT[J] THEN PUT FILE (PRINT) EDIT (NODE(MAPFROM[I]));
/* CANNOT APPEAR IN ANY SENTENTIAL FORM. */ SKIP, 2 AI;
END;
/* ALGORITHM FOR DETECTING PRODUCTIONS THAT CANNOT BE USED TO
DERIVE A SENTENCE, C.F. REFERENCE. */
DO K := 1 TO NO_PRODS;
IF MARK[I] THEN GO TO END1;
DO L := 1 TO NO_PRODS WHILE (PRODUCT[I] = PRODUCT[I]);
/* LINEAR LOOK-UP FOR NON-TERMINAL AS A LHS. */
IF L = 2 TO NO_PRODS WHILE (PRODUCT[I] = PRODUCT[I]);
IF L <= NO_PRODS THEN GOTO END2;
DO M := 1 TO NO_PRODS WHILE (PRODUCT[I] = PRODUCT[I]);
IF M = 2 TO NO_PRODS WHILE (PRODUCT[I] = PRODUCT[I]);
END;
END;
END;
END;
DO J := 1 TO NO_PRODS;
IF MARK[I] THEN GOTO END1;
DO L := 1 TO NO_PRODS;
IF MARK[I] THEN GOTO END1;
DO K := 1 TO NO_PRODS;
IF MARK[I] THEN GOTO END1;
DO J := 1 TO NO_PRODS;
DO I := 1 TO NO_PRODS;
IF PRODUCT[I] = PRODUCT[J] THEN GOTO
DO K := 1 TO NO_PRODS;
IF PRODUCT[I] = PRODUCT[J] THEN GOTO
END;
PUT FILE (PRINT) EDIT (ENDPRODUCT[I], 1),
STARTS A DUPLICATE RHS FOR PRODUCTIONS. */
/* NOTICE NO ERROR ON DUPLICATE RHS SINCE THE SLR(1) METHOD IS
UNAFFECTIONED BY SUCH THINGS, HOWEVER A MESSAGE IS PRINTED BECAUSE
OFTEN THIS CONDITION LEADS TO UNSOLVABLE INHIBITIZE STATES. */
END;
END;
/* OUTPUT */
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
END;
IF I.E. THE FIRST AT THIS POINT FENCE IS '*

I.*

CHECK TAKE OFF EXECUTED AND SET REDUCE TO NUMBER OF ELEMENTS OF THIS SET IN QUEUE.

/* IF Symbol = 0 THEN */

IF REDUCEINSETS = 0 THEN REDUCEINSETS = SET(J); ELSE DO:

IF REDUCEINSETS > 0 THEN REDUCEINSETS = REDUCEINSETS + 1;

UNDO;

IF TOP > MULT_REDUCE_SET(J) THEN GO TO ERR;

MULT_REDUCE_SET(J) = SET(J);

END;

ELSE:

TOP = TOP + 2;

IF TOP > MULT_REDUCE_SET(J) THEN GO TO ERR;

MULT_REDUCE_SET(J) = REDUCEINSETS;

REDUCEINSETS = 2;

END;

MARKER(J) = 1;

GO TO PROCLOSED;

END.

/* NO CLOSURE FOR TERMINAL SYMBOLS. */

IF MAP(SYMBOL) = NGN THEN GO TO PROCLOSED;

PLACE = PLACE + 1;

PROCLOSED:

END;

SLIM(SET(J)) = SET_LIM;

EXPAND:

/* FIND FIRST * IN MARKER (PARALLEL TO EXISTING SETS) AND DETERMINE VIA BINARY SEARCH WHICH SET IT BELONGS TO.*/

DO CANDIDATE = CANDIDATE + 1 TO CONFIG_SET_LIM WHILE (MARKER(CANDIDATE) > 0)

END;

IF CANDIDATE > SLIM_SET THEN GO TO LKQ_FINISH;

SET(J) = CANDIDATE;

END;

GO TO CKLS;

IF UNC L THEN:

DELETE(J);

GO TO EXIT_BINARY_SEARCH;

END;

PENCE(J) = 0;

GO TO EXIT_BINARY_SEARCH;

END;

PENCE(J) = 1;

GO TO EXIT_BINARY_SEARCH;

END;

PENCE(J) = CANDIDATE - SLIM(J);

IF CANDIDATE > SLIM(J) THEN GO TO EXIT_BINARY_SEARCH;

IF CANDIDATE = SLIM(J) THEN GO TO EXIT_BINARY_SEARCH;

ELSE IF PENCE(J) = 0 THEN GO TO CKLS;

END;

END OF BINARY SEARCH - AT THIS POINT FENCE IS THE SET THE CANDIDATE FOR EXPANSION (CANDIDATE) IS IN.

SELECT ALL ENTRIES OF THIS SET WITH THE SAME SYMBOL TO RIGHT OF INPUT ENTER ELEMENTS IN TAY AND OUT POSITIONS +/- IN TRYOUT;

EXIT_BINARY_SEARCH;

TRYKN(J) = 1;

TRYKN(J) = 0;

MARKER(J) = 1;

DO CANDIDATE = 1 TO CONFIG_SET_LIM WHILE (MARKER(CANDIDATE) = 1)

END;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

/* TURN LEFT */

OUT_SWITCH = SET(J) + 1;

IF SLIM(J) > J THEN GO TO EXIT_BINARY_SEARCH;

IF J = 1 THEN GO TO EXIT_BINARY_SEARCH;

END;

END;

END.

END OF BINARY SEARCH - AT THIS POINT FENCE IS THE SET THE CANDIDATE FOR EXPANSION (CANDIDATE) IS IN.

SELECT ALL ENTRIES OF THIS SET WITH THE SAME SYMBOL TO RIGHT OF INPUT ENTER ELEMENTS IN TAY AND OUT POSITIONS +/- IN TRYOUT;

EXIT_BINARY_SEARCH;

TRYKN(J) = 1;

TRYKN(J) = 0;

MARKER(J) = 1;

DO CANDIDATE = 1 TO CONFIG_SET_LIM WHILE (MARKER(J) = 0)

END;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

/* TURN LEFT */

OUT_SWITCH = SET(J) + 1;

IF SLIM(J) > J THEN GO TO EXIT_BINARY_SEARCH;

IF J = 1 THEN GO TO EXIT_BINARY_SEARCH;

END;

END;

END.

END OF BINARY SEARCH - AT THIS POINT FENCE IS THE SET THE CANDIDATE FOR EXPANSION (CANDIDATE) IS IN.

SELECT ALL ENTRIES OF THIS SET WITH THE SAME SYMBOL TO RIGHT OF INPUT ENTER ELEMENTS IN TAY AND OUT POSITIONS +/- IN TRYOUT;

EXIT_BINARY_SEARCH;

TRYKN(J) = 1;

TRYKN(J) = 0;

MARKER(J) = 1;

DO CANDIDATE = 1 TO CONFIG_SET_LIM WHILE (MARKER(J) = 0)

END;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

END;

END.

END OF BINARY SEARCH - AT THIS POINT FENCE IS THE SET THE CANDIDATE FOR EXPANSION (CANDIDATE) IS IN.

SELECT ALL ENTRIES OF THIS SET WITH THE SAME SYMBOL TO RIGHT OF INPUT ENTER ELEMENTS IN TAY AND OUT POSITIONS +/- IN TRYOUT;

EXIT_BINARY_SEARCH;

TRYKN(J) = 1;

TRYKN(J) = 0;

MARKER(J) = 1;

DO CANDIDATE = 1 TO CONFIG_SET_LIM WHILE (MARKER(J) = 0)

END;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

END;

END.

END OF BINARY SEARCH - AT THIS POINT FENCE IS THE SET THE CANDIDATE FOR EXPANSION (CANDIDATE) IS IN.

SELECT ALL ENTRIES OF THIS SET WITH THE SAME SYMBOL TO RIGHT OF INPUT ENTER ELEMENTS IN TAY AND OUT POSITIONS +/- IN TRYOUT;

EXIT_BINARY_SEARCH;

TRYKN(J) = 1;

TRYKN(J) = 0;

MARKER(J) = 1;

DO CANDIDATE = 1 TO CONFIG_SET_LIM WHILE (MARKER(J) = 0)

END;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

IF MARKER(J) = 0 THEN GO TO NOTSAME;

END;

END.
T,n
IN
IL: NOW SEE SAME:
SET TRANS Continue TO USE LIMIJ-1.
TAaLE(FENCE,MAPTO(SYMBOL I=J;
TRYK~T; TRYDuT(K)
UN( MAPTOISYMBOL I=NSjS;
NEW_SET
NU_SETS= NSETS;
MASTER_ERROR BIT
SIZE SNAP SIGNAL
FINAL GENERATE SECTION...*SIP,A1;
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LKO GENERATE:
END LANG GENERATE:
*/
heap=FILE OUTPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DECLARE
LANG=FILE INPUT STREAM.
END LANG GENERATE:
SLRl GENERATE:
BEGIN:
DO J=2 TO NO_Syms;
  IF TABLE(J,J)=0 THEN DO ELSE_TABLE(J,J)=DIE;
END;
EO;
/*
NOW PROCESS ALL REDUCE STATES, THAT IS, FOR ALL STATES REQUIRING
A REDUCTION ENTER THE APPROPRIATE NEGATIVE PRODUCTION NUMBER
IN THE TERMINAL SYMBOL COLUMNS FOR TERMINALS IN FOLLOW(state,*). */
DO I=2 TO NO_SETS;
  IF REDUCE(I)=0 THEN GO TO SKIP_REDUCE;
  IF REDUCE(I)<0 THEN DC=REDUCE(I)+1 TO NO_SYMS;
  IF FOLLOW(MAP_TABLE(PROD,REDUCE(I),1),1) THEN DO;
    IF TABLE(I,J)<0 THEN DO;
      PUT FILE PRINT SKIP EDIT
      (*STATE *,I*) IS INADEQUATE AND THE SIMPLE *
      *TRANSITION IS UNDER *.NOylie(MAP_PRINT);*
      *TRYING TO REPLACE *TABLE(I,J) WITH *
      *REDUCE(I)*ASkip*,A AND,CNAMS);*
      MASTER_ERROR*418;
      ELSE END:
    ELSE ELSE_TABLE(I,J)=REDUCE(I);*
    END;
  END;
/* NULL than 1 REDUCTION FOR THIS SET, */
ELSE
END PROCEDURE

DECLARE M (*, *) BIT EID ALIGNED

END FILE

END LINEAR STRUCTURE

NEW IS

END

END FILE

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END

END LINEAR STRUCTURE

NEW IS

END
(* UNSTACKTOP1,STACKTP1,STACKTP2,STACKTOP *)

/* FIXED BINARY */

/* STACK AND STACKS ARE PUSH CLSN STACK VECTORS */

/* SEARCH FOR THE NODE WHICH WILL BE THE FATHER OF THE ONE TO BE INSERTED. TRAVERSE THE PATH FOR LATER USE */

/* UNSTACKTOP1,STACKTOP1,STACK,STACKTOP1 */)
APPENDIX D

LOGIC BLOCK DIAGRAM
START

READ INPUT PARAMETERS FROM FILE PARMIN

OUTPUT HEADING

SET DEFAULT PARAMETERS IF NECESSARY

BEGIN INPUT SECTION

80 BYTE RECORD FROM FILE PRODIN

END INPUT

SET BYTES #73-80 TO NON-BLANKS

SPIN THROUGH CONSECUTIVE BLANKS

NEXTSYM

NAME ← CONSECUTIVE NON-BLANKS

INSERT (SEARCH) NAME IN SYMBOL TABLE

PUNCTUATION

COMMA SEMI-COLON COLON PERIOD

FILL NEXT COLUMN OF PRESENT ROW OF PROD

NEXTSYM
COPY COL 1 OF THIS ROW TO COL 1 OF NEXT ROW; ENTER NEXT COLUMN OF THIS ROW; RESET COLUMN POINTER TO 1 AND INCREMENT ROW POINTER.

ENTER NEXT COLUMN. NOTE: COULD Optionally BRANCH TO READ NEXT RECORD IF IT IS KNOWN THAT, IF A RECORD CONTAINS A PERIOD, THEN IT IS THE LAST SYMBOL.

RESET COLUMN POINTER TO 1; INCREMENT ROW POINTER AND ENTER.

SET THE MAPPING VECTORS: COUNT THE NON-TERMINAL; ENDEx (COUNT) ← ROW POINTER; MAPTO (SYMBOL TABLE POSITION) ← COUNT; MAPFROM (COUNT) ← SYMBoL TABLE POSITION.

FIXUP LOOP TO SET MAPTO AND MAPFROM FOR TERMINAL SYMBOLS.

OUTPUT STATISTICS ON PRODUCTIONS.

BEGIN DEBUG SECTION.

FORM "WITHIN" RELATION, THEN TRANSITIVE CLOSURE.

ANY '0' IN 2nd ROW EXCEPT FOR FIRST TWO COLUMNS MEANS CORRESPONDING SYMBOL NOT "WITHIN" - OUTPUT DIAGNOSTIC IF ANY.
DETECTION OF USELESS PRODUCTIONS
C.F. REFERENCE - OUTPUT DIAGNOSTIC IF ANY

DETECTION OF DUPLICATE RIGHT-HAND-SIDES - OUTPUT DIAGNOSTIC IF ANY

BEGIN CONFIGURATION SET COMPUTATION

INITIALIZE FIRST SET TO FIRST PRODUCTION WITH DOT TO THE RIGHT (FINAL STATE), SECOND SET TO FIRST PRODUCTION WITH DOT TO THE LEFT (INITIAL STATE)

CLOSE

GET NEXT ITEM OF SET BEING CLOSED

YES BASIS ENTRY

NO

SYMBOL ← PROD (ITEM, 2)

DOT TO RIGHT

NO

YES

SYMBOL ← PROD (ITEM, 2)

ITEM IS A REDUCTION ENTRY, ENTER THIS ITEM IN REDUCE (SET NUMBER) IF EMPTY - ELSE SET TO NUMBER OF ENTRIES AND PUT ITEM IN QUEUE

SET MARKER (ITEM NUMBER) TO 1

AT PRODCLOSED BRANCH TO CLOSE IF ALL ITEMS NOT PROCESSED ELSE BRANCH TO EXPAND

PRODCLOSED

h

SYMBOL ← 0

SYMBOL ← PROD (ITEM, DOT POSITION)

SYMBOL = 0

NO

YES
YES

SYMBOL A TERMINAL

ENTER ALL PRODUCTIONS WITH SYMBOL AS A LHS IN THIS SET WITH DOT TO LEFT PROVIDING DUPLICATION OF PREVIOUS SET ENTRIES AVOIDED

PRODCLOSE

SET SLIM (SET NUMBER) TO LATEST ENTERED ITEM'S POSITION

EXPANSION

ALL MARKERS SET TO 1

YES

LRO-FINIS

NO

GET SET NUMBER CONTAINING AN ITEM WHOSE MARKER IS NOT SET TO 1

BUFFER UP THIS ITEM AND ALL OTHER ENTRIES OF THIS SET THAT HAVE A COMMON SYMBOL TO RIGHT OF DOT, SET MARKER FOR EACH

FOR ALL BASIS SETS WITH THE SAME NUMBER OF ENTRIES, CHECK BUFFER AGAINST SUCH SETS TO DETERMINE DUPLICATION (BOTH ITEMS AND DOT POSITIONS MATCHED)

NO

DUPLICATE

YES

TABLE (SET, SYMBOL) = FOUND DUPLICATE SET NUMBER

ENTER BUFFERED ITEMS AS THE BASIS SET OF A NEW SET, ENTER DOT POSITIONS + 1 INTO DOT POSITION ARRAY

TABLE (SET, SYMBOL) = NEW SET NUMBER

CLOSE

LRO_FINIS

OUTPUT CONFIGURATION SETS

SLR(1) TABLE GENERATION (TRANSITION ENTRIES HAVE BEEN MADE)
Compute "inverse" reflexive transitive closure of tail symbol matrix for non-terminals.

Compute follow matrix per algorithm in thesis.

Fill in reduction entries.

The following logic is applied to each row of the table:

\[
\text{REDUCE}(i) > 0 \rightarrow \text{REDUCE}(i) \leq 0
\]

For all terminal symbol columns corresponding to symbols in follow of the LHS of the indicated reduction, enter -REDUCE(i) in those columns providing a previous entry has not been made in that table position. If so, then state is unsolvably inadequate - set master error switch.

Reduce(i) holds number of elements in queue to process as reductions, discard each after processing.

After every row processed, do the following:

Count and list inadequate states by detecting two different transitions or a reduction and a transition in the same state - rows of table are processed left to right and first inadequate condition ends processing of that state.

Output SLR(1) table and other data.

END.
VITA

Joseph Lee Gray

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF A SLR(1) PARSING ALGORITHM

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Poplar Bluff, Missouri, April 24, 1944, the son of Mr. and Mrs. Howard Gray.

Education: Graduated from Poplar Bluff High School, Poplar Bluff, Missouri, in May, 1962; received Bachelor of Arts degree from California State University at Long Beach, Long Beach, California, in January, 1971, with a major in Mathematics; completed requirements for the Master of Science degree at Oklahoma State University in May, 1973.

Professional Experience: Graduate assistant, Oklahoma State University, Computing and Information Sciences Department, Stillwater, Oklahoma, August, 1971, to December, 1972; computer repairman and instructor, United States Army, May, 1966, to May, 1969.