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:

James R. Van Doren
Thesis Adviser

G. E. Hedrick

D D Fisher

N Durham
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

LIST OF TABLES

LIST OF FIGURES

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 concensus 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 <u>rules</u> (also called <u>rewriting rules</u> or <u>productions</u>) of the form U_i: u_i. where U_i is a metasymbol and $u_i \in V^*$. 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_i is called the goal symbol (also distinguished symbol). This definition is overrestrictive but serves the purpose of this thesis. U_i 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 (V_T, V_N, P, S) where V_T is the set of terminal symbols, 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, $V_T = \{?, +, **, (,), i\}$, $V_N = \{S, E, T, P\}$, 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:

PRESENT STRING	APPLIED RULE
(1) S	
(2) ?E?	1
(3) ?E+T?	2
(4) ?T+T?	3
(5) ?P+T?	5
(6) ?i+T?	7
(7) ?i+P?	5
(8) ?i+i?	7

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

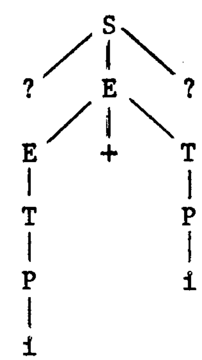


Figure 1. Syntax Tree for $S \to^* ?i+i?$

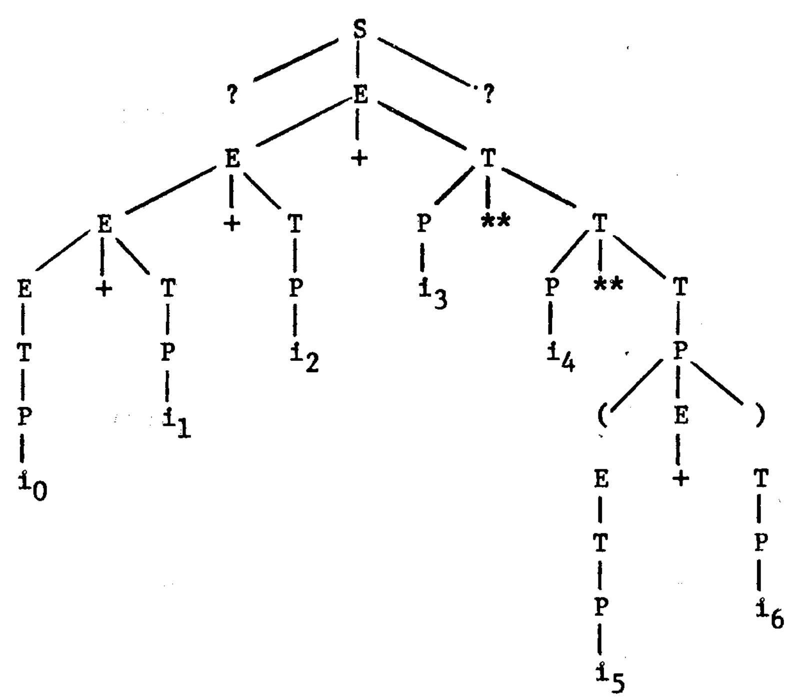


Figure 2. Syntax Tree for $S \to^* ?i_0+i_1+i_2+i_3**i_4**(i_5+i_6)?$

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. G_1 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 G_1 and syntax trees for different sentences of $L(G_1)$ 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 <u>ambiguous</u> 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 ambiguous.

Parsing

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 G_1, 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 <u>recognizing</u> or <u>parsing</u>) elements of a language. The first, and possibly easiest to understand, is the <u>top-down method</u>. 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 nonterminal symbol. Essentially what is done is to "draw" the syntax tree from top to bottom (root to leaves). In parsing the sentence ?i+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 +'s. Therefore, a T must be found; but a T is one or more P's separated by **'s; 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 $L(G_1)$. 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 +'s; 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 <u>bottom-up method</u>. 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 G_1, 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:

(0) 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.

(1) Find the handle of s_i.

(2) Replace the handle of s_i by the name of its father in the syntax tree.

(3) Prune the handle from the tree.

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

	PRESENT STRING	HANDLE	STRING AFTER STEP 2
(1)	?i+i?	i	?P+i?
(2)	?P+i?	P	?T+i?
(3)	?T+i?	T	?E+i?
(4)	?E+i?	i	?E+P?
(5)	?E+P?	P	?E+T?
(6)	?E+T?	E+T	?E?
(7)	?E?	?E?	S
(8)	S		

If the steps in the "present string" column are followed backwards, the derivation $S \rightarrow^* ?i+i?$ 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 $P \in V^*$; 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 $A = B$, $A \subset B$, $B \subset A$, $A \cap B \neq \emptyset$ 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 R b, and a corresponds to row i, and b corresponds to column j, then the ij^{th} entry is 1. If a $\not R$ 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 R b. That is, for $a \in A$, $b \in B$, and a 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 R a for every $a \in A$ and every $a \in B$

Symmetry. a R b if and only if b R a

Transitivity. a R b $\wedge$ b R c if and only if a 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 $\wedge$ j = k if and only if i = k

The relation, H, defined on V of G_1 by H = {(A,b) | $A \in V_N$, $b \in V$, $C \in V^*$, 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 convenient 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 $E\ H\ T \wedge T\ H\ P \Longrightarrow E\ H\ P$, and $E\ H\ P \wedge P\ H\ (\Longrightarrow E\ H\ ($. 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.

	S	E	T	P	?	+	**	(	)	i
S	0	0	0	0	1	0	0	0	0	0
E	0	1	1	0	0	0	0	0	0	0
T	0	0	0	1	0	0	0	0	0	0
P	0	0	0	0	0	0	0	1	0	1

Figure 3. Boolean Matrix Representation of $H(G_1)$

?	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	0	0	0	0	0	0
**	0	0	0	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0	0	0	0
)	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0

Figure 3. (Continued)

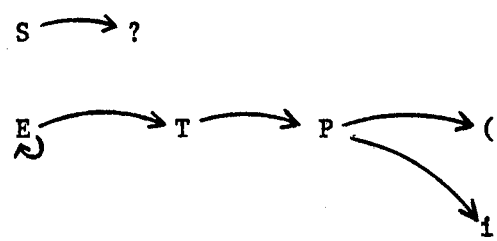


Figure 4. Graph Representation of $H(G_1)$

The product of two relations, say R on $A \times B$ and P on $C \times D$, is defined by $a\ RP\ d$ if and only if there exists an $e \in B \cap C$ such that $c\ R\ e \wedge e\ P\ d$ is true. If P is a product relation, say QT, such that $e\ QT\ d$ so that there does exist an f such that $e\ Q\ f \wedge f\ T\ d$ is true, then, for the relation RP, which is actually PQT, it is true that $c\ R\ e \wedge e\ Q\ f \wedge f\ T\ d$. But $\wedge$ 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 $R^n = RR^{n-1}$ where $n>0$ and $R^1 = R$ and the *transitive closure* of R by $a\ R^+\ b$ if and only if there exists a c such that $a\ R^n\ c$ for some $n>0$. If the *identity relation* is denoted by R^0, that is, $a\ R^0\ b$ if and only if $a = b$, then the *reflexive transitive closure*, R^*, can be defined as $a\ R^*\ b$ if and only if $a\ R^n\ b$ for $n\geq0$. 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 $H^+(A) = \{b \in V \mid A \rightarrow^* bC$ where $C \in V^*\}$. $H^+(G_1)$ can be represented by the Boolean matrix in Figure 5.

	S	E	T	P	?	+	**	(	)	i
S	0	0	0	0	1	0	0	0	0	0
E	0	1	1	1	0	0	0	1	0	1
T	0	0	0	1	0	0	0	1	0	1
P	0	0	0	0	0	0	0	1	0	1
?	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	0	0	0	0	0	0
**	0	0	0	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0	0	0	0

Figure 5. Boolean Matrix Representation of $H^+(G_1)$

)	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0

Figure 5. (Continued)

Translating Figure 5 into a graph, Figure 6 results:

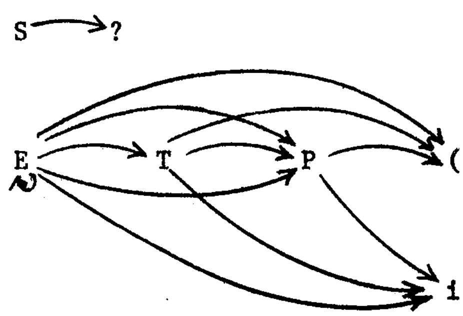


Figure 6. Graph Representation of $H^+(G_1)$

$H^*(G_1)$, 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 ijth element of C is defined by

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

but, for Boolean matrix multiplication, the ijth element of C is defined by

$$c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \ldots \vee (a_{in} \wedge b_{nj})$$

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 $H^1 \cup H^2$ 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 $T \to i$. Likewise, for higher powers, $H^1 \cup H^2 \ldots \cup H^i$ in effect connects arcs of length 1, 2, ..., 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 $A \times B$ whose rows correspond to the elements of A and whose columns correspond to elements of B. It is necessary that $A \subseteq B$ and that, if row i corresponds to $x \in B$. (An example of such an M is the first four rows and all columns of Figure 3.) The PL/1 program segment follows.

```
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;
```

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:., 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 <u>right sentential form</u>, that is, a is a rightmost derivation of the goal symbol, then $FIRST^k$ (a) is defined to be the first k terminal symbols derivable from a. That is, $FIRST^k$ (a) = $\{w \in V_T^* \mid a \rightarrow^* wx,\ x \in V_T^*$ and either w is k symbols long or w is less than k symbols long and $x = \emptyset\}$. If $a \in V_T^*$, then $FIRST^k$ (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 $FIRST^k$ (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 <u>stack</u>, <u>error</u>, or

reduce i, where i is a production number.

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

The process ends when the final state (a particular row of the table) is entered. The problem of entering the final state with unexpended suffix does not exist since special delimiters are placed before and after the text to be processed. Also, there is a 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 $(A) \subset H^{+}(A)$, 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 $(A \rightarrow a.b,u)$ where $A \rightarrow ab$ 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.

CLOSURE - A set begins with items specified by expansion. The first set begins with $(S \rightarrow .?E?, \emptyset)$. If $(A \rightarrow a.Bc,u)$ is in the set, then $(B \rightarrow .d,v)$ is added to the set for productions B: d. for any $d \in V^*$ and $v \in$ FIRST (cu). Here, $a,c \in V^*$ and $B \in V_N$. 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 ($B \rightarrow aX.b,u$) and ($B \rightarrow a.Xb,u$) 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.

($B \rightarrow a.b,c_1$), ..., ($B \rightarrow a.b,c_m$) is denoted by ($B \rightarrow a.b,c_1/c_2/.../c_m$). 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

SET NAME	NO.	ITEMS	NOTES
A_0	1.	S→.E,∅	initial set
	2.	E→.AA,∅	
	3.	A→.aA,a/b	a,b∈H^+(A)
	4.	A→.b,a/b	a,b∈H^+(A)
A_1	1.	S→E.,∅	from A_0.1
A_2	1.	E→A.A,∅	from A_0.2
	2.	A→.aA,∅	
	3.	A→.b,∅	
A_3	1.	A→a.A,a/b	from A_0.3
	2.	A→.aA,a/b	
	3.	A→.b,a/b	
A_4	1.	A→b.,a/b	from A_0.4
A_5	1.	E→AA.,∅	from A_2.1
A_6	1.	A→a.A,∅	from A_2.2
	2.	A→.aA,∅	
	3.	A→.b,∅	
A_7	1.	A→b.,∅	from A_2.3
A_8	1.	A→aA.,a/b	from A_3.1
A_9	1.	A→aA.,∅	from A_6.1

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

SET NAME	NO.	ITEMS	NOTES
A_0	1.	S→.?E?,∅	
A_1	1.	S→?.E?,∅	from A_0.1
	2.	E→.aAb,?	
	3.	E→.aBc,?	
	4.	E→.dAc,?	
	5.	E→.dBb,?	
A_2	1.	S→?E.?,∅	from A_1.1
A_3	1.	E→a.Ab,?	from A_1.2
	2.	E→a.Bc,?	from A_1.3
	3.	A→.fA,b	
	4.	A→.f,b	
	5.	B→.fB,c	
	6.	B→.f,c	
A_4	1.	E→d.Ac,?	from A_1.4
	2.	E→d.Bb,?	from A_1.5
	3.	A→.fA,c	
	4.	A→.f,c	
	5.	B→.fB,b	
	6.	B→.f,b	
A_5	1.	S→?E?.,∅	"final" set from A_2.1
A_6	1.	E→aA.b,?	from A_3.1
A_7	1.	E→aB.c,?	from A_3.2
A_8	1.	A→f.A,b	from A_3.3
	2.	A→f.,b	from A_3.4
	3.	B→f.B,c	from A_3.5
	4.	B→f.,c	from A_3.6
	5.	A→.fA,b	
	6.	A→.f,b	
	7.	B→.fB,c	
	8.	B→.f,c	
A_9	1.	E→dA.c,?	from A_4.1
A_{10}	1.	E→dB.b,?	from A_4.2
A_{11}	1.	A→f.A,c	A_{11} is not a duplicate of A_8

TABLE II (Continued)

SET NAME	NO.	ITEMS	NOTES
	2.	A→f.,c	
	3.	B→f.B,b	
	4.	B→f.,b	
	5.	A→.fA,c	
	6.	A→.f,c	
	7.	B→.fB,b	
	8.	B→.f,b	
A_{12}	1.	E→aAb.,?	from A_6.1
A_{13}	1.	E→aBc.,?	from A_7.1
A_{14}	1.	A→fA.,b	from A_8.1
A_{15}	1.	B→fB.,c	from A_8.3
A_{16}	1.	E→dAc.,?	from A_9.1
A_{17}	1.	E→dBb.,?	from A_{10}.1
A_{18}	1.	A→fA.,c	from A_{11}.1
A_{19}	1.	B→fB.,b	from A_{11}.3

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 G_3. 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 $v \in$ FIRST (bu), that is, for all terminal symbols that can legitimately follow a in this state.

(3) If (S→?B?,∅) is in A, then p (∅) = accept.

(4) p (u) = 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 catenated with the unexpended suffix at any step yield a right sentential form. Working from bottom to top, this results in S→?E?→?aBc?→?afBc?→?affc?, which is indeed the rightmost derivation sequence for ?affc?.

TABLE III

LR(1) TABLE FOR G_3

	p							g										
STATE	?	a	b	c	d	f	Ø	S	E	A	B	?	a	b	c	d	f
0	S											1					
1		S			S				2				3			4	
2	S											5					
3						S				6	7						8
4						S				9	10						11
5							A										
6			S											12			
7				S											13		
8			7	9		S				14	15						8
9				S											16		
10			S											17			
11			9	7		S				18	19						11
12	2																
13	3																
14			6														
15				8													
16	4																
17	5																
18				6													
19			8														

STACK	UNEXPENDED SUFFIX	ACTION
0	?affc?	initial condition, read ?
0?1	affc?	read a
0?1a3	ffc?	read f
0?1a3f8	fc?	read f
0?1a3f8f8	c?	reduce B: f.
0?1a3f8B15	c?	reduce B: f, B.
0?1a3B7	c?	read c
0?1a3B7c13	?	reduce E: a, B, c.
0?1E2	?	read ?
0?1E2?5	Ø	accept

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 $F(A) = \{a \mid S \to^{*} bAc$ and $a =$ FIRST (c) where $A \in V_N$, $a \in V_T$, $b \in V^*$, 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 $(A \to b.,\emptyset)$, $b \in V^*$, in each set by $(A \to b.,a)$ 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 ∅ column is completely eliminated since the only possible entries are reduction entries and accept; however, there are no reduction entries in the ∅ 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

STATE	E	A	B	?	a	b	c	d	f
0				1					
1	2				3			4	
2				5					
3		6	7						8
4		9	10						11
6						12			
7							13		
8		14	15			-7	-9		8
9							16		
10						17			
11		18	19			-9	-7		11
12				-2					
13				-3					
14						-6			
15							-8		
16				-4					
17				-5					
18							-6		
19						-8			

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

SET NO.	ITEMS	NOTES
1.	S→?E?.	final state
2.	S→.?E?	initial state
3.	S→?.E?	from 2
	E→.E+T	closure entries for
	E→.T	the single basis
	T→.P**T	entry; closure
	T→.P	ceases when dot is
	P→.i	left of terminal
	P→.(E)	symbols
4.	S→?E.?	from 3; expansion gives final state
	E→E.+T	from 3
5.	E→T.	from 3 or 8; no expansion here
6.	T→P.**T	from 3 or 8
	T→P.	
7.	P→i.	from 3 or 8
8.	P→(.E)	from 3 or 8
	E→.E+T	indirect recursion
	E→.T	lengthens the set of
	T→.P**T	configuration sets
	T→.P	
	P→.i	
	P→.(E)	
9.	E→E+.T	from 4
	T→.P**T	
	T→.P	
	P→.i	
	P→.(E)	
10.	T→P**.T	from 6
	T→.P**T	
	P→.i	
	P→.(E)	
11.	P→(E.)	from 8

TABLE V (Continued)

SET NO.	ITEMS	NOTES
	E→E.+T	from 8
12.	E→E+T.	from 9
13.	T→P**T.	from 10
14.	P→(E).	from 11

TABLE VI

SLR(1) TABLE FOR G_1

STATE	S	E	T	P	?	+	**	i	(	)
2					3					
3		4	5	6				7	8	
4					1	9				
5					-3	-3				-3
6					-5	-5	10			-5
7					-6	-6	-6			-6
8		11	5	6				7	8	
9			12	6				7	8	
10			13	6				7	8	
11										14
12						-2	-2			-2
13						-4	-4			-4
14						-7	-7			

It is now shown how to understand at least part of the structure of $L(G_1)$ 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.

STACK	UNEXPENDED SUFFIX	NOTES
2	?i+i+i?	initial 3=T(2,?)
2?3	i+i+i?	7=T(3,i)
2?3i7	+i+i?	-6=T(7,+) and 6=T(3,P)
2?3P6	+i+i?	-5=T(6,+) and 5=T(3,T)
2?3T5	+i+i?	-3=T(5,+) and 4=T(3,E)
2?3E4	+i+i?	9=T(4,+)
2?3E4+9	i+i?	7=T(9,i)
2?3E4+9i7	+i?	-6=T(7,+) and 6=T(9,P)
2?3E4+9P6	+i?	-5=T(6,+) and 12=T(9,T)
2?3E4+9T12	+i?	-2=T(12,+) and 4=T(3,E)
2?3E4	+i?	9=T(4,+)
2?3E4+9	i?	7=T(9,i)
2?3E4+9i7	?	-6=T(7,?) and 6=T(9,P)
2?3E4+9P6	?	-5=T(6,?) and 12=T(9,T)
2?3E4+9T12	?	-2=T(12,?) and 4=T(3,E)
2?3E4	?	1=T(4,?)
2?3E4?1	∅	final state-accept

Figure 8. Parsing ?i+i+i? Using Table VI

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

G_1, 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 G_1, 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 $T \rightarrow P.{**}T$ 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 $T \rightarrow P.{**}T$ 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, G_1 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 $F(A) = \{b \mid$ there exists a production $C: a,A,B,c$ where $c,a \in V^*$, $A \in V_N$, $B \in V_T$ and $b = B$ or $B \in V_N$ and $b \in V_T$ and $b \in H^{+}(B)\}$. 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 $A \epsilon V_N$ and the "true" columns representing the elements of FOLLOW (A). For an operator grammar (6), $H^+(G)$ is not needed since every $A \epsilon V_N$ 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 $T(A) = \{B \epsilon V_N \mid B \rightarrow^* aA \text{ where } A \epsilon V_N, a \epsilon V^*\}$. 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 $s \epsilon V_T$ such that there is a transition under s from N.

(4) The resulting set is FOLLOW.

Since FOLLOW is computed for every $A \epsilon V_N$ 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 $A \epsilon V_N$; 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 $(C_1,C_2) \leftarrow$ "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 $A \epsilon V_N$, which does not occur unless it is assumed the production A: P. is present during construction of T* for some $A \epsilon V_N$. But it is also true that the P^{th} row must have a "true" in it, that is, P must have an $A \epsilon V_N$ 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 $A \epsilon V_N$, 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? 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 G_3, which was shown to be LR(1), but is not SLR(1), follows. (In fact, it is not SLR(k) for any k.)

TABLE VII

SLR(1) CONFIGURATION SETS FOR G_3

SET NO.	ITEMS
1	S→?E?.
2	S→.?E?
3	S→?.E?
	E→.aAb
	E→.aBc
	E→.dAc
	E→.dBb
4	S→?E.?
5	E→a.Ab
	E→a.Bc
	A→.fA
	A→.f
	B→.fB
	B→.f

TABLE VII (Continued)

SET NO.	ITEMS
6	$E\to d.Ac$
	$E\to d.Bb$
	$A\to .fA$
	$A\to .f$
	$B\to .fB$
	$B\to .f$
7	$E\to aA.b$
8	$E\to aB.c$
9	$A\to f.A$
	$A\to f.$
	$B\to f.B$
	$B\to f.$
	$A\to .fA$
	$A\to .f$
	$B\to .fB$
	$B\to .f$
10	$E\to dA.c$
11	$E\to dB.b$
12	$E\to aAb.$
13	$E\to aBc.$
14	$A\to fA.$
15	$B\to fB.$
16	$E\to dAc.$
17	$E\to dBb.$

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 VIII

SLR(1) CONFIGURATION SETS FOR G_4

SET NO.	ITEMS
1	S→?G?.
2	S→.?G?
3	S→?.G?
	G→.A
	G→.CB
	G→.Abc
	A→.a
	c→.A
4	S→?G.?
5	G→A.
	G→A.bc
	C→A.
7	A→a.
8	G→Ab.c
9	G→cB.
10	B→b.
11	G→Abc.

TABLE IX

SLR(1) TABLE FOR G_4

STATE	G	A	B	C	?	b	c	a
2					3			
3	4	5		6				7
4					1			
5					-2	-7/8		
6			9			10		
7					-5	-5		
8							11	
9					-3			
10					-6			
11					-4			

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 $L(G_4)$ is {?a?, ?ab?, ?abc?}. Figure 9 shows an attempted parse of ?abc?.

STACK	UNEXPENDED SUFFIX	NOTES
2	?abc?	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?3A5b8c11	?	11=T(8,c)
2?3G4	?	-4=T(11,?) 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

(1) Aho, A. and J. Ullman. "The Care and Feeding of LR(k) Grammars." Proceedings of the Third Annual ACM Symposium on Theory of Computing, (May, 1971), 159-170.

(2) Aho, A. and J. Ullman. "A Technique for Speeding Up LR(k) Parsers." Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, (May, 1972). (to be published)

(3) DeRemer, F. L. "Practical Translation for LR(k) Languages." Ph.D. thesis, MIT, Cambridge, Massachusetts, (September, 1969).

(4) DeRemer, F. L. "Simple LR(k) Grammars." Communications of the ACM, 14, 7, (July, 1971), 453-460.

(5) DeRemer, F. L. "Simple LR(k) Grammars - Definition and Implementation." Computer Evolution Report, 2, 4, (September, 1970), University of California, Santa Cruz.

(6) Feldman, J. and D. Gries. "Translator Writing Systems." Communications of the ACM, 11, 2, (February, 1968), 77-91.

(7) Gries, David. Compiler Construction for Digital Computers, New York: John Wiley & Sons, 1971.

(8) Hopcroft, J. and J. Ullman. Formal Languages and their Relation to Automata, New York: Addison-Wesley, 1969.

(9) Knuth, D. E. The Art of Computer Programming, Reading: Addison-Wesley, 1969.

(10) Knuth, D. E. "On the Translation of Languages from Left to Right." Information and Control, 8, (October, 1965), 607-639.

(11) McKeeman, W., J. Horning, and D. Wortman. A Compiler Generator, Englewood Cliffs: Prentice-Hall, Inc., 1970.

(12) Van Doren, J. and J. Gray. "An Algorithm for Maintaining Dynamic AVL Trees." Proceedings of the Fourth International Symposium on Computing and Information Sciences, (1972). (to be published)

(13) van Wijngaarden, A. (ed.), B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. Report on the Algorithmic Language ALGOL 68, Offprint from Numerishe Mathematic 14: 79-218. Berlin: Springer-Verlag, 1969.

(14) Warshall, S. "A Theorem on Boolean Matrices." Journal of the ACM, 9, (January, 1962), 11-12.

APPENDIXES

APPENDIX A

LIST OF SYMBOLS

APPENDIX A

LIST OF SYMBOLS

SYMBOL	MEANING	PAGE OF FIRST OCCURRENCE
CF	context-free	1
TWS	translator writing systems	2
V	vocabulary of a grammar	4
V^*	all strings of elements of V	4
,	is followed by	5
;	exclusive "or"	5
:	may be rewritten as	5
.	delimiter	5
ϵ	set inclusion	5
LHS	left hand side	5
RHS	right hand side	5
V_T	the terminal symbols of V	5
V_N	the non-terminal symbols of V	5
{ }	set delimiters	6
$\rightarrow$	a direct derivation	6
$\rightarrow^*$	a derivation (closure of $\rightarrow$)	6
DPDA	deterministic push down automata	14
$A \times B$	the Cartesian product of A and B	15
$\subset$	is a subset of	15
$\cap$	intersection	15
a R b	a is related to b	15
$\wedge$	logical and	16
$\Rightarrow$	implies	17
$\cup$	union	18
$\vee$	logical or	20
$\sum$	summation	20

APPENDIX B

USER'S GUIDE

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.

PARAMETER NUMBER	DESCRIPTION
1	number of productions
2	maximum number of symbols in any production
3	maximum number of characters in any symbol or at least ≥ number of characters to make every symbol unique
4	maximum number of unique symbols in the grammar
5	number of items in all configuration sets combined
6	number of configuration sets
7	maximum number of basis entries for any configuration set
8	= 1 to activate the DEBUG facility
9	= 1 to count and list solvable inadequate states

PARAMETER NUMBER	DESCRIPTION
10	= 1 for full printed output
11	= 1 for punched output in a form to be read by the parsing routine

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 parameter 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 PL1LFCG
//PL1L.SYSIN DD *
  --SOURCE DECK--
//GO.PARMIN DD *
  --PARAMETER CARDS--
//GO.PRODIN DD *
  --PRODUCTIONS--
//GO.PRINT DD SYSOUT=A
```

```
//GO.PUNCH DD SYSOUT=B,DCB=BLKSIZE=80
//
```

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
//STEPLIB 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.

GRAMMAR

	G_1	ALGOL 60	ALGOL 68 (subset)	BASIC (simple precedence form)	BASIC
Number of productions	7	181	159	99	85
Number of parts	4	6	6	5	9
Number of symbols	10	141	99	102	89
Number of characters	4	31	10	10	10
Number of sets	15	304	310	174	148
Combined length of sets	50	2191	5592	957	816
Number of basis entries	3	5	10	3	3
Reduction queue	0	22	0	0	0

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

```
/* TITLE: SLR(1) PARSING TABLE GENERATOR  (J.L. GRAY, O.S.U. 1972)

SUBJECT: GENERATION OF SLR(1) PARSING TABLE

AUTHOR: JOSEPH L. GRAY

INSTALLATION: OKLAHOMA STATE UNIVERSITY  IBM 360/65
              PL/1 LEVEL F VERSION 5.2C

DATE: FALL SEMESTER 1972

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. COMPILER CONSTRUCTION - GRIES
2. SIMPLE LR(K) GRAMMARS - DE REMER  CACM 14  P 453-460  JULY 1971
3. PRACTICAL TRANSLATORS FOR LR(K) LANGUAGES - DE REMER  PH.D. THESIS
   MIT  SEPT 1969
4. SIMPLE LR(K) GRAMMARS - DEFINITION AND IMPLEMENTATION - DE REMER
5. THE CARE AND FEEDING OF LR(K) GRAMMARS - AHO AND ULLMAN  PROC.
   THIRD ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING  MAY 1971
6. A TECHNIQUE FOR SPEEDING UP LR(K) PARSERS - AHO AND ULLMAN  ACM
   SYMPOSIUM ON THEORY OF COMPUTING  1972
7. ON THE TRANSLATION OF LANGUAGES FROM LEFT TO RIGHT - KNUTH
   INFORMATION AND CONTROL 8  1965
8. A THEOREM ON BOOLEAN MATRICES - WARSHALL  JACM P 11-12      1962
9. AN ALGORITHM FOR MAINTAINING DYNAMIC AVL TREES - VAN DOREN AND GRAY
   SUBMITTED TO FOURTH INTERNATIONAL SYMPOSIUM ON COMPUTING AND
   INFORMATION SCIENCE


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.
SLR1:                    PROC
REUSABLE:                  BEGIN
THE_WHOLE_THING:             BEGIN
READER_SECTION:                BEGIN
                               END READER_SECTION
DEBUG_SECTION:                 BEGIN
                               END DEBUG_SECTION
TABLE_GENERATE_SECTION:        BEGIN
LR0_GENERATE:                    BEGIN
                                 END LR0_GENERATE
SLR1_GENERATE:                   BEGIN
                                 END SLR1_GENERATE
                               END TABLE_GENERATE_SECTION
WARSHAL:                       PROC
                               END WARSHAL
BSTSLR:                        PROC
                               END BSTSLR
                             END THE_WHOLE_THING
                           GO TO REUSABLE
ERROR:                     ERROR MESSAGE OUTPUT
                           GO TO REUSABLE
                           END REUSABLE
                                                   END SLR1
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 CAPABITITIES OF THE SOURCE LANGUAGE. FOR SMALLER
HOST 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.

WHOLE 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 BSTSLR, 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.

LR0 GENERATE: THIS SECTION FIRST GENERATES THE CONFIGURATION SETS
AS IF THE GRAMMAR IS LR(0) THEN THE TRANSITION ENTRIES ARE PLACED
IN THE SLR(1) TABLE. THE FILLING IN OF REDUCTION ENTRIES IS
POSTPONED UNTIL THE FOLLOWING SECTION.

SLR1 GENERATE: THIS SECTION GENERATES THE COMPLETE SLR(1) 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:

BSTSLR: 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.


INPUT:

FROM FILE PARMIN THE FOLLOWING PARAMETERS ARE READ IN 11 FIELDS OF 4.
 1. N >= NUMBER OF PRODUCTIONS TO BE INPUT
 2. N >= MAXIMUM NUMBER OF PARTS IN ANY PRODUCTION (INCLUDING LHS)
 3. N >= MAXIMUM NUMBER OF CHARACTERS IN ANY INPUT SYMBOL (MAY BE
        LESS - ONLY NEED N LARGE ENOUGH TO MAKE SYMBOLS UNIQUE)
 4. N = MAXIMUM NUMBER OF DISTINCT SYMBOLS IN THE GRAMMAR
 5. N = CONFIGURATION SET LIMIT (FOR ALL SETS COMBINED)
 6. N = EXPECTED NUMBER OF CONFIGURATION SETS
 7. N = MAXIMUM NUMBER OF BASIS ENTRIES FOR ANY SET
 8. N = 1 TO ACTIVATE DEBUG SECTION
 9. N = 1 TO COUNT AND LIST INADEQUATE STATES
```

10. N = 1 FOR FULL PRINTED OUTPUT
11. N = 1 TO PUNCH SLR(1) TABLE AND OTHER DATA NEEDED FOR PARSER

THERE ARE DEFAULTS FOR 3 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.
IF MORE THAN ONE GRAMMAR IS INPUT, THEN THE PARAMETERS FOR EACH GRAMMAR ARE SIMPLY ENTERED IN THE PROPER ORDER. FROM FILE PRODIN, THE PRODUCTIONS ARE INPUT. THE FORMAT IS THE LHS (LEFT-HAND-SIDE) IMMEDIATELY FOLLOWED BY A COLON, FOLLOWED BY 1 OR MORE BLANKS, THEN THE RHS (RIGHT-HAND-SIDE) PARTS EACH FOLLOWED BY A COMMA AND 1 OR MORE BLANKS. THE RIGHTMOST PART OF AN ALTERNATIVE IS FOLLOWED BY A SEMICOLON AND 1 OR MORE BLANKS IF IT IS NOT THE LAST ALTERNATIVE; OTHERWISE, IT IS FOLLOWED BY A PERIOD AND 1 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. FOR MULTIPLE GRAMMAR INPUT, EACH GRAMMAR IS DELIMITED BY A CARD WITH A PERIOD IN COLUMN 1. NOTICE THIS ALLOWS THE PRODUCTIONS TO BE SEQUENCED WITHOUT AFFECTING THE ROUTINE.

OUTPUT:

ALL SIGNIFICANT INTERNAL TABLES AND STATISTICS ARE PRINTED AND LABELLED IF THE PRINT PARAMETER IS ENABLED. ALSO, ALL DATA NEEDED BY THE PARSER IS PUNCHED IF SO SELECTED BY THE USER. THE PARSER IS ENCLOSED AS A COMMENT. NOTICE THAT BY ALTERING THE DD STATEMENT FOR PUNCH, THE OUTPUT COULD BE ROUTED TO A DATA SET ON SECONDARY STORAGE. THIS IS MENTIONED SINCE, FOR EXAMPLE, THE PUNCHED OUTPUT FOR THE BASIC GRAMMAR IS APPROXIMATELY 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 PROD. THE CODE FOR EACH SYMBOL IS ITS LINEAR POSITION IN THE BINARY TREE STRUCTURED SYMBOL TABLE BUILT BY BSTSLR. THE INPUT SYMBOLS ARE SAVED AND SENT TO THE PARSER FOR ERROR MESSAGE CAPABILITIES AND, IN THE CASE OF TERMINAL SYMBOLS, FOR SCANNING PURPOSES. THREE MAPPING ARRAYS ARE MAINTAINED. MAPTO HAS AN ENTRY FOR EACH SYMBOL SUCH THAT BY APPLYING MAPTO 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. MAPFROM IS THE INVERSE OF MAPTO. ENDEX IS BUILT DURING INPUT SUCH THAT ENDEX APPLIED TO MAPTO APPLIED TO A CODED NON-TERMINAL YIELDS THE FIRST (LEXICOGRAPHICALLY) PRODUCTION IN WHICH THE SYMBOL IS THE LEFT-HAND-SIDE. TREE IS THE SYMBOL TABLE MAINTAINED BY BSTSLR AND IS DOCUMENTED ELSEWHERE C.F. REFERENCE. TABLE IS THE LR(0) THEN SLR(1) TABLE. EACH ROW DEFINES A SET, AND THE COLUMNS CORRESPOND TO THE SYMBOLS (MAPPED). SET IS A VECTOR THAT HOLDS ALL CONFIGURATION SETS. SLIM HOLDS THE LAST POSITION IN SET FOR EACH SET, AND BASIS HOLDS THE LAST POSITION IN SET OF THE BASIS PORTION OF EACH SET. DOT_POSITION IS AN ARRAY WHICH HOLDS THE DOT POSITION OF EACH BASIS ENTRY OF EACH SET (AN ENTRY OF 2 MEANS THE DOT IS TO THE LEFT OF THE RHS). MARKER IS A BIT VECTOR PARALLEL TO SET THAT IS SET TO 1 IF THE CORRESPONDING SET ELEMENT EITHER CANNOT OR HAS BEEN USED IN EXPANSION.

PROGRAM LOGIC:

THE INPUT-ENCODE SECTION IS STRAIGHT-FORWARD, AND THE USER WILL HAVE NO TROUBLE DETECTING THE LOGIC BY FOLLOWING THE SOURCE CODE. IF DEBUG IS SELECTED, THEN THE DEBUG SECTION IS ENTERED. THE DEBUG SECTION CAN BE DELETED WITHOUT AFFECTING THE PROGRAM. IT IS SIMPLY AN IMPLEMENTATION OF SOME OF THE GRAMMAR CHECKS OF GRIES. THE HEART OF THE PROGRAM IS THE TABLE GENERATE SECTION. IN THE LR0 SECTION, THE FIRST SET IS INITIALIZED TO PRODUCTION 1 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 WHOSE LHS IS THE SYMBOL TO THE RIGHT OF THE DOT. THESE 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 CLOSED. THIS CONTINUES UNTIL THE SYMBOL TO THE RIGHT OF THE DOT OF ALL UNCLOSED CLOSURE ENTRIES IS A TERMINAL OR A CLOSURE WOULD DUPLICATE A SET ELEMENT. NOW EXPANSION IS USED TO INITIATE A NEW SET. THE "CANDIDATE" FOR EXPANSION IS THE FIRST SET ENTRY WHOSE MARKER BIT IS 0. FOR WHICHEVER SET IT IS IN, ALL OF THAT SET'S ENTRIES WITH THE SAME SYMBOL TO THE RIGHT OF THE DOT ARE MARKED AND THEN USED TO FORM THE BASIS ENTRIES (THE DOT IS MOVED RIGHT 1 POSITION) OF A NEW SET PROVIDING SUCH ACTION WOULD NOT CAUSE DUPLICATION OF AN EXISTING SET. IN BOTH CLOSURE AND EXPANSION, A DUPLICATE IS NOT ONLY 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 (FUTURE STATE) WITH A REDUCTION ASSOCIATED WITH IT. THE SET ELEMENT, A PRODUCTION NUMBER, IS NEGATED AND ENTERED INTO REDUCE(I) PROVIDING THERE IS NO PREVIOUS ENTRY IN REDUCE(I). IF THERE IS, THEN REDUCE(I) IS SET TO THE NUMBER OF SUCH ENTRIES; AND THE ENTRIES THEMSELVES ARE STORED IN A QUEUE (MULT_REDUCE_Q). ANY ENTRIES WITH THE DOT TO THE RIGHT ARE MARKED (TAKEN OFF EXPANSION LIST) SINCE, IF THE DOT IS TO THE RIGHT, THEY CANNOT BE USED FOR EXPANSION SINCE THE DOT CANNOT BE MOVED FURTHER TO THE RIGHT. KEEP IN MIND THAT THE DOT POSITION FOR BASIS ENTRIES IS IN THE ARRAY DOT_POSITION WHEREAS THE DOT POSITION OF CLOSURE ENTRIES IS ASSUMED TO BE 2 (TO THE LEFT). THE ACTION OF CLOSING THEN EXPANDING CONTINUES UNTIL ALL ENTRIES ARE MARKED. DURING EXPANSION, THE NUMBER OF THE NEW SET GENERATED BY A CERTAIN SYMBOL TO THE RIGHT OF THE DOT WHILE WITHIN A CERTAIN SET IS ENTERED INTO TABLE. THAT IS, TABLE(I,J) <-- K WHERE I IS THE SET THE PROGRAM IS WORKING WITH, J IS THE MAPPED SYMBOL TO THE RIGHT OF THE DOT, AND K IS THE NEW SET GENERATED. A SIMILAR ENTRY IS MADE IF K IS THE SET WHICH WOULD BE DUPLICATED BY A PARTICULAR EXPANSION.
THE SLR1_GENERATE SECTION COMPUTES THE FOLLOW FUNCTION PER DE REMER'S THEOREM AND THEN PROCEEDS TO ENTER THE REDUCTIONS (PREVIOUSLY STORED IN REDUCE AND/OR MULT_REDUCE_Q) INTO ALL COLUMNS REPRESENTING SYMBOLS IN FOLLOW(A) WHERE A IS THE LHS OF THE PRODUCTION INVOLVED IN THE REDUCTION(S) OF A PARTICULAR ROW (SET) OF TABLE. AFTER SUCH ENTRY, THE ROW IS NOW, BY DEFINITION, A STATE OF THE PARSING TABLE. THAT IS, THE ROW NOW CONTAINS BOTH (POSSIBLY) STATE TRANSITIONS AND REDUCTIONS, HENCE A STATE. INADEQUATE STATES ARE THOSE WITH MORE THAN 1 REDUCTION OR A REDUCTION AND A STATE TRANSITION UNDER A TERMINAL SYMBOL. IF MORE THAN 1 ENTRY IS ATTEMPTED IN ANY TABLE POSITION, THEN THE GRAMMAR IS NOT SLR(1).

*** THE FOLLOWING IS A SAMPLE PARSER WHICH USES THE SLR(1) TABLES ***

```
###--SAMPLE PARSER-SCANNER FOR ARITH EXPR--###
PARSER: PROCEDURE OPTIONS (MAIN);
        DECLARE
          PRINT FILE PRINT,            /* OUTPUT FILE * /
          PRSIN FILE INPUT STREAM,     /* INPUT FROM SLR(1) GEN * /
          CARD (80) CHARACTER (1),
          (NO_PRODS,NO_PARTS,NO_SYMS,NO_CHARS,NO_SETS,NO_NON)
            FIXED BINARY (31,0);
        GET FILE (PRSIN) EDIT
          (NO_SYMS,NO_CHARS,NO_NON,NO_PARTS,NO_PRODS,NO_SETS)
          (6 F(3));
PRSR:BEGIN;
        DECLARE
          (FLUSH,GETNEXT) ENTRY RETURNS (FIXED BINARY (31,0)),
          POINT ENTRY,
          1 TREE,
            2 NODE (0:NO_SYMS) CHARACTER (NO_CHARS),
            2 LL (0:NO_SYMS) FIXED BINARY,
            2 RL (0:NO_SYMS) FIXED BINARY,
            2 TAG (NO_SYMS) BIT (2) ALIGNED,
            2 AVAIL FIXED BINARY (31,0),
            2 COUNT FIXED BINARY (31,0),
          (FLAG,POS) FIXED BINARY (31,0),
          PROD (NO_PRODS,NO_PARTS) FIXED BINARY,
          TABLE (2:NO_SETS,2:NO_SYMS) FIXED BINARY,
          (MAPTO,MAPFROM) (NO_SYMS) FIXED BINARY,
          STACK (20) FIXED BINARY INITIAL (2,3),
          TOP FIXED BINARY (31,0) INITIAL (2),
          (SYMBOL,TEMPSYM) FIXED BINARY (31,0),
          BSTSRC ENTRY (CHARACTER(NO_CHARS),,,),
          (I,J,L_RHS,TSC) FIXED BINARY (31,0);
        GET FILE (PRSIN) EDIT
          (AVAIL,COUNT,RL(0),(NODE(I),LL(I),RL(I),MAPTO(I),MAPFROM(I),
          TAG(I)  DO I=1 TO NO_SYMS),
          ((PROD(I,J) DO J=1 TO NO_PARTS) DO I=1 TO NO_PRODS),
          ((TABLE(I,J) DO J=2 TO NO_SYMS) DO I=2 TO NO_SETS))
          (3 F(3),(NO_SYMS)(A(NO_CHARS),4 F(3),B(2)),
          (NO_PARTS*NO_PRODS) F(3),(NO_SYMS*NO_SETS) F(4));
/*
NOTE: TAG, MAPFROM, AVAIL AND COUNT ARE NOT NEEDED FOR THIS PARSER.
ONLY STATES ARE STACKED, SYMBOL HOLDS NEXT INPUT SYMBOL TO
BE PROCESSED, TEMPSYM HOLDS SAME EXCEPT AFTER REDUCTION AT WHICH
TIME IT HOLDS A LHS.
* /
GET_SYM:
        SYMBOL=GETNEXT;
RETURN_FROM_ERROR:
        PUT FILE (PRINT) SKIP EDIT
          ('CURRENT INPUT SYMBOL --> ',NODE(SYMBOL))(2 A);
BACKUP:
        TEMPSYM=SYMBOL;
DRIVE:  TSC=TABLE(STACK(TOP),MAPTO(TEMPSYM));
        IF TSC > 1 THEN
          DO;
            TOP=TOP+1;
            IF TOP > HBOUND(STACK,1) THEN GO TO OVER;
            STACK(TOP)=TSC;
            PUT FILE (PRINT) SKIP EDIT
              ('STATE STACKED --> ',TSC)(A,F(3));
            IF TEMPSYM = SYMBOL THEN GO TO GET_SYM;
            ELSE GO TO BACKUP;
          END;
        IF TSC < 0 THEN
          DO;
            PUT FILE (PRINT) SKIP EDIT
              ('ATTEMPTING REDUCTION - PRESENT STACK --> ',
              (STACK(I) DO I=1 TO TOP))(A,(TOP) F(4));
            TEMPSYM=PROD(-TSC,1);
            DO L_RHS=1 TO NO_PARTS-2 WHILE (PROD(-TSC,L_RHS+2) ¬= 0);
            END;
            TOP=TOP-L_RHS;
            IF TOP < 2 THEN GO TO UNDER;
            PUT FILE (PRINT) SKIP EDIT
              ('REDUCTION ON PRODUCTION --> ',-TSC,'. ',
              (NODE(PROD(-TSC,J)) DO J=1 TO NO_PARTS
              WHILE (PROD(-TSC,J) ¬= 0)))(A,F(3),A,(NO_PARTS) A);
            GO TO DRIVE;
          END;
        IF TSC = 1 THEN GO TO ACCEPT;
CALL_POINT:
        STACK(1)=2;
        STACK(2)=3;
        TOP=2;
        PUT FILE(PRINT) SKIP EDIT('** ERROR')(A);
        CALL POINT;
        SYMBOL=FLUSH;
        GO TO RETURN_FROM_ERROR;
OVER:
        PUT FILE (PRINT) SKIP EDIT ('** STACK OVERFLOW - PROBABLE ',
          'CAUSE --> NESTING LEVEL GREATER THAN ',
          HBOUND(STACK,1)-NO_PARTS,' **')(2 A,F(3),A);
        GO TO CALL_POINT;
UNDER:
        PUT FILE (PRINT) SKIP EDIT ('** STACK UNDERFLOW **')(A);
        GO TO CALL_POINT;
ACCEPT:
        PUT FILE (PRINT) SKIP EDIT ('** PROGRAM ACCEPTED **')(A);
        GO TO ENDMAIN;
/* GETNEXT IS THE PERTINENT SCANNER. * /
GETNEXT: PROC RETURNS (FIXED BINARY(31,0));
        DECLARE
          IP FIXED BINARY (31,0) INITIAL (0B) STATIC,
          I FIXED BINARY (31,0);
FLSHSYM:
        IP=IP+1;
        CALL BSTSRC (CARD(IP),FLAG,POS,TREE);
        IF POS ¬= 0 THEN RETURN (POS);
        ELSE RETURN (2); /* "2" IS THE TRAILING DELIMITER. * /
/* PRINT PRESENT RECORD AND CURRENT SYMBOL. * /
POINT: ENTRY;
        PUT FILE(PRINT) SKIP EDIT
          ((CARD(I) DO I=1 TO 80),'$')(80 A(1),SKIP,X(IP),A);
        RETURN;
/* FLUSH TO NEXT STATEMENT ON ERROR. * /
FLUSH: ENTRY RETURNS (FIXED BINARY(31,0));
        IP=0;
        PUT FILE(PRINT) LIST ('**FLUSHING TO NEXT CARD**');
        GET EDIT(CARD) (80 A(1));
        GO TO FLSHSYM;
        END GETNEXT;
```

```
BSTSRC: PROCEDURE (ITEM,FLAG,POS,TREE);
   /*
PROCEDURE BSTSRC IS THE SEARCH SECTION OF BST C.F. 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) FIXED BIN (31,0),
        ITEM CHAR (*),
        1 TREE,
          2 NODE (*) CHAR (*),
          2 LL (*) FIXED BIN,
          2 RL (*) FIXED BIN,
          2 TAG (*) BIT (*) ALIGNED,
          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) ;
        CURR=RL(0) ;
        DO WHILE (CURR ¬= 0) ;
          IF ITEM = NODE(CURR) THEN
            /* RETURN SUCCESS * /
            DO ;
              FLAG=4 ;
              POS=CURR ;
              RETURN ;
            END ;
          IF ITEM > NODE(CURR) THEN CURR=RL(CURR) ;
          ELSE CURR=LL(CURR) ;
        END ;
        /* RETURN FAILURE * /
        POS=0 ;
        FLAG=5 ;
        RETURN ;
      END SEARCH ;
     END BSTSRC;
     END PRSR;
ENDMAIN:
END PARSER;
###--END OF SAMPLE PARSER--###
VARIABLE DESCRIPTION (ALL SECTIONS):
BAD       - WORKING VARIABLE - 0 FOR ANY SYMBOL NOT "WITHIN" THE USER'S
            PSEUDO GOAL SYMBOL
BASIS     - A VECTOR HOLDING THE POSITION OF EXTENT OF EACH BASIS SET
            IN THE VECTOR HOLDING ALL CONFIGURATION SETS
BUF       - INPUT BUFFER FOR PRODUCTIONS
BUFI      - VECTOR OVERLAID ON BUF
CANDIDATE - 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
DEBUG_GRAMMAR - INPUT PARAMETER, 1 IF USER WANTS ACTION VARIABLE
          REPRESENTS
DOT_POSITION - A MATRIX HOLDING THE DOT POSITIONS OF BASIS SETS'
          ELEMENTS
DOT_SWITCH - FALSE WHEN SCANNING NON BASIS ELEMENTS, TRUE OTHERWISE
ELEMENT - THE FIRST TRANSITION IN THE ROW OF TABLE BEING SCANNED
ENDEX   - A VECTOR SUCH THAT ENDEX (MAPTO(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)
I,J,K,L - LOOP INDICES AND LOCAL WORKING VARIABLES
LIMIT_BASIS - INPUT PARAMETER, LIMIT OF ANY BASIS SET
LNAME   - LENGTH OF INPUT PRODUCTION SYMBOL
MAPTO   - A VECTOR SUCH THAT MAPTO (ANY SYMBOL) MAPS THE SYMBOLS TO
          COLUMNS OF A MATRIX SUCH THAT THE NONTERMINALS ARE GROUPED
          AS ARE THE TERMINALS
MAPFROM - THE INVERSE OF MAPTO
MARKER  - A BIT VECTOR WHOSE I TH ENTRY IS 1 IF THE I TH
          CONFIGURATION SET ELEMENT CANNOT BE USED FOR EXPANSION (OR
          HAS BEEN USED)
MASTER_ERROR - ERROR SWITCH, TRUE IF UNABLE TO GENERATE SLR(1) TABLE
MULT_REDUCE_Q - A QUEUE USED TO HOLD REDUCTIONS FOR A GIVEN STATE IF
          MORE THAN 1
NAME    - AN INPUT PRODUCTION SYMBOL
NBASIS  - COUNTER OF BASIS ELEMENTS
NCHARS  - COUNTER OF NON BLANKS IN NAME
NO_BASIS - INPUT PARAMETER, MAXIMUM NUMBER OF BASIS ELEMENTS FOR ANY
          SET
NO_CHARS - INPUT PARAMETER, MAXIMUM NUMBER OF CHARACTERS IN ANY INPUT
          PRODUCTION SYMBOL
NO_INAD - INADEQUATE STATE COUNTER
NO_NON  - NONTERMINAL COUNTER
MARK    - A BIT VECTOR WHOSE I TH ENTRY IS 1 IF THE I TH PRODUCTION
          CAN DERIVE A TERMINAL STRING
NO_PARTS - INPUT PARAMETER, MAXIMUM NUMBER OF PARTS PER PRODUCTION
NO_PRODS - INPUT PARAMETER, MAXIMUM NUMBER OF INPUT PRODUCTIONS
NO_SETS - INPUT PARAMETER, MAXIMUM NUMBER OF CONFIGURATION SETS
NO_SYMS - INPUT PARAMETER, MAXIMUM NUMBER OF INPUT SYMBOLS
NO_TERM - NUMBER OF TERMINAL SYMBOLS
NPARTS  - PARTS COUNTER
NSETS   - CONFIGURATION SETS COUNTER
PARMIN  - INPUT FILE FOR PARAMETERS
PLACE   - THE FIRST PRODUCTION OF A GROUP WITH THE SAME LHS
PRINT   - OUTPUT PRINT FILE
PRODIN  - INPUT FILE FOR PRODUCTIONS (BLOCKSIZE = 80)
PUNCH   - OUTPUT PUNCH FILE
PROD    - AN ARRAY OF ENCODED PRODUCTION - THE CODE FOR A SYMBOL IS
          ITS POSITION IN THE SYMBOL TABLE
PT      - POINTER TO UNRECOGNIZED PORTION OF BUF
RED     - TRUE AS SOON AS A REDUCTION IS DETECTED IN PRESENT STATE
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 MULT_REDUCE_Q
SET     - THE VECTOR HOLDING ALL CONFIGURATION SETS
SET_LIMIT - THE "TOP" OF SET
SIG     - TRUE IF ANY PRODUCTION BECAME "MARKED" DURING LAST PASS
SLIM    - A VECTOR HOLDING THE EXTENT IN SET OF EACH CONFIGURATION
          SET
```

```
      DECLARE
        BSTINT ENTRY,
        BSTSLR ENTRY (CHARACTER(NO_CHARS),,,),
        MARSHAL ENTRY,
        1 TREE,
          2 NODE (0:NO_SYMS) CHARACTER (NO_CHARS) INITIAL (' '),
          2 LL (0:NO_SYMS) FIXED BINARY (15,0),
          2 RL (0:NO_SYMS) FIXED BINARY (15,0),
          2 TAG (NO_SYMS) BIT (2) ALIGNED,
          2 AVAIL FIXED BINARY (31,0),
          2 COUNT FIXED BINARY (31,0),
        PROD (NO_PRODS+2,NO_PARTS) FIXED BINARY (15,0)
          INITIAL (1,2,3,2,((NO_PRODS+2)*NO_PARTS) 0),
        ENDEX (NO_PRODS+1) FIXED BINARY (15,0) INITIAL (1),
        MAPTO (NO_SYMS) FIXED BINARY (15,0)
          INITIAL (1,(NO_SYMS) 0),
        MAPFROM (NO_SYMS) FIXED BINARY (15,0) INITIAL (1),
        NO_NON FIXED BINARY (31,0) INITIAL (1),
        NO_TERM FIXED BINARY (31,0) INITIAL (0);
/* THIS IS THE INPUT SECTION. */
READER_SECTION:
    BEGIN;
      DECLARE
        PRODIN FILE INPUT RECORD,
        BUF CHARACTER (80),
        BUFI (80) CHARACTER (1) DEFINED BUF,
        NAME CHARACTER (80) VARYING,
        (I,J) FIXED BINARY (31,0) INITIAL (1),
        (NCHARS,NPARTS) FIXED BINARY (31,0) INITIAL (4),
        (FLAG,POS,PT,LNAME) FIXED BINARY (31,0),
        SWITCH LABEL (GETCARD,NEXTSYM,SEMI);
      ON ENDFILE (PRODIN) GO TO ENDINPUT;
      ON SIZE SNAP SIGNAL ERROR;
      ON SUBRG SNAP SIGNAL ERROR;
      ON STRG SNAP SIGNAL ERROR;
      ON ERROR SNAP GO TO ERRO2;
      PUT FILE (PRINT) EDIT
        ('...BEGIN OUTPUT FOR INPUT-ENCODE SECTION...')(SKIP(3),A);
/*
INITIALIZE TREE USED AS SYMBOL TABLE AND INSERT GENERATED
GOAL SYMBOL AND SPECIAL DELIMITERS.
*/
      PUT FILE (PRINT) SKIP EDIT ('INPUT PRODUCTIONS',(17) '-',
        '  1.  GOAL : "?" , USER''S GOAL SYMBOL , "?"')
        (2 (COL(1),A),SKIP,A);
      CALL BSTINT (TREE);
      CALL BSTSLR ('GOAL',FLAG,POS,TREE);
      CALL BSTSLR ('"?"',FLAG,POS,TREE);
GETCARD:
      READ FILE (PRODIN) INTO (BUF);
      PT=1;
/* CHECK FOR "EOF" (ALLOWS MULTIPLE GRAMMAR INPUT). */
      IF BUFI(1) = '.' THEN GO TO ENDINPUT;
/* CARD MUST END WITH NON-BLANK TO PREVENT STRINGRANGE. */
      SUBSTR(BUF,73)=(8) '@';
NEXTSYM:
      DO PT=PT BY 1 WHILE (BUFI(PT) = ' ');
      END;
      NAME=SUBSTR(BUF,PT,INDEX(SUBSTR(BUF,PT),' '));
      LNAME=LENGTH(NAME);
```

```
SWITCH  - LABEL SWITCH SET PER INPUT PUNCTUATION
SYMBOL  - THE SYMBOL TO THE RIGHT OF THE DOT IN THE PRESENT SET
          ELEMENT
TABLE   - THE SLR(1) TABLE, EACH ROW IS A STATE, THE COLUMNS REPRESENT
          THE SYMBOLS, A POSITIVE ENTRY IS A STATE TRANSITION AND A
          NEGATIVE ENTRY IS A REDUCTION
TAIL    - THE TRANSITIVE CLOSURE OF THE TAIL SYMBOL MATRIX
TEMPDOT - THE DOT POSITION OF THE PRESENT SET ELEMENT
TERM    - TRUE WHEN A TRANSITION UNDER A TERMINAL SYMBOL IS IN THE
          PRESENT STATE
TOP     - TOP OF THE QUEUE
TREE    - A STRUCTURE REPRESENTING THE SYMBOL TABLE
TRY     - VALUE OF "GOTO" FUNCTION
TRYDOT  - DOT POSITIONS OF ELEMENTS OF TRY
TRYKNT  - NUMBER OF ELEMENTS IN TRY
U       - UPPER BOUND IN BINARY SEARCH
WITHIN  - BIT MATRIX OF "WITHIN" RELATION (AND CLOSURE)
*/
(SIZE,SUBRG,STRG):
SLR1: PROCEDURE OPTIONS (MAIN);
REUSABLE:
    BEGIN;
      DECLARE
        PARMIN FILE INPUT STREAM,
        (DT,TM) CHAR (6),
        PRINT FILE PRINT,
        (NO_CHARS,NO_PRODS,NO_PARTS,CONFIG_SET_LIMIT,NO_SETS,
          NO_BASIS,NO_SYMS,DEBUG_GRAMMAR,NO_PRINT,NO_PUNCH,
          COUNT_INADEQUATE_STATES) FIXED BINARY (31,0);
      ON ENDFILE (PARMIN) GO TO ENDMAIN;
      ON SUBRG SNAP SIGNAL ERROR;
      ON SIZE  SNAP SIGNAL ERROR;
      ON STRG  SNAP SIGNAL ERROR;
      ON ERROR SNAP GO TO ERRO1;
      OPEN FILE (PRINT) PAGESIZE (66) LINESIZE (132);
      DT=DATE;
      TM=TIME;
      GET FILE (PARMIN) EDIT
        (NO_PRODS,NO_PARTS,NO_CHARS,NO_SYMS,CONFIG_SET_LIMIT,
        NO_SETS,NO_BASIS,DEBUG_GRAMMAR,COUNT_INADEQUATE_STATES,
        NO_PRINT,NO_PUNCH)(COL(1),11 F(4));
      PUT FILE (PRINT) EDIT
        ('SLR(1) TABLE GENERATOR OUTPUT','DATE: ',SUBSTR(DT,3,2),
        ':',SUBSTR(DT,5,2),':',SUBSTR(DT,1,2),
        '(J.L. GRAY, COMPUTING AND INFORMATION SCIENCES DEPT., ',
        'O.S.U.)','TIME: ',SUBSTR(TM,1,2),':',SUBSTR(TM,3,2),':',
        SUBSTR(TM,5,2))(LINE(1),COL(51),A,COL(115),6 A,COL(35),2 A,
        COL(115),6 A);
/* SET DEFAULTS, IF NECESSARY. */
      IF NO_PRODS = 0 THEN NO_PRODS=50;
      IF NO_CHARS > 78 THEN NO_CHARS=78;
      IF NO_CHARS < 4 THEN NO_CHARS=4;
      IF NO_PARTS < 4 THEN NO_PARTS=4;
      IF CONFIG_SET_LIMIT = 0 THEN CONFIG_SET_LIMIT=5*NO_PRODS;
      IF NO_SETS = 0 THEN NO_SETS=2*(NO_PRODS+2);
      IF NO_BASIS = 0 THEN NO_BASIS=NO_PRODS/10;
      IF NO_SYMS = 0 THEN NO_SYMS=2*NO_PRODS;
/* THIS BEGIN BLOCK IS FOR DYNAMIC DECLARATION PURPOSES. */
THE_WHOLE_THING:
    BEGIN;
```

```
        IF LNAME < 3 THEN
          DO;
            IF LNAME = 0 THEN GO TO GETCARD;
            ELSE GO TO ERRO3;
          END;
        NCHARS=MAX(NCHARS,LNAME-2);
        PT=PT+LNAME;
/* INSERT IF NOT PRESENT ELSE EFFECTIVELY A SEARCH. */
        CALL BSTSLR (SUBSTR(NAME,1,LNAME-2),FLAG,POS,TREE);
        IF SUBSTR(NAME,LNAME-1,1) = ',' THEN
          DO;
            SWITCH=NEXTSYM;
ENTER:
            J=J+1;
            NPARTS=MAX(NPARTS,J);
            PROD(I,J)=POS;
            PUT FILE (PRINT) EDIT (NAME)(A);
            GO TO SWITCH;
          END;
        IF SUBSTR(NAME,LNAME-1,1) = ':' THEN
          DO;
            I=I+1;
            J=1;
            PROD(I,1)=POS;
            PUT FILE (PRINT) EDIT (I,'.  ',NAME)(COL(1),F(3),2 A);
            IF PROD(I,1) ¬= PROD(I-1,1) THEN
/* SET MAP ARRAYS FOR NON-TERMINAL. */
              DO;
                NO_NON=NO_NON+1;
                ENDEX(NO_NON)=I;
                MAPTO(POS)=NO_NON;
                MAPFROM(NO_NON)=POS;
              END;
            GO TO NEXTSYM;
          END;
        IF SUBSTR(NAME,LNAME-1,1) = '.' THEN
          DO;
/*
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-1,1) = ';' THEN
          DO;
            SWITCH=SEMI;
            GO TO ENTER;
SEMI:
            PROD(I+1,1)=PROD(I,1);
            I=I+1;
            PUT FILE (PRINT) SKIP EDIT (' ')(COL(9),A);
            J=1;
            GO TO NEXTSYM;
          END;
        GO TO ERRO4;
ENDINPUT:
/* OUTPUT STATISTICS ON INPUT GRAMMAR. */
        PUT FILE (PRINT) SKIP EDIT
          ('USER REQUESTED     ACTUALLY NEEDED',(34) '-',
           'NUMBER OF PARTS',NO_PARTS,NPARTS,'NUMBER OF PRODUCTIONS',
           NO_PRODS,I-1,'TOTAL SYMBOLS',NO_SYMS,COUNT,
           'NUMBER OF CHARACTERS',NO_CHARS,NCHARS)
           (SKIP,COL(30),A,SKIP,COL(30),A,4 (COL(8),A,COL(34),F(4),
           X(15),F(4)));
        NO_CHARS=NCHARS;
        NO_PARTS=NPARTS;
        NO_PRODS=I;
/* FIXUP LOOP TO SET MAP ARRAYS FOR TERMINAL SYMBOLS. */
        DO I=2 TO COUNT;
          IF MAPTO(I) = 0 THEN
            DO;
              NO_TERM=NO_TERM+1;
              NO_SYMS=NO_NON+NO_TERM;
              MAPTO(I)=NO_SYMS;
              MAPFROM(NO_SYMS)=I;
            END;
        END;
        PUT FILE (PRINT) EDIT ('NUMBER OF NON-TERMINALS IS ',NO_NON,
          'NUMBER OF TERMINALS IS ',NO_TERM)(SKIP,2 (SKIP,A,F(3)));
        IF NO_PRINT ¬= 1 THEN GO TO BYPASS1;
        PUT FILE (PRINT) SKIP (2) EDIT ('PROD#  LHS',(' RHS' DO I=1 TO
          NO_PARTS-1),(30) '-')((NO_PARTS) A,SKIP,A);
        PUT FILE (PRINT) EDIT ((I,'. ',(PROD(I,J) DO J=1 TO NO_PARTS)
          DO I=1 TO NO_PRODS))(SKIP,F(4),A(2),(NO_PARTS) F(4));
        PUT FILE (PRINT) SKIP (2) EDIT
          ('NODE# TO FROM NODE',(18)'-',(I,MAPTO(I),MAPFROM(I),NODE(I)
          DO I=1 TO COUNT))(A,SKIP,A,(COUNT)(SKIP,3 F(4),X(2),
          A(NO_CHARS)));
BYPASS1:
        IF NO_SYMS ¬= COUNT THEN GO TO ERRO5;
        PUT FILE (PRINT) EDIT
          ('...END OUTPUT FOR INPUT-ENCODE SECTION...')(SKIP,A);
     END READER_SECTION;
        IF DEBUG_GRAMMAR ¬= 1 THEN GO TO TABLE_GENERATE_SECTION;
/*
DEBUG DETECTS DEBUG PRODUCTIONS BY CONSTRUCTING THE RELATION
WITHIN+ AND ALGORITHM 2.8.3 P.42 -- COMPILER CONSTRUCTION - GRIES.
*/
DEBUG_SECTION:
     BEGIN;
        DECLARE
          WITHIN(2:NO_NON,2:NO_NON) BIT (1)
            INITIAL ((NO_NON*NO_NON)(1) '0'B) ALIGNED,
          MARK (NO_PRODS) BIT (1) INITIAL ((NO_PRODS)(1)'0'B) ALIGNED,
          SIG BIT (1) ALIGNED,
          (I,J,K,L) FIXED BINARY (31,0);
        ON SUBRG SNAP SIGNAL ERROR;
        ON STRG  SNAP SIGNAL ERROR;
        ON SIZE  SNAP SIGNAL ERROR;
        ON ERROR SNAP GO TO ERRO6;
        PUT FILE (PRINT) EDIT
          ('...BEGIN OUTPUT FOR DEBUG SECTION...')(SKIP,A);
/*
WITHIN'S ROWS CORRESPOND TO NON-TERMINALS AND HAVE A "1"
FOR EACH RHS PART "WITHIN" A LA GRIES.
*/
        DO I=2 TO NO_PRODS;
          DO J=2 TO NO_PARTS WHILE (PROD(I,J) ¬= 0);
            IF MAPTO(PROD(I,J)) <= NO_NON THEN
```

```
          WITHIN(MAPTO(PROD(I,1)),MAPTO(PROD(I,J)))='1'B;
        END;
      END;
/* CLOSURE VIA WARSHALL ALGORITHM. */
      CALL WARSHAL (WITHIN);
/*
ANY ZERO IN USER'S GOAL ROW (COL 3 FORWARD) MEANS SOME SYMBOL IS
NOT "WITHIN" THE USER'S GOAL.
*/
      DO J=3 TO NO_NON;
        IF ¬WITHIN(2,J) THEN
          PUT FILE (PRINT) EDIT (NODE(MAPFROM(J)),
            ' CANNOT APPEAR IN ANY SENTENTIAL FORM.')(SKIP,2 A);
      END;
/*
ALGORITHM FOR DETECTING PRODUCTIONS THAT CANNOT BE USED TO
DERIVE A SENTENCE, C.F. REFERENCE.
*/
TWO83:
      SIG='0'B;
      DO I=1 TO NO_PRODS;
        IF MARK(I) THEN GO TO END1;
        DO J=2 TO NO_PARTS WHILE (PROD(I,J) ¬= 0);
          IF MAPTO(PROD(I,J)) > NO_NON THEN GO TO END2;
/* LINEAR LOOK-UP FOR NON-TERMINAL AS A LHS. */
          DO L=2 TO NO_PRODS WHILE (PROD(L,1) ¬= PROD(I,J));
          END;
          IF L <= NO_PRODS THEN
            DO;
              DO K=L TO NO_PRODS WHILE(PROD(K,1) = PROD(L,1));
                IF MARK(K) THEN GO TO END2;
              END;
              GO TO END1;
            END;
          ELSE
            DO;
              PUT FILE (PRINT) EDIT
                (NODE(PROD(I,J)),' IS NOT A LHS.')(SKIP,2 A);
            END;
END2:
        END;
        MARK(I)='1'B;
        SIG='1'B;
END1:
      END;
      DO J=1 TO NO_PRODS;
        IF ¬MARK(J) THEN
          DO;
            IF SIG THEN GO TO TWO83;
            DO I=2 TO NO_PRODS-1;
              IF ¬MARK(I) THEN PUT FILE (PRINT) EDIT
                (I,' TH PRODUCTION USELESS.')(SKIP,F(3),A);
            END;
            GO TO DUPTEST;
          END;
      END;
DUPTEST:
/* NOW CHECK FOR DUPLICATE RHS. */
      DO I=1 TO NO_PRODS;
        DO J=I+1 TO NO_PRODS;
          IF PROD(J,2) = PROD(I,2) THEN
           DO;
             DO K=3 TO NO_PARTS;
               IF PROD(I,K) ¬= PROD(J,K) THEN GO TO NOTDUP;
             END;
             PUT FILE (PRINT) EDIT (NODE(PROD(I,2)),
               ' STARTS A DUPLICATE RHS FOR PRODUCTIONS ',
               J,' AND ',I,'.')(SKIP,A(NO_CHARS),3 (A,F(3)));
/*
NOTICE NO ERROR ON DUPLICATE RHS SINCE THE SLR(1) METHOD IS
UNAFFECTED BY SUCH THINGS, HOWEVER A MESSAGE IS PRINTED BECAUSE
OFTEN THIS CONDITION LEADS TO UNSOLVABLE INADEQUATE STATES.
*/
           END;
NOTDUP:
        END;
      END;
      PUT FILE (PRINT) EDIT
        ('...END OUTPUT FOR DEBUG SECTION...')(SKIP,A);
    END DEBUG_SECTION;
/* DECLARE GLOBAL STORAGE FOR LR0 AND SLR1. */
TABLE_GENERATE_SECTION:
    BEGIN;
      DECLARE
        REDUCE (2:NO_SETS) FIXED BINARY (15,0)
          INITIAL ((NO_SETS) 0),
        MULT_REDUCE_Q (NO_TERM) FIXED BINARY (15,0),
        TABLE (2:NO_SETS,2:NO_SYMS) FIXED BINARY (15,0);
      TABLE=0;
/* CONFIGURATION SET AND GOTO FUNCTION GENERATOR. */
LR0_GENERATE:
    BEGIN;
      DECLARE
        (NSETS,SET_LIMIT) FIXED BINARY (31,0) INITIAL (2),
        TOP FIXED BINARY (31,0) INITIAL (0),
        (CANDIDATE,NBASIS) FIXED BINARY (31,0) INITIAL (1),
        (U,SYMBOL,PLACE,FENCE,TRYKNT,LIMIT_BASIS,I,J,K,
          L,TEMPDOT) FIXED BINARY (31,0),
        SET (CONFIG_SET_LIMIT) FIXED BINARY (15,0) INITIAL ((2) 1),
        SLIM (0:NO_SETS) FIXED BINARY (15,0) INITIAL (0,1),
        BASIS (NO_SETS) FIXED BINARY (15,0) INITIAL ((2) 1),
        TRY (NO_BASIS) FIXED BINARY (15,0),
        TRYOUT (NO_BASIS) FIXED BINARY (15,0),
        DOT_POSITION (NO_SETS,NO_BASIS) FIXED BINARY (15,0)
          INITIAL (5,(NO_BASIS-1) *,2),
        DOT_SWITCH BIT (1) ALIGNED,
        MARKER (CONFIG_SET_LIMIT) BIT (1)
          INITIAL ((CONFIG_SET_LIMIT)(1) '0'B) ALIGNED;
      ON SIZE SNAP SIGNAL ERROR;
      ON SUBRG SNAP SIGNAL ERROR;
      ON STRG  SNAP SIGNAL ERROR;
      ON ERROR SNAP GO TO ERR07;
      PUT FILE (PRINT) EDIT
        ('...BEGIN OUTPUT FOR LR(0) GENERATE SECTION...')(SKIP,A);
CLOSE:
        LIMIT_BASIS=BASIS(NSETS)+SLIM(NSETS-1);
        DO J=SLIM(NSETS-1)+1 BY 1 WHILE (J <= SET_LIMIT);
/* TRUE IF SET(J) IS AN ELEMENT OF A BASIS SET. */
          IF J <= LIMIT_BASIS THEN
            DO;
```

```
            TEMPDOT=DOT_POSITION(NSETS,J-SLIM(NSETS-1));
/* CHECK FOR DOT TO RIGHT OF RHS. */
            IF TEMPDOT > NO_PARTS THEN SYMBOL=0;
            ELSE SYMBOL=PROD(SET(J),TEMPDOT);
          END;
        ELSE SYMBOL=PROD(SET(J),2);
/*
SYMBOL IS SYMBOL TO RIGHT OF DOT - IF NO SYMBOL THEN SET DOT_RIGHT AND
TAKE OFF EXPANSION ELIGIBILITY LIST, ENTERING PRODUCTION NUMBER
IN REDUCE IF EMPTY (NEGATIVE VALUE ENTERED) ELSE PUT IN QUEUE
AND SET REDUCE TO NUMBER OF ELEMENTS OF THIS SET IN QUEUE.
*/
        IF SYMBOL = 0 THEN
          DO;
            IF REDUCE(NSETS) = 0 THEN REDUCE(NSETS)=-SET(J);
            ELSE
              DO;
                IF REDUCE(NSETS) > 0 THEN
                  DO;
                    REDUCE(NSETS)=REDUCE(NSETS)+1;
                    TOP=TOP+1;
                    IF TOP > HBOUND(MULT_REDUCE_Q,1) THEN
                      GO TO ERR10;
                    MULT_REDUCE_Q(TOP)=-SET(J);
                  END;
                ELSE
                  DO;
                    TOP=TOP+2;
                    IF TOP > HBOUND(MULT_REDUCE_Q,1) THEN
                      GO TO ERR10;
                    MULT_REDUCE_Q(TOP-1)=REDUCE(NSETS);
                    MULT_REDUCE_Q(TOP)=-SET(J);
                    REDUCE(NSETS)=2;
                  END;
              END;
            MARKER(J)='1'B;
            GO TO PRODCLOSED;
          END;
/* NO CLOSURE FOR TERMINAL SYMBOLS. */
        IF MAPTO(SYMBOL) > NO_NON THEN GO TO PRODCLOSED;
        PLACE=ENDEX(MAPTO(SYMBOL));
/*
CHECK IF DUPLICATE WITHIN THIS SET --LINEAR LOOKUP - NO NEED
TO LOOK THROUGH BASIS ENTRIES AS THEY DO NOT HAVE DOT
TO LEFT LIKE PLACE DOES (NOT TRUE FOR GOAL BUT ITS UNIQUE I.E.
IF "GOAL" IS A RHS THEN TROUBLE - NOTICE THAT THIS LOOP IS NOT
EXECUTED FOR FIRST LEVEL CLOSURES - I.E. THE FIRST STEP
OF THE CLOSURE FOR A BASIS SET ELEMENT.
*/
        DO K=LIMIT_BASIS+1 TO SET_LIMIT;
          IF PLACE = SET(K) THEN GO TO PRODCLOSED;
        END;
/*
NOT DUPLICATE - THEREFORE ENTER PLACE (AND OTHERS WITH SAME LHS) INTO
THIS SET.
*/
        DO SET_LIMIT=SET_LIMIT+1 BY 1;
          IF SET_LIMIT > HBOUND(SET,1) THEN GO TO ERR11;
          SET(SET_LIMIT)=PLACE;
          IF PROD(PLACE,1) ¬= PROD(PLACE+1,1) THEN GO TO PRODCLOSED;
          PLACE=PLACE+1;
        END;
PRODCLOSED:
      END;
      SLIM(NSETS)=SET_LIMIT;
EXPAND:
/*
FIND FIRST '0' IN MARKER (PARALLEL TO EXISTING SETS) AND DETERMINE
VIA BINARY SEARCH WHICH SET IT BELONGS TO.
*/
        DO CANDIDATE=CANDIDATE+1 TO CONFIG_SET_LIMIT
          WHILE (MARKER(CANDIDATE));
        END;
        IF CANDIDATE > SLIM(NSETS) THEN GO TO LRO_FINIS;
        U=NSETS;
        L=1;
CKLU:
        IF U < L THEN
          DO;
            FENCE=L;
            GO TO EXIT_BINARY_SEARCH;
          END;
        FENCE=(L+U)/2;
        IF CANDIDATE = SLIM(FENCE) THEN GO TO EXIT_BINARY_SEARCH;
        IF CANDIDATE < SLIM(FENCE) THEN U=FENCE-1;
        ELSE L=FENCE+1;
        GO TO CKLU;
/*
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
DOT. ENTER ELEMENTS IN TRY AND DOT POSITIONS +1 IN TRYDOT.
*/
EXIT_BINARY_SEARCH:
        TRYKNT=1;
        TRY(1)=SET(CANDIDATE);
        MARKER(CANDIDATE)='1'B;
/* TRUE IF CANDIDATE NOT IN BASIS SET,THEREFORE DOT IS LEFT OF RHS. */
        IF CANDIDATE-SLIM(FENCE-1) > BASIS(FENCE) THEN
          DO;
            DOT_SWITCH='0'B;
            L,K=2;
            TRYDOT(1)=3;
          END;
        ELSE
          DO;
            DOT_SWITCH='1'B;
            K=DOT_POSITION(FENCE,CANDIDATE-SLIM(FENCE-1));
            TRYDOT(1)=K+1;
          END;
        SYMBOL=PROD(SET(CANDIDATE),K);
        DO J=CANDIDATE+1 TO SLIM(FENCE);
          IF DOT_SWITCH THEN
            DO;
              IF MARKER(J) THEN GO TO NOT_SAME;
/* TURN OFF DOT_SWITCH AS SOON AS OUT OF BASIS SET. */
              IF J-SLIM(FENCE-1) > BASIS(FENCE) THEN
                DO;
                  DOT_SWITCH='0'B;
                  L=2;
```

```
                END;
              ELSE L=DOT_POSITION(FENCE,J-SLIM(FENCE-1));
            END;
          IF PROD(SET(J),L) = SYMBOL THEN
            DO;
              TRYKNT=TRYKNT+1;
/* BASIS SET OVERFLOW??? */
              IF TRYKNT > NO_BASIS THEN GO TO ERR12;
              TRY(TRYKNT)=SET(J);
              TRYDOT(TRYKNT)=L+1;
              MARKER(J)='1'B;
            END;
NOT_SAME:
        END;
/*
NOW SEE IF TRY WOULD START A NEW SET OR JUST DUPLICATE AN EXISTING
SET. METHOD IS TO CHECK ALL EXISTING SETS (1 TO NSETS) WHOSE BASIS
ENTRY EQUALS TRYKNT AND CHECK BOTH ENTRIES AND DOT POSITIONS.
IF INEQUALITY EXISTS WITH ALL BASIS ELEMENTS OF EACH SET THEN TRY AND
TRYDOT AND TRYKNT ARE USED TO INITIALIZE A NEW SET (NSETS<-NSETS+1)
AND TO SET TABLE.
*/
        DO J=1 TO NSETS;
          IF BASIS(J) = TRYKNT THEN
            DO;
              DO K=1 TO TRYKNT;
                DO L=1 TO TRYKNT;
                  IF TRY(K) = SET(SLIM(J-1)+L) &
                     TRYDOT(K) = DOT_POSITION(J,L) THEN GO TO IN_SET;
                END;
                GO TO NEW_SET;
IN_SET:
              END;
/*
SET TRANSITION UNDER THIS SYMBOL IN TABLE- TRY IS DUPLICATED BY (J) TH
BASIS SET.
*/
              TABLE(FENCE,MAPTO(SYMBOL))=J;
              GO TO EXPAND;
            END;
NEW_SET:
        END;
        NSETS=NSETS+1;
        IF NSETS > NO_SETS THEN GO TO ERR13;
/* SET TRANSITION TO THIS NEW STATE (SET, THAT IS). */
        TABLE(FENCE,MAPTO(SYMBOL))=NSETS;
        NBASIS=MAX(NBASIS,TRYKNT);
        DO J=1 TO TRYKNT;
          SET_LIMIT=SET_LIMIT+1;
          IF SET_LIMIT > HBOUND(SET,1) THEN GO TO ERR11;
          SET(SET_LIMIT)=TRY(J);
          DOT_POSITION(NSETS,J)=TRYDOT(J);
        END;
        SLIM(NSETS)=SET_LIMIT;
        BASIS(NSETS)=TRYKNT;
        GO TO CLOSE;
LRO_FINIS:
        PUT FILE (PRINT) SKIP EDIT
          ('USER REQUESTED     ACTUALLY NEEDED',(34) '-',
           'NUMBER OF SETS',NO_SETS,NSETS,'LENGTH OF SETS',
           CONFIG_SET_LIMIT,SET_LIMIT,'NUMBER IN BASIS SET',NO_BASIS,
           NBASIS)(SKIP,COL(30),A,SKIP,COL(30),A,3 (COL(10),A,COL(34),
           F(4),X(15),F(4)));
        IF NO_PRINT ¬= 1 THEN GO TO BYPASS2;
/* CONFIGURATION SET OUTPUT. */
        PUT FILE (PRINT) SKIP (5) EDIT
          ('CONFIGURATION SETS...',
           'POSITION  ELEMENT  DOT  BOUND  BASIS  SET #',
           (44) '-')(A,2 (SKIP,A));
        DO I=1 TO NSETS;
          J=SLIM(I-1)+1;
          PUT FILE (PRINT) EDIT
            (J,SET(J),DOT_POSITION(I,1),SLIM(I),BASIS(I),I,
             NODE(PROD(SET(J),1)),' --> ',
             (NODE(PROD(SET(J),L)) DO L=2 TO DOT_POSITION(I,1)-1),'.',
             (NODE(PROD(SET(J),L)) DO L=DOT_POSITION(I,1) TO NO_PARTS))
             (SKIP,F(4),X(6),F(4),X(5),F(2),X(3),F(4),X(3),F(3),
             X(5),F(3),COL(53),(NO_PARTS+2) (A,X(1)));
          PUT FILE (PRINT) EDIT
            ((K,SET(K),DOT_POSITION(I,K-J+1),
             NODE(PROD(SET(K),1)),' --> ',
             (NODE(PROD(SET(K),L)) DO L=2 TO DOT_POSITION(I,K-J+1)-1),
             '.',(NODE(PROD(SET(K),L)) DO L=DOT_POSITION(I,K-J+1) TO
             NO_PARTS) DO K=J+1 TO BASIS(I)+J-1))
             (SKIP,F(4),X(6),F(4),X(5),F(2),
             COL(53),(NO_PARTS+2) (A,X(1)));
          PUT FILE (PRINT) EDIT
            ((K,SET(K),'2',NODE(PROD(SET(K),1)),' --> .',
             (NODE(PROD(SET(K),L)) DO L=2 TO NO_PARTS)
             DO K=BASIS(I)+J TO SLIM(I)))
             (SKIP,F(4),X(6),F(4),X(6),A(1),
             COL(53),(NO_PARTS+1) (A,X(1)));
        END;
BYPASS2:
        NO_SETS=NSETS;
        PUT FILE (PRINT) EDIT
          ('...END OUTPUT FOR LR(0) GENERATE SECTION...')(SKIP,A);
     END LRO_GENERATE;
/* SLR(1) TABLE GENERATOR. */
SLR1_GENERATE:
     BEGIN;
        DECLARE
          PUNCH FILE OUTPUT STREAM,
          (ELEMENT,NO_INAD) FIXED BINARY (31,0) INITIAL (0),
          TOP FIXED BINARY (31,0) INITIAL (1),
          (I,J,K,L) FIXED BINARY (31,0),
          TAIL (2:NO_NON,2:NO_NON) BIT (1)
            INITIAL ((NO_NON-2)('1'B,(NO_NON-1)(1)'0'B),'1'B) ALIGNED,
          FOLLOW (2:NO_NON,NO_NON+1:NO_SYMS) BIT (1)
            INITIAL ((NO_NON*NO_TERM)(1) '0'B) ALIGNED,
          MASTER_ERROR BIT (1) INITIAL ('0'B) ALIGNED,
          (RED,TERM) BIT (1) ALIGNED;
        ON SIZE SNAP SIGNAL ERROR;
        ON SUBRG SNAP SIGNAL ERROR;
        ON STRG SNAP SIGNAL ERROR;
        ON ERROR SNAP GO TO ERR08;
        PUT FILE (PRINT) EDIT
          ('...BEGIN SLR(1) GENERATE SECTION OUTPUT...')(SKIP,A);
/*
FORM TAIL SYMBOL (NONTERMINAL ONLY) TRANSITIVE CLOSURE MATRIX
```

```
(TAIL INITIALIZED TO AN IDENTITY MATRIX OF DIMENSION NO_NON).
*/
        DO I=2 TO NO_PRODS;
          DO J=2 TO NO_PARTS-1 WHILE (PROD(I,J+1) ¬= 0);
          END;
          IF MAPTO(PROD(I,J)) <= NO_NON THEN
            TAIL(MAPTO(PROD(I,1)),MAPTO(PROD(I,J)))='1'B;
        END;
        CALL WARSHAL (TAIL);
/*
COMPUTE FOLLOW PER DEREMER'S THEOREM AND BOOLEAN MATRIX TECHNIQUES
SIMILAR TO WARSHALL'S ALGORITHM. NOTICE THAT FOLLOW OF EVERY NON-TERM
IS COMPUTED RATHER THAN JUST ONES FOR INADEQUATE STATES AND
THAT THE "TRANSPOSE" OF THE TAIL MATRIX IS USED.
*/
        DO J=2 TO NO_NON;
          DO I=2 TO NO_NON;
            IF TAIL(I,J) THEN
/*
IF THE FOLLOWING "DO" WAS ONLY EXECUTED FOR K=THE INADEQUATE STATE
THEN THIS ROUTINE WOULD DUPLICATE DE REMER'S METHOD. THAT IS, FOR
A REDUCE STATE THE REDUCTION WOULD BE ENTERED IN ALL TERMINAL
COLUMNS OF THE TABLE.
*/
              DO K=2 TO NO_SETS;
                IF TABLE(K,I) ¬= 0 THEN
                  DO L=NO_NON+1 TO NO_SYMS;
                    IF TABLE(TABLE(K,I),L) ¬= 0 THEN FOLLOW(J,L)='1'B;
                  END;
              END;
          END;
        END;
/*
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
            DO J=NO_NON+1 TO NO_SYMS;
              IF FOLLOW(MAPTO(PROD(-REDUCE(I),1)),J) THEN
                DO;
                  IF TABLE(I,J) ¬= 0 THEN
                    DO;
                      PUT FILE (PRINT) SKIP EDIT
                        ('STATE ',I,' IS INADEQUATE AND THE SIMPLE ',
                        '1-LOOK AHEAD SETS ARE NOT DISJOINT.',
                        'TRANSITION IS UNDER ',NODE(MAPFROM(J))
                        ,' = ',MAPFROM(J),' IN COLUMN ',J-1,
                        ', TRYING TO REPLACE ',TABLE(I,J),' WITH ',
                        REDUCE(I))(A,F(3),2 A,SKIP,A,A(NO_CHARS),A,
                        F(3),A,F(3),A,F(4),A,F(4));
                      MASTER_ERROR='1'B;
                    END;
                  ELSE TABLE(I,J)=REDUCE(I);
                END;
            END;
/* MORE THAN 1 REDUCTION FOR THIS SET. */
          ELSE
            DO TOP=TOP TO TOP+REDUCE(I)-1;
              DO J=NO_NON+1 TO NO_SYMS;
                IF FOLLOW(MAPTO(PROD(-MULT_REDUCE_Q(TOP),1)),J) THEN
                  DO;
                    IF TABLE(I,J) ¬= 0 THEN
                      DO;
                        PUT FILE (PRINT) SKIP EDIT
                          ('STATE ',I,' IS INADEQUATE AND THE SIMPLE '
                          ,'1-LOOK AHEAD SETS ARE NOT DISJOINT.',
                          'TRANSITION IS UNDER ',NODE(MAPFROM(J))
                          ,' = ',MAPFROM(J),' IN COLUMN ',J-1,
                          ' TRYING TO REPLACE ',TABLE(I,J),' WITH ',
                          MULT_REDUCE_Q(TOP))(A,F(3),2 A,SKIP,A,
                          A(NO_CHARS),A,F(3),A,F(3),A,F(4),A,F(4));
                        MASTER_ERROR='1'B;
                      END;
                    ELSE TABLE(I,J)=MULT_REDUCE_Q(TOP);
                  END;
              END;
            END;
SKIP_REDUCE:
        END;
        IF COUNT_INADEQUATE_STATES ¬= 1 THEN GO TO PRTOT;
/*
NOW COUNT THE INADEQUATE STATES, IF FOR ANY REASON THE STATE IS
FOUND TO BE INADEQUATE THEN IT IS NOTED AND NO FURTHER CHECKING
IS DONE FOR THAT STATE.
*/
        PUT FILE (PRINT) SKIP (4) EDIT
          ('RESULTS OF INADEQUATE STATE COUNTER (NOT INCLUDING ',
          'UNSOLVABLE STATES) FOLLOW...')(2 A);
        DO I=2 TO NO_SETS;
          ELEMENT=0;
          TERM,RED='0'B;
          DO J=2 TO NO_SYMS;
            IF TABLE(I,J) = 0 THEN GO TO ENDBLCK;
            IF TABLE(I,J) < 0 THEN
              DO;
                IF RED THEN
                  DO;
/* CHECK FOR SAME REDUCTION IN THIS SET. */
                    IF TABLE(I,J) ¬= ELEMENT THEN
                      DO;
                        PUT FILE (PRINT) EDIT
                          ('STATE ',I,' IS INADEQUATE BECAUSE OF '
                          ,'MULTIPLE REDUCTIONS.')(SKIP,A,F(4),2 A);
                        NO_INAD=NO_INAD+1;
                        GO TO ENDSTCK;
                      END;
                  END;
                ELSE
                  DO;
                    RED='1'B;
                    IF TERM THEN GO TO MIXED;
                    ELEMENT=TABLE(I,J);
                  END;
              END;
            ELSE
              DO;
                IF J > NO_NON THEN
```

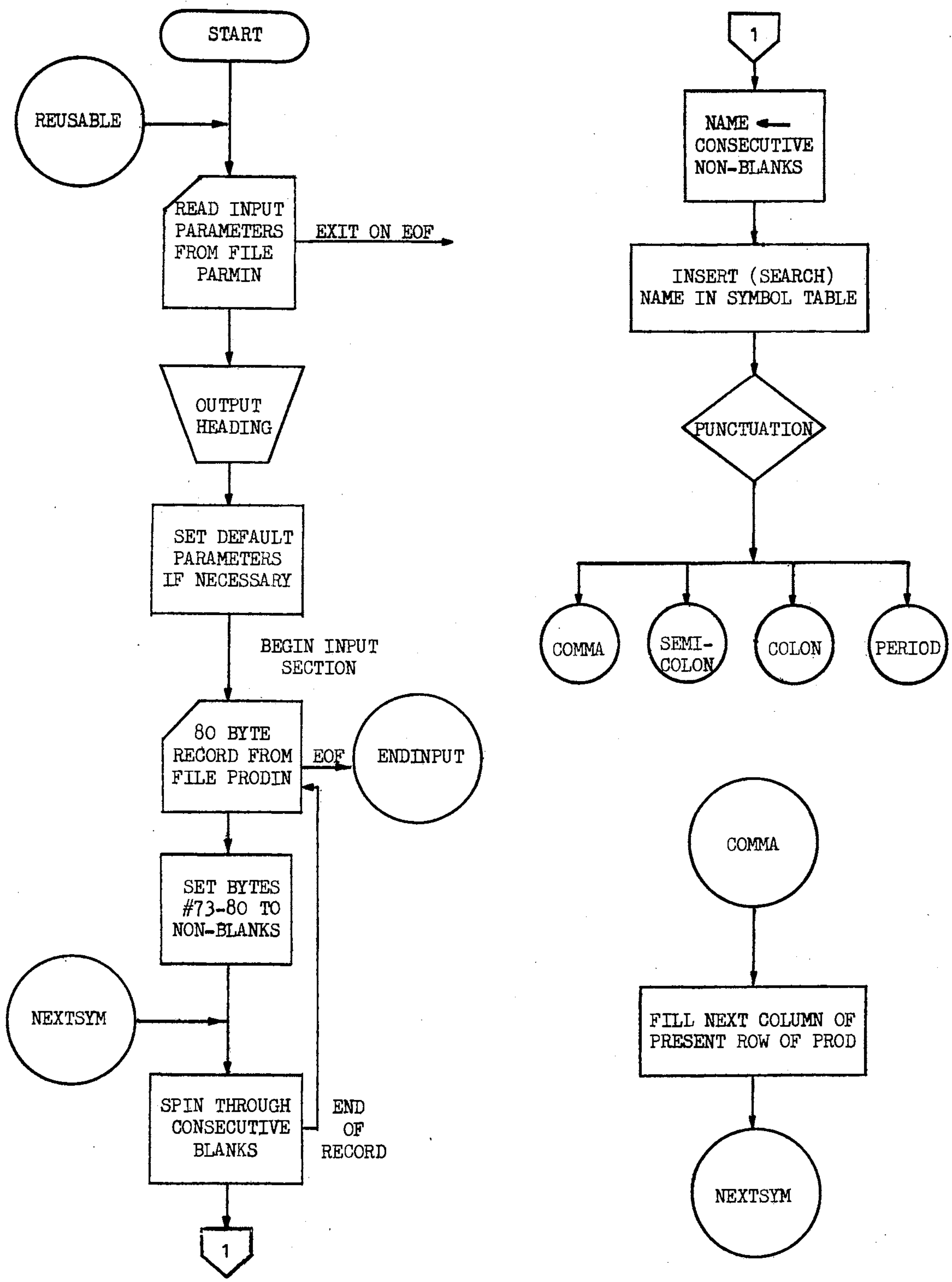
```
                    DO;
                      TERM='1'B;
                      IF RED THEN
MIXED:
                        DO;
                          PUT FILE (PRINT) EDIT
                            ('STATE ',I,' IS INADEQUATE BECAUSE OF '
                            ,'CONTAINING BOTH A REDUCTION AND A ',
                            'TRANSITION.')(SKIP,A,F(4),3 A);
                          NO_INAD=NO_INAD+1;
                          GO TO ENDSTCK;
                        END;
                    END;
                END;
ENDBLCK:
            END;
ENDSTCK:
        END;
        PUT FILE (PRINT) EDIT
          ('THERE IS A TOTAL OF ',NO_INAD,' INADEQUATE STATES.')
          (SKIP,A,F(3),A);
PRTOT:
        IF NO_PRINT ¬= 1 THEN GO TO BYPASS3;
        PUT FILE (PRINT) SKIP (5) EDIT
          ('COLUMNS REPRESENT...',('COL(',I-1,') = ',NODE(MAPFROM(I))
          DO I=2 TO NO_SYMS))(SKIP,A,(NO_SYMS)(COL(1),
          (132/(15+NO_CHARS))(A,F(3),A,A(NO_CHARS),X(4))));
        PUT FILE (PRINT) SKIP(5) LIST ('GENERATED SLR(1) TABLE...');
        DO I=2 TO NO_SETS;
          PUT FILE (PRINT) EDIT
            (I,'.',(TABLE(I,J) DO J=2 TO NO_SYMS))
            (SKIP,F(3),A,99 (32 F(4),COL(5)));
        END;
BYPASS3:
        IF MASTER_ERROR THEN GO TO ERR09;
        IF NO_PUNCH =1 THEN
          PUT FILE (PUNCH) EDIT
            (NO_SYMS,NO_CHARS,NO_NON,NO_PARTS,NO_PRODS,NO_SETS,
            AVAIL,COUNT,RL(0),(NODE(I),LL(I),RL(I),MAPTO(I),MAPFROM(I)
            ,TAG(I)  DO I=1 TO NO_SYMS),
            ((PROD(I,J) DO J=1 TO NO_PARTS) DO I=1 TO NO_PRODS),
            ((TABLE(I,J) DO J=2 TO NO_SYMS) DO I=2 TO NO_SETS))
            (COL(1),9 F(3),(NO_SYMS)(A(NO_CHARS),4 F(3),B(2)),
            (NO_PARTS*NO_PRODS) F(3),(NO_SYMS*NO_SETS) F(4));
        PUT FILE (PRINT) EDIT
        ('...END SLR(1) GENERATE SECTION OUTPUT...')(SKIP,A);
    END SLR1_GENERATE;
    END TABLE_GENERATE_SECTION;
/* WARSHALL ALGORITHM FOR COMPUTING CLOSURE MATRICES. */
WARSHAL: PROCEDURE (M);
/*
   WARSHALL ALGORITHM  (C.F. JACM 1962  P. 11)
   1. M(I,J)0 = M(I,J)
   2. M(I,J)K+1 = M(I,J)K | (M(I,K+1)K & M(K+1,J)K)
   3. M(I,J)STAR = M(I,J)D
   WHERE D = NO. OF ROWS OF M (A SQUARE BOOLEAN MATRIX).
*/
        DECLARE
          M (*,*) BIT (1) ALIGNED,
          (I,J,K) FIXED BINARY (31,0);
        DO K=LBOUND(M,1) TO HBOUND(M,1);
          DO I=LBOUND(M,1) TO HBOUND(M,1);
            IF M(I,K) THEN
              DO J=LBOUND(M,2) TO HBOUND(M,2);
                IF M(K,J) THEN M(I,J)='1'B;
              END;
          END;
        END;
END WARSHAL;
/* BALANCED BINARY SEARCH TREE SYMBOL TABLE MAINTENANCE. */
BSTSLR: PROCEDURE (ITEM,FLAG,POS,TREE);
      /*
      PROCEDURE BSTSLR IS THE IMPLEMENTATION OF AN ALGORITHM FOR
      PROCESSING AND MAINTAINING A DYNAMIC INFORMATION
      STRUCTURE IN THE FORM OF A PARTICULAR TYPE OF BINARY
      SEARCH TREE, AN AVL TREE.
      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) FIXED BINARY (31,0),
          ITEM CHAR (*),
          1 TREE,
            2 NODE (*) CHARACTER (*),
            2 LL (*) FIXED BINARY (15,0),
            2 RL (*) FIXED BINARY (15,0),
            2 TAG (*) BIT (*) ALIGNED,
            2 AVAIL FIXED BINARY (31,0),
            2 COUNT FIXED BINARY (31,0),
          L1 (0:32767) FIXED BINARY (15,0) BASED (L1PNT),
          L2 (0:32767) FIXED BINARY (15,0) BASED (L2PNT);
INSERT:
     BEGIN;
        /*
        ATTEMPT TO INSERT THE SPECIFIED NODE IN THE TREE.
        RETRACE THE SEARCH PATH TO PERFORM BALANCE TAG
        MAINTENANCE AND AT MOST ONE RESTRUCTURING.
        THREE BALANCE TAG CONDITIONS WHICH REQUIRE SEPARATE
        ACTION MAY OCCUR AT A NODE DURING PATH RETRACING:
          1) TAG='00'B - SET BALANCE TAG IN THE DIRECTION OF
             INSERTION AND RETRACE FURTHER (LONGER PATH);
          2) TAG IS UNBALANCED IN THE OPPOSITE DIRECTION
             FROM INSERTION - SET TAG TO '00'B AND EXIT;
          3) TAG IS UNBALANCED IN THE SAME DIRECTION AS
             INSERTION - NODE IS "CRITICAL"; RESTRUCTURE THE
             SUBTREE IT HEADS AND EXIT;
        RESTRUCTURING CONSISTS OF TWO BASIC CASES WITHIN
        LEFT-RIGHT SYMMETRY:
          1) CRITICAL NODE IS LEFT(RIGHT) HEAVY AND ITS LEFT
             (RIGHT) DESCENDANT IS LEFT(RIGHT) HEAVY -
             ROTATE SUBTREE COMPONENTS;
          2) CRITICAL NODE IS LEFT(RIGHT) HEAVY AND ITS LEFT
             (RIGHT) SUBTREE IS RIGHT(LEFT) HEAVY - SPLIT
             ROTATE SUBTREE COMPONENTS.
        */
          DECLARE
```

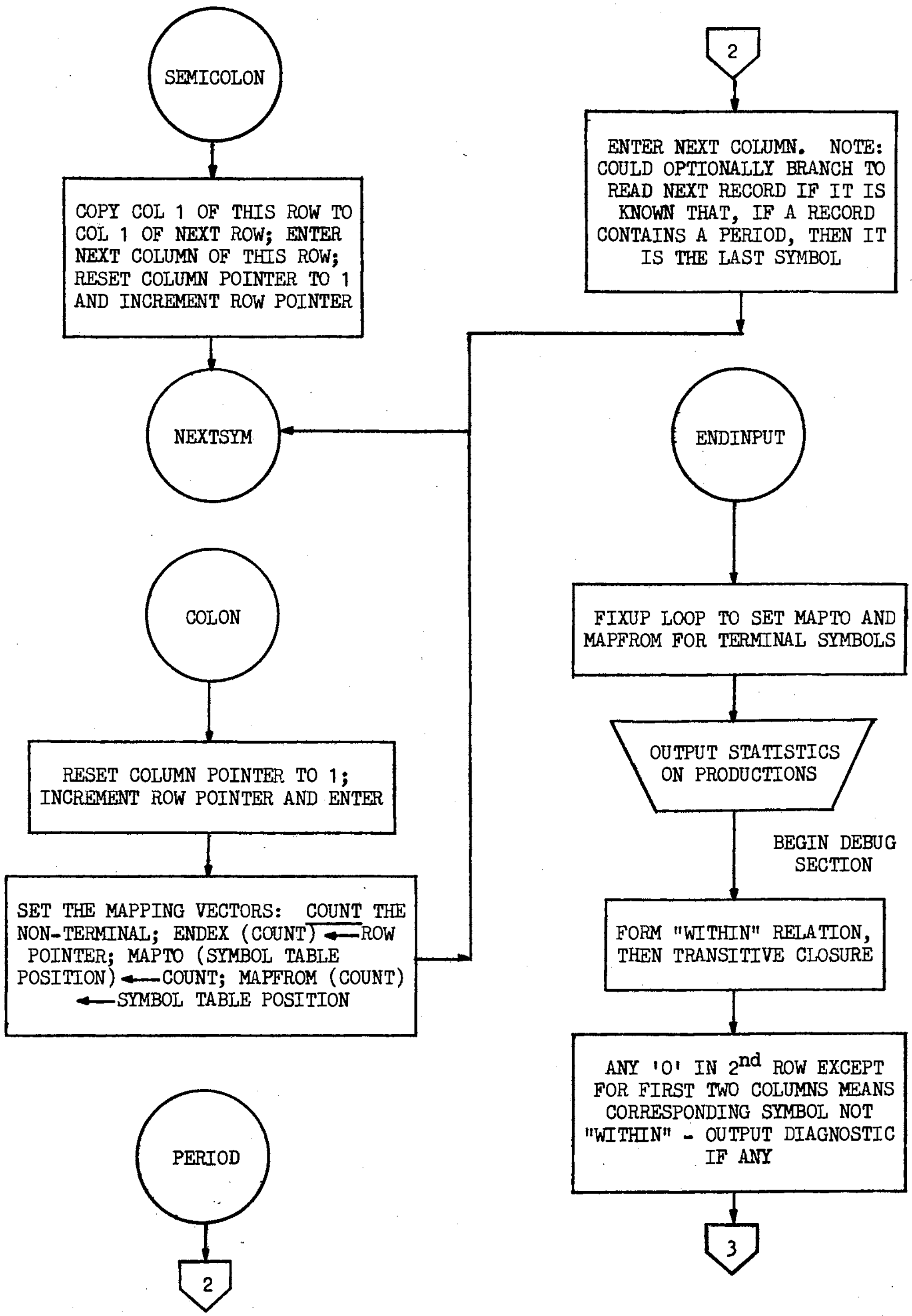
```
  (CURR,STACKTOP,STACKTP1,STACKTP2,TOP)
     FIXED BINARY (31,0),
/* STACK AND STKFLG ARE PUSH DOWN STACK VECTORS.
TREE SIZE >=46367 REQUIRES LARGER STACK */
  STACK (0:21) FIXED BINARY (15,0) INITIAL (0),
  STKFLG (0:21) BIT (1) INITIAL ('1'B) ALIGNED,
  BOOL BIT (1) ALIGNED ;
/* SEARCH FOR THE NODE WHICH WILL BE THE FATHER OF
THE NODE TO BE INSERTED. TRACE THE PATH FOR LATER
USE */
CURR,STACK(1)=RL(0) ;
DO TOP=1 BY 1 WHILE (CURR ¬= 0) ;
  STACK(TOP)=CURR ;
  IF ITEM=NODE(CURR) THEN
    /* DUPLICATE KEY */
    DO ;
      FLAG=4 ;
      POS=CURR ;
      RETURN ;
    END ;
  STKFLG(TOP)=ITEM > NODE(CURR) ;
  IF STKFLG(TOP) THEN CURR=RL(CURR) ;
  ELSE CURR=LL(CURR) ;
END ;
IF AVAIL = 0 THEN
  /* RETURN SPACE OVERFLOW CODE */
  DO ;
    FLAG=6 ;
    POS=0 ;
    RETURN ;
  END ;
/* GET SPACE FROM AVAILABILITY LIST */
STACK(TOP)=AVAIL ;
TOP=TOP-1 ;
IF STKFLG(TOP) THEN RL(STACK(TOP))=AVAIL ;
ELSE LL(STACK(TOP))=AVAIL ;
NODE(AVAIL)=ITEM ;
COUNT=COUNT+1 ;
FLAG=2 ;
POS=AVAIL ;
AVAIL=RL(AVAIL) ;
RL(STACK(TOP+1))=0 ;
/* ROOT NODE? */
IF TOP = 0 THEN RETURN ;
/* RETRACING */
DO WHILE (TAG(STACK(TOP)) = '00'B) ;
  /* CONDITION 1 */
  IF STKFLG(TOP) THEN TAG(STACK(TOP))='01'B ;
  ELSE TAG(STACK(TOP))='10'B ;
  TOP=TOP-1 ;
  IF TOP = 0 THEN RETURN ;
END ;
BOOL=TAG(STACK(TOP)) = '10'B ;
IF (BOOL&STKFLG(TOP)) | ¬(BOOL|STKFLG(TOP)) THEN
  /* CONDITION 2 */
  DO ;
    TAG(STACK(TOP))='00'B ;
    RETURN ;
  END ;
/* CONDITION 3 - RESTRUCTURE */
         STACKTOP=STACK(TOP) ;
         STACKTP1=STACK(TOP+1) ;
         STACKTP2=STACK(TOP+2) ;
         TAG(STACKTOP),TAG(STACKTP1)='00'B ;
         /* POINTERS FOR RIGHT OR LEFT SYMMETRY */
         IF STKFLG(TOP) THEN
           DO ;
             L1PNT=ADDR(RL) ;
             L2PNT=ADDR(LL) ;
           END ;
         ELSE
           DO ;
             L1PNT=ADDR(LL) ;
             L2PNT=ADDR(RL) ;
           END ;
         IF STKFLG(TOP) ¬= STKFLG(TOP+1) THEN GO TO CASE2 ;
         /* CASE 1 RESTRUCTURING */
         IF STKFLG(TOP-1) THEN RL(STACK(TOP-1))=STACKTP1 ;
         ELSE LL(STACK(TOP-1))=STACKTP1 ;
         L1(STACKTOP)=L2(STACKTP1) ;
         L2(STACKTP1)=STACKTOP ;
         RETURN ;
CASE2:
         /* CASE 2 RESTRUCTURING */
         IF STKFLG(TOP-1) THEN RL(STACK(TOP-1))=STACKTP2 ;
         ELSE LL(STACK(TOP-1))=STACKTP2 ;
         /* BALANCE TAG VARIATIONS */
         IF L2(STACKTOP) ¬= 0 THEN
           DO ;
             TAG(STACKTP2)='00'B ;
             IF STKFLG(TOP+2) THEN
               DO ;
                 IF STKFLG(TOP) THEN TAG(STACKTOP)='10'B ;
                 ELSE TAG(STACKTP1)='10'B ;
               END ;
             ELSE
               DO ;
                 IF STKFLG(TOP) THEN TAG(STACKTP1)='01'B ;
                 ELSE TAG(STACKTOP)='01'B ;
               END ;
           END ;
         L2(STACKTP1)=L1(STACKTP2) ;
         L1(STACKTP2)=L1(STACKTOP) ;
         L1(STACKTOP)=L2(STACKTP2) ;
         L2(STACKTP2)=STACKTOP ;
         RETURN ;
      END INSERT;
BSTINT:ENTRY (TREE);
INITIAL:
      BEGIN;
        /*
        CONSTRUCT  AVAILABILITY LIST BY USING RIGHT LINK
        FIELDS OF EACH AVAILABLE NODE POSITION.  SET OTHER
        COMPONENTS TO NULL VALUES.
        */
          DECLARE I FIXED BINARY (31,0);
          AVAIL=1 ;
          DO I=2 TO HBOUND(RL,1) ;
            RL(I-1)=I ;
          END ;
```

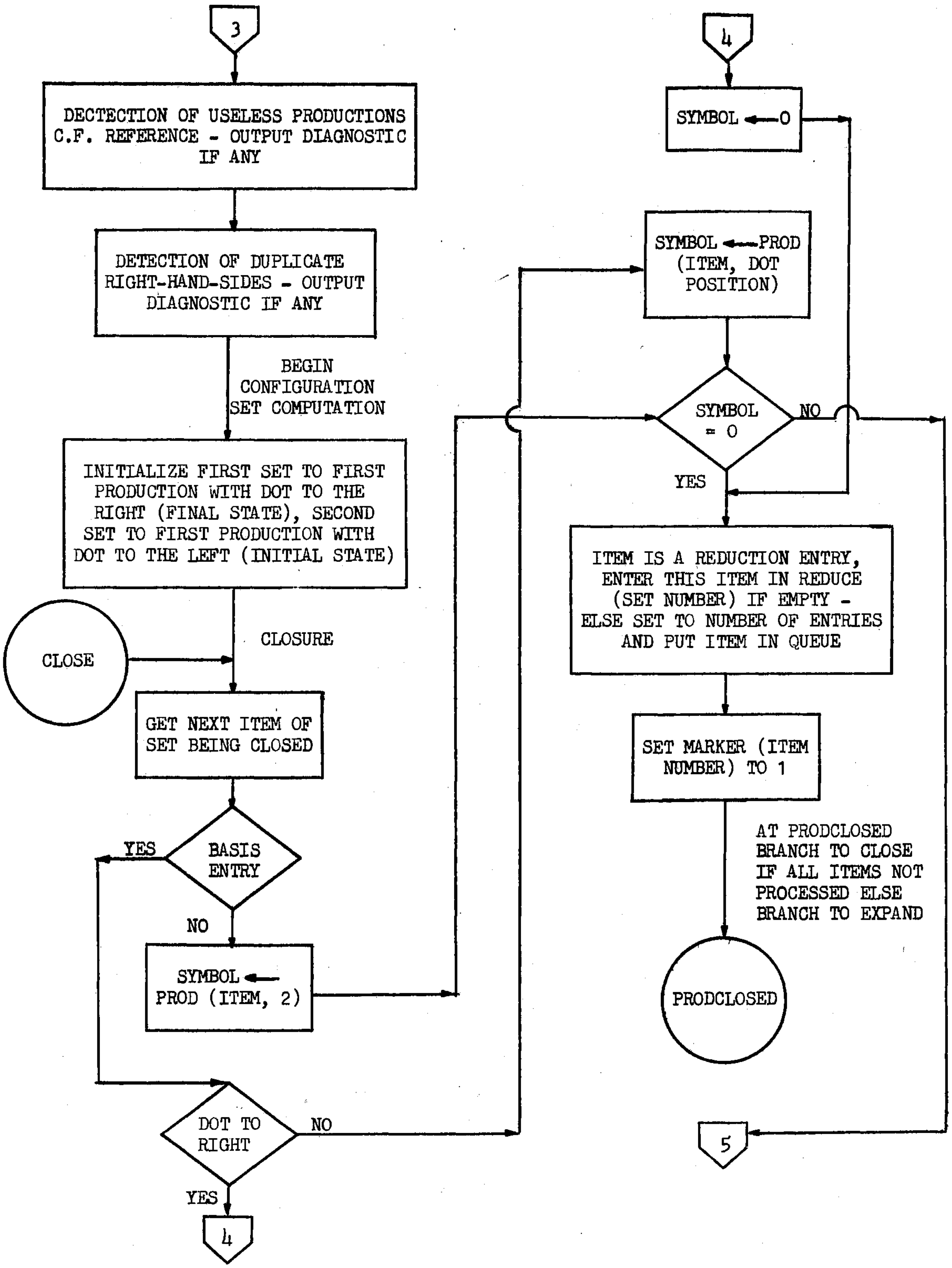
```
            RL(HBOUND(RL,1)),RL(C)=0 ;
            LL=0 ;
            TAG='00'B ;
            CCUNT=0;
            RETURN ;
      END INITIAL;
END BSTSLR;
      END THE_WHOLE_THING;
         GC TC REUSABLE;
ERR01:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - IN INPUT PARMAMETERS---')(A);
         GC TC REUSABLE;
ERR02:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - IN INPUT ENCODE SECTION---')(A);
         GC TC REUSABLE;
ERR03:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - INPUT PRODUCTION PART TOO SHORT---')(A);
         GO TO REUSABLE;
ERR04:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - MISSING PRODUCTION PUNCTUATION---')(A);
         GO TO REUSABLE;
ERR05:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - INPUT PRODUCTION ERROR, PROBABLY LHS NOT ',
            'CONTIGUOUS')(2 A);
         GO TO REUSABLE;
ERR06:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - IN DUBUG SECTION---')(A);
         GC TC REUSABLE;
ERR07:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - IN LR(0) SECTION---')(A);
         GC TC REUSABLE;
ERR08:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - IN SLR(1) SECTION---')(A);
         GC TC REUSABLE;
ERR09:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - UNSOLVABLE INADEQUATE STATE---')(A);
         GO TO REUSABLE;
ERR10: PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - OVERFLOW OF REDUCTION QUEUE---')(A);
         GO TO REUSABLE;
ERR11:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - CONFIGURATION SET OVERFLOW---')(A);
         GO TO REUSABLE;
ERR12:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - BASIS SET OVERFLOW---')(A);
         GO TO REUSABLE;
ERR13:   PUT FILE(PRINT) SKIP EDIT
            ('---ERROR - NUMBER OF SETS EXCEEDED---')(A);
         GO TO REUSABLE;
      END REUSABLE;
ENDMAIN:
END SLR1;
```

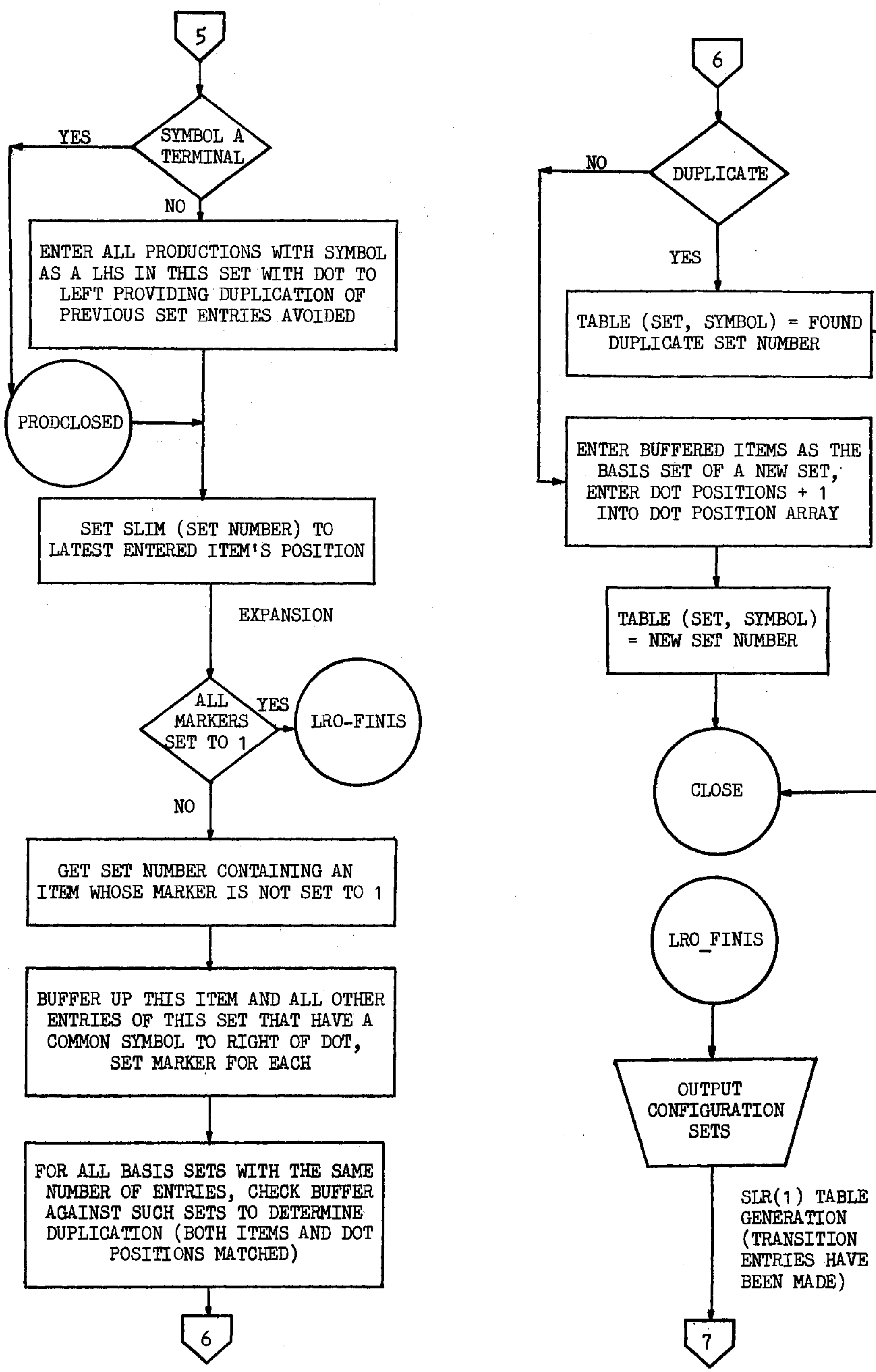
APPENDIX D

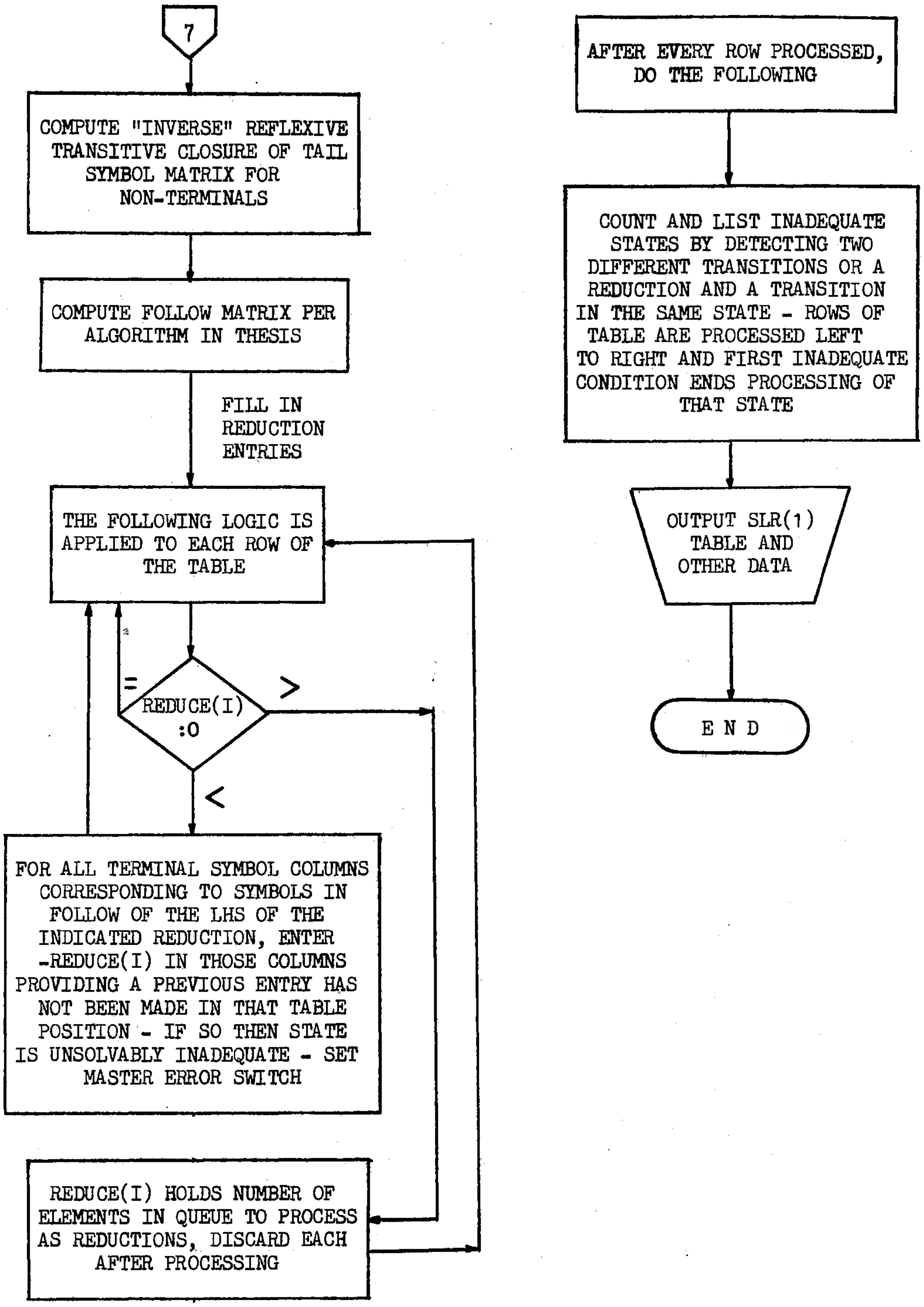
LOGIC BLOCK DIAGRAM


START
REUSABLE
READ INPUT PARAMETERS FROM FILE PARMIN
EXIT ON EOF
OUTPUT HEADING
SET DEFAULT PARAMETERS IF NECESSARY
BEGIN INPUT SECTION
80 BYTE RECORD FROM FILE PRODIN
EOF
ENDINPUT
SET BYTES #73-80 TO NON-BLANKS
NEXTSYM
SPIN THROUGH CONSECUTIVE BLANKS
END OF RECORD
1
1
NAME ← CONSECUTIVE NON-BLANKS
INSERT (SEARCH) NAME IN SYMBOL TABLE
PUNCTUATION
COMMA
SEMI-COLON
COLON
PERIOD
COMMA
FILL NEXT COLUMN OF PRESENT ROW OF PROD
NEXTSYM


SEMICOLON
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
NEXTSYM
COLON
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
PERIOD
2
2
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
ENDINPUT
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
3


3
DECTECTION 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
CLOSURE
GET NEXT ITEM OF SET BEING CLOSED
BASIS ENTRY
YES
NO
SYMBOL ← PROD (ITEM, 2)
DOT TO RIGHT
NO
YES
4
4
SYMBOL ← 0
SYMBOL ← PROD (ITEM, DOT POSITION)
SYMBOL = 0
NO
YES
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
5


5
SYMBOL A TERMINAL
YES
NO
ENTER ALL PRODUCTIONS WITH SYMBOL AS A LHS IN THIS SET WITH DOT TO LEFT PROVIDING DUPLICATION OF PREVIOUS SET ENTRIES AVOIDED
PRODCLOSED
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)
6
6
DUPLICATE
NO
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)
7


7
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
REDUCE(I) :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
E N D

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.