A compiler generating system for a table driven compiler using automatic language expansion

Philip Mason Mills

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
MILLS, Philip Mason, 1936–
A COMPILER GENERATING SYSTEM FOR A TABLEDrIVEN
COMPILER USING AUTOMATIC LANGUAGE EXPANSION.

Iowa State University, Ph.D., 1971
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan
A compiler generating system for a table driven compiler using automatic language expansion

By

Philip Mason Mills

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1971
PLEASE NOTE:

Some Pages have indistinct print. Filmed as received.

UNIVERSITY MICROFILMS
# Table of Contents

<table>
<thead>
<tr>
<th>CHAPTER I. INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages and Recognizers</td>
<td>1</td>
</tr>
<tr>
<td>Definitions and Terminology</td>
<td>3</td>
</tr>
<tr>
<td>Objectives</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER II. TRANSITION MATRIX COMPILER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanner</td>
<td>11</td>
</tr>
<tr>
<td>Recognizer</td>
<td>17</td>
</tr>
<tr>
<td>Semantic Processor</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER III. RECOGNIZER FOR THE EXPANDABLE COMPILER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Construction of an Expanded Operator Grammar</td>
<td>33</td>
</tr>
<tr>
<td>Finding the $f$ Function</td>
<td>37</td>
</tr>
<tr>
<td>Example of Constructing an EOG</td>
<td>48</td>
</tr>
<tr>
<td>Construction of the Recognizer</td>
<td>53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER IV. PROPERTIES OF THE EXPANDED OPERATOR GRAMMAR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>The Eleven Forms of a Production in the EOG</td>
<td>78</td>
</tr>
<tr>
<td>Conditions for an Unambiguous EOG</td>
<td>78</td>
</tr>
<tr>
<td>Relationship Between EOG and Original OG</td>
<td>86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER V. IMPLEMENTATION</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Construction of the Tables</td>
<td>100</td>
</tr>
<tr>
<td>Expanding the Compiler</td>
<td>101</td>
</tr>
<tr>
<td>Stretching the Conditions</td>
<td>106</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER VI. CONCLUSIONS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>108</td>
</tr>
</tbody>
</table>
CHAPTER I. INTRODUCTION

There are many levels of understanding in the solution of a problem on a computer, from the flow chart of an algorithm down to the circuit theory or flip-flop level. It is important that each level be understandable in terms of its underlying level. The idea of level has led to the development of high level computer languages so that computer users could state problems in a language more natural to them rather than the computer's machine instructions. Problem oriented languages free the problem analyzer from machine considerations, so that he only has to consider the procedure for solving his problem.

A compiler is a computer program which translates instructions in a computer language into basic machine instructions. The compiler accepts an input program in one language and outputs basic machine language, with the requirement that there must be one single common meaning. This requires the compiler writer to have a clear understanding of how each statement of the higher level language will be mapped into a set of basic machine instructions.

As the applications of computers become more and more diverse, programming language requirements are becoming continually more diverse. It is highly beneficial to the users and computer manufacturers if users are supplied with language facilities appropriate to their problem area. There are, in
fact, a number of problem areas, such as, scientific, data processing information retrieval, symbol manipulation, system programming, etc. Each problem area has its data structures, operations on these structures, and vocabularies for writing languages. Given that a wide set of language facilities are to be provided, the question is, how?

Three different approaches to provide such facilities are discussed here.

One approach is to provide a large variety of different programming languages, each with its own compiler and run time routines. This approach is losing favor because of its high cost and the difficulties encountered where applications overlap from one problem area to another.

The second approach now in use is to provide a single language which has all the facilities which might reasonably be required for any problem. The idea is that the user will not bother with the parts in which he has no interest. PL/1 (11) is perhaps the best example of this kind of approach. However, this approach has a number of drawbacks. First, because of the large size, the overhead associated with learning the language, compiling text, writing the compiler, maintenance, and modification of the language is high. In regard to learning the language, it is hard for a user to know which facilities are useful to him without some knowledge of all the facilities in the language. Second, the limited scope would become more of a drawback as application areas become more
diverse. For example, PL/1 (11) does not have features appropriate for simulation and lacks pattern replacement facilities.

The third approach which is gaining favor in compiler writing is the use of automatic construction in building compilers. This approach consists of one or more programs that accepts a description of a language and produces a part or all of a compiler as its output. Automatic construction of parts of a compiler means less work, leaving time for other considerations, such as code optimization.

The main objective of this research is to develop a method of constructing a compiler so that language facilities may be added or changed easily. The automatic construction approach was chosen to meet this objective because of its relative ease in modifying a compiler. This approach requires a formal description of the programming language for which a compiler is to be generated.

Languages and Recognizers

The syntax of a language is a set of specifications which describe the sentences that are permissible in the language. The recognizer for a language is an algorithm for deciding if a string of source elements belongs to a language. The following is a set of specifications (commonly called productions) for describing a simple language.
The five productions above are called a grammar. The purpose of a grammar is to describe all sentences of a language with a reasonable number of rules. The language described by this grammar consists of the following four sentences:

John loves Mary
John loves John
Mary loves John
Mary loves Mary

A language used to describe another language is called a meta-language. The productions are described in a metalanguage. The terms SENTENCE, NOUN, PREDICATE, and VERB are called meta-symbols or nonterminal symbols and John, Mary, and loves are enclosed in apostrophes to show that they are terminal symbols. The grammar has a nonterminal starting symbol which is the symbol SENTENCE in the above grammar. The sentences in the language are generated by replacing the nonterminals on the right hand side of the starting symbol production with their definitions. If there is more than one replacement possible,
a derivation is made for all possible replacements and the
process is repeated until all the symbols on the right hand
side are terminal symbols. This process will result in all
possible sentences in the language. A description of a pro-
gramming language in a metalanguage called Backus Normal Form
(BNF), can be found in the ALGOL 60 Report (15).

Given a grammar the sentences of the language can be
generated by deriving them from the starting symbol. However,
compilers have the opposite problem, that is, given a sentence
x and a grammar construct the derivation of x. The process of
constructing a derivation of x is called parsing, recognizing
or analyzing the sentence. There are two different parsing
strategies called top-down and bottom-up.

The true top-down recognizer is goal-oriented and uses a
trial and error process to match the sentence to be parsed.
A left-to-right parse is described here. The main goal is
the starting symbol. Subgoals are established by first
looking at the right hand side of the production that starts
with the starting symbol. In the grammar of Figure 1 the
first production SENTENCE = NOUN PREDICATE starts with the
starting symbol, therefore, the first subgoal would be NOUN.
If the subgoal is nonterminal, a new subgoal is established
using the right hand side referred to by the present goal.
For NOUN there are two productions, NOUN = 'john' and
NOUN = 'mary'. The process is repeated until the subgoal is
a terminal symbol. If the productions are taken in the order
they appear in the grammar, the nonterminal subgoal is 'john'.
Note that the subgoals are established from the grammar and do not necessarily represent a part of the sentence to be analyzed. At this point an attempt is made to match the first previously unmatched symbol in the sentence. The sentence, mary loves john, is now chosen to be parsed. The subgoal 'john' does not match the first symbol 'mary' in the sentence. At this point it is necessary to back up as many levels as required to obtain an alternative production. New subgoals are established from the alternate production until a terminal symbol is reached. In the example, we back up to NOUN and take the second alternate production NOUN = 'mary'. This gives a new subgoal 'mary' which matches the first symbol in the sentence. We now are able to report success for the subgoal NOUN in the production SENTENCE = NOUN PREDICATE so the next subgoal is PREDICATE. This process is repeated until the entire sentence is matched. The paper by Floyd (6) gives a more complete description of a top-down parse.

A bottom-up recognizer has essentially no long range goals except that of the starting symbol. The string or sentence is searched for substrings which are right parts of productions. They are then replaced by the corresponding left side of the production giving a new string. This is repeated until the string is reduced to the starting symbol. The paper by Feldman and Gries (5) has a more complete description of a bottom-up parse.
Both types of recognizers will normally apply productions which will turn out to be incorrect. This will require a back up to a point where an alternative may be tried. This requires modifying goals and restoring parts of strings. To reduce the number of possibly incorrect reductions some top-down recognizers use a table with look-ahead to see whether the subgoal can start with the initial symbol of the present substring.

Some bottom-up recognizers look at the context around a possible right hand part of a production to decide whether to make a reduction or not.

Definitions and Terminology

Terms that have been used informally are now described in the formal notations of grammars and languages. The word parse in the definitions and throughout this dissertation will refer to a bottom-up parse.

**Definition 1.**

An alphabet \( A \) is a finite nonempty set of symbols. A word over an alphabet \( A \) is a finite sequence (or string) of symbols of \( A \). A word has length (possible zero) and is formed by concatenating symbols of \( A \). The set of all words over an alphabet \( A \) is denoted by \( A^* \).

**Definition 2.**

A language \( L \) is some subset of \( A^* \). Each word of \( A^* \) belonging to the language \( L \) is called a sentence.
**Definition 3.**

A grammar $G$ is a way of describing which subset of $A^*$ belongs to the language $L$.

There are different types of grammars but only one type is defined here.

**Definition 4.**

A context free phrase-structure grammar $G$ is represented by a quadruple $(V_n, V_t, P, U_1)$.

1) $V_n = \text{an alphabet of symbols called nonterminal symbols distinct from alphabet } A$.

2) $V_t = \text{a set of symbols in alphabet } A \text{ called terminal symbols}$.

3) $P = \text{a finite set of productions}$.

4) $U_1 = \text{a starting symbol called the distinguished symbol}$.

A new alphabet $V$ is defined as the union of $V_n$ and $V_t$. The set of all strings over $V$, including the empty string, is denoted by $V^*$. Small Latin letters are used to denote strings in $V^*$. The set of all strings over $V_t$ is denoted by $V_t^*$. Each element $P_i$ of $P$ is called a production and is in the following form $U_1 \rightarrow d_1$ with the following properties.

(1) Each $d_1$ is a nonempty string in $V^*$.

(2) Each $U_1$ is in $V_n$.

(3) There is exactly one $U_1$ which occurs in no $d_1$.

(4) $U_1 \neq d_1$.

$U_1$ is called the left part and $d_1$ the right part of a production.
Definition 5.

Let $P$ be a finite set of productions $P_1, P_2, \ldots, P_n$. $f$ directly produces $h$ (denoted by $f \Rightarrow h$) and conversely $h$ directly reduces into $f$, if and only if there exist strings $e, b$ (possibly empty) such that $f = eUb$, $h = edb$, and the production $U \Rightarrow d$ is an element of $P$.

Definition 6.

$f$ produces $h$ (denoted by $f \Rightarrow^* h$) and conversely $h$ reduces into $f$, if and only if there exist strings $e_0, \ldots, e_n$ such that $f = e_0$, $e_n = h$ and $e_{i-1} \Rightarrow e_i$ ($i = 1, \ldots, n$). $h$ is also said to be a derivation of $f$.

Definition 7.

$s$ belonging to $V^*$ is called a sentential form of $G$ if either $U_1 = s$ or $U_1 \Rightarrow^* s$. Subscripted $s$'s are used to denote sentential forms.

Definition 8.

The set of sentential forms consisting only of terminal symbols are the set of sentences constituting the phrase structure language $L(G)$, that is $L(G) = \{ d | U_1 \Rightarrow^* d, d \in V^*_t \}$.

Given a context free phrase-structure grammar, sentences of the language can be generated by deriving them from the distinguished symbol. Compilers have the opposite problem. Given a sentence $d$ and a grammar $G$ construct a derivation of $d$. This is called parsing, recognizing, or analyzing the sentence.
Definition 9.
A parse of a sentence \( s_0 \) in a language \( L(G) \) into the symbol \( U_1 \) is the application of a sequence of productions \( P_1, P_2, \ldots, P_n \) such that \( P_i \) is of the form \( (U_j \rightarrow d_i) \) and directly reduces \( s_{i-1} = e_1 d_i b_i \) into \( s_i = e_i U_j b_i \) for \( (i = 1, \ldots, n) \) and \( s_n = U_1 \). The \( d_i \)'s are called prime phrases of the \( s_i \)'s (sentential forms).

Definition 10.
A canonical parse is the parse which proceeds strictly from left to right in a sentence and reduces the left most part of a sentence as far as possible before proceeding further to the right. That is, the application of the productions \( P_1, P_2, \ldots, P_n \) to reduce \( s_0 \) into \( U_1 \) is canonical if, and only if, for \( (i = 1, \ldots, n) \) \( d_i \) is not contained in \( e_i \) for all \( k \) greater than \( i \). Every parse has a canonical form (simply rearrange the productions to form a canonical parse).

Definition 11.
If no production \( P_i \) of the phrase structure grammar \( G \) takes the form \( U_i \rightarrow eU_j U_k b \) for some (possibly empty) strings \( e, b \) and nonterminal symbols \( U_j \) and \( U_k \) then \( G \) is called an operator grammar (OG).

Definition 12.
Define a function \( \mathcal{S} \) as follows:
\[
\mathcal{S}(A) = \{ B \mid U_1 \notin B \quad eBAB, (e, b) \in V^* , (B, A) \in V \}
\]
\( \mathcal{S}(A) \) is just the set of symbols which are adjacent and to left of \( A \) in some sentential form.
Definition 13.

Define a function LEFT as follows:

\[
\text{LEFT}(U_i) = \left\{ \{D\mid U_i \not\Rightarrow^* D e, \ e \in \Sigma^*, \ D \in V, \ U_i \in V_n \} \cup U_i \right\}
\]

or

\[
\text{LEFT}(T_i) = T_i, \text{ where } T_i \in V_t
\]

LEFT\((U_i)\) is the set of symbols in \(V\) that can occur on the left end of a derivation of \(U_i\) and \(U_i\) itself.

Definition 14.

Define a function RIGHT as follows:

\[
\text{RIGHT}(U_i) = \left\{ \{D\mid U_i \not\Rightarrow^* e D, \ e \in \Sigma^*, \ D \in V, \ U_i \in V_n \} \cup U_i \right\}
\]

\[
\text{RIGHT}(T_i) = T_i, \text{ where } T_i \in V_t
\]

RIGHT\((U_i)\) is the set of symbols in \(V\) that can occur on the right end of a derivation of \(U_i\) and \(U_i\) itself.

Objectives

There is no absolute measure of performance of a compiler such as efficiency. The efficiency of a compiler depends not only on the ability to conserve both time and space during translation but also depends on the time and space used by the object program. Not all these goals are mutually compatible. For example, to optimize the object code at compile time would require more compile time but would produce faster code at run time.

The use of an extensive error detection and recovery facility to give the user a detailed description of his errors may add additional compile time but improve the efficiency in
debugging the user's program. Since there is no set measure of performance, compiler designers have varied greatly in their choice of compromises. The compromises chosen in a compiler design usually reflect the type of environment in which the compiler is to be used.

Although there may be different design goals in constructing compilers there are functions which all compilers must perform. The compiler can be segregated into three organizationally distinct parts, the lexical analyzer or scanner, the recognizer or parser and the semantic processor or code generator. There have been variations of these three parts used in automatically constructed compilers. As Cheathem and Standish (3) point out some of the techniques used in some compilers have proven to be more efficient than others. It is desirable, therefore, to use those techniques which have proven to be most efficient in compiler writing.

The term compiler-compiler is a loose term that is often used to refer to a system that uses automatic construction in implementing a compiler. The compiler-compiler accepts a description of a language and produces all or part of a compiler as its output. The compiler-compiler accepts two types of inputs as follows:

(1) A description of the source language syntax, written in a metasyntactic language.

(2) A description of relationship between the source language syntax and its semantics.
Semantics refer to the meaning to be given to parts of the source language. The semantics may be expressed in terms of basic machine language up to a high level semantic language. The compiler-compiler builds the recognizer for the compiler, based on the syntax description and for that reason is said to be a syntax-directed compiler. The recognizer calls the semantic processes as a sentence is being parsed.

From the description of compiler-compilers it is apparent that all compiler-compilers could meet the objective of being able to easily change or expand language facilities, therefore, the objective will be restated with additional goals.

The objective is to design a compiler generating system with the following design criteria.

1. To be able to easily change or expand language facilities provided by the compiler produced from the system.

2. To have the system produce a compiler that will operate at speeds that are competitive with hand coded ad hoc compilers.

3. To leave to the user the freedom to develop as an extensive error detection, error recovery and error message facility as he desires.

4. To leave to the user the freedom to select the level of code optimization as he desires.

To meet these goals the design of the scanner chosen for the compiler generating system has the characteristics de-
scribed by Cheathem and Standish (3) as the most efficient. The scanner is described in detail in Chapter II. Also an efficient recognizer is to be developed that can be automatically constructed. The recognizer uses a deterministic method for parsing sentences. The deterministic approach uses the top element in a pushdown stack and the next incoming symbol to make decisions for parsing a sentence. Deterministic pushdown automata is described in more detail by Feldman and Gries (5). The deterministic approach eliminates the problem of back up and testing alternates found in many parsers. Chapter III is a complete description of the recognizer which is the main contribution of this dissertation.

In the last decade several families of compiler-compilers have been built, each having one or more successors. The following families, Brooker-Morris (1), Meta (19), TMG (15), and COGENT (18) build compilers that use the top-down method for parsing. The more efficient of these compilers use speed-up methods as follows:

(1) Order the alternatives, the most frequently used first.

(2) See if the head of the string belongs to the $\text{LEFT}(N)$ where $N$ is the subgoal.

(3) If there are productions with their right parts as follows, $X_1X_2X_3\ldots X_kU$ and $X_1X_2X_3\ldots X_kV$, then on
failing to recognize U, do not back up and attempt to recognize V.

The following families of compiler-compilers, FSL (4), TGS (17) build compilers with bottom-up recognizers. FSL uses a Floyd-Evans reduction technique along with a high level semantic language. Some of the compiler-compilers are written in a high level language designed for that use. In general, all of these compiler-compilers are faced with the problem of testing alternatives in parsing a sentence. A more detailed description of compiler-compilers is given by Feldman and Gries (5).

At this point in time no compiler-compiler has been constructed which uses a deterministic recognizer. However, compilers have been constructed by hand which use a deterministic method, for example, Grau (7) and Campbell (2). The recognizers are implemented using a transition matrix \( M \), where each possible state is represented by a row and each terminal symbol is represented by a column. At each step in the parse the matrix element \( M_{ij} \) determines a subroutine to control the parse.

The compiler to be designed to meet the previously stated objectives is named Expandable Compiler. The Expandable Compiler uses the same scanner and semantic processor as described for the transition matrix compiler in Chapter II. The recognizer for the Expandable Compiler is discussed in
Chapter III, but this section can best be understood by first reading the description for the transition matrix recognizer in Chapter II.

A compiler was constructed for Ten Statement FORTRAN (13) with only scalar variables using the design discussed in Chapters II and III. Parts of this compiler are used in Chapters II and III for illustrative purposes. The compiler was written in PL/1 (11) and outputs code in IBM 360 Assembly Language (12). A high level language like PL/1 was used because of its ability to illustrate compiler functions.
CHAPTER II. TRANSITION MATRIX COMPILER

The transition matrix compiler here is separated into three organizationally distinct parts, the scanner, the recognizer and semantic processor. Although the functions of each of these subparts are to be discussed separately, in actual use, there is a great deal of interaction between the parts. In general, the control is in the recognizer which behaves like a main program calling the scanner and semantic routines when their functions are required.

Scanner

The scanner is that part of the compiler which reads in the source program characters and constructs the elements of the source language. The elements to be constructed or recognized are the identifiers, reserved words, constants, operators and punctuation. The scanner also performs the conversion of these elements into an internal machine form, in this case, an integer value. There are some good reasons for separating the scanner from the recognizer.

The grammar describing the syntax of the source elements is much simpler than that of the language. This fact allows for simpler and more efficient parsing techniques to be used. To have an efficient compiler requires an extremely efficient scanner because it is not uncommon for 50% or more of the compile time to be spent in the scanner. Some of the symbol proce-
essing can be better done in an ad hoc manner, such as the
FORTRAN statement DO 15 1=. It is necessary to look at more
symbols to determine if this is a DO statement or an assign­
ment statement. Also, fixed fields, comments and continu­
ation statements in FORTRAN are best handled in an ad hoc
manner.

This part of the compiler is to be hand coded by the pro­
grammer constructing the compiler. However, a formal approach
is to be used for the scanner because the need for ease in
changing the scanner at a later date and because it results
in an efficient scanner. As an example, a Ten Statement
FORTRAN (13) scanner is used to describe the approach. The
productions in Figure 2 describe the source elements in Ten
Statement FORTRAN.

\begin{verbatim}
INT = INT D
REAL = INT '.' INT
REAL = INT '.' INT
REAL = INT
REXP = REAL 'E' S INT
REXP = REAL 'E' INT
D = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
S = '+' | '-'
EX = '*' | '
MUL = '*'
DEL = S | Y
Y = '/' | '!' | '(!)' | '!' | '!',
ID = L
ID = ID L
ID = ID D
RB = '!' | L L '!
L = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
L = 'K' | 'L' | 'M' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
L = 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '$'
INT = D
\end{verbatim}

Figure 2. Productions for source elements
The | in Figure 2 represents the word OR and specifies an alternate definition for the nonterminal symbol on the left of the production.

Figure 3 shows a state diagram for recognizing the FORTRAN source elements from the incoming character string in accordance with the productions of Figure 2. The names used in the state diagram and productions are described at the bottom of Figure 3. The state diagram is obtained from the productions in an ad hoc manner and is not unique. However, in the construction of the state diagram an attempt should be made to minimize the number of states.

The state diagram allows one or more blanks between the source elements. In the state diagram the paths to the OUTS are taken for all other legal characters except those defined at that state. For example, if the system is in the state REAL, then OUT2 is taken for all symbols except a digit or the letter E. OUT1, OUT2, OUT3, OUT6, and OUT7 all would cause the scanner to pass a corresponding single integer to the recognizer. OUT5 requires a search of a name table to tell if the ID is a reserved word or an identifier. A different integer value is passed to the recognizer corresponding to each reserved word recognized. A single integer value is passed to the recognizer for all identifiers. OUT8 passes a different integer value to the recognizer, depending on the single character detected. OUT4 also passes a different integer
\[ D = \text{digit} \quad L = \text{letter} \quad E = \text{itself} \quad b = \text{blank} \]
\[ S = \div \quad \ast = \text{ST} = \text{start} \]
\[ \text{DEL} = S, Y \quad \text{Where } Y = \div, (,), =, , \text{comma} \]

<table>
<thead>
<tr>
<th>OUT</th>
<th>ABBREVIATION</th>
<th>RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT1</td>
<td>REXP</td>
<td>Real constant with exponent</td>
</tr>
<tr>
<td>OUT2</td>
<td>REAL</td>
<td>Real constant</td>
</tr>
<tr>
<td>OUT3</td>
<td>INT</td>
<td>Integer constant</td>
</tr>
<tr>
<td>OUT4</td>
<td>RB</td>
<td>Relational operator</td>
</tr>
<tr>
<td>OUT5</td>
<td>ID</td>
<td>Reserved word or identifier</td>
</tr>
<tr>
<td>OUT6</td>
<td>EX</td>
<td>Exponential symbol</td>
</tr>
<tr>
<td>OUT7</td>
<td>MUL</td>
<td>Multiply symbol</td>
</tr>
<tr>
<td>OUT8</td>
<td>DEL</td>
<td>Operators and delimiters</td>
</tr>
</tbody>
</table>

Figure 3. State diagram for scanner
value to the recognizer, depending on the relational operator recognized.

The state diagram is implemented using a transition matrix where each possible state is represented by a row and each type of source character is represented by a column. Figure 4 shows the transition matrix for the state diagram in Figure 3. The letter A at the top of the right hand column represents all source characters other than the ones described in the grammar. The names in the transition matrix are names of routines to be executed for controlling the scanning process. The control routines are described in PL/1 (11) in Figure 5. LABV is assumed to be a PL/1 label vector representing the matrix in Figure 4. A vector is used in place of a matrix because of the simpler computation required to locate an entry. The integer variable IS is used as a pointer for the states and IX is an integer variable used as a pointer for the type of incoming character symbol. The following is a description of how the incoming character symbol is converted to the integer values shown at the top of the matrix in Figure 4.

The bit pattern used to encode the symbol is used as an integer number. This integer number is used as a subscript to a vector containing the proper integer subscript for the scanner's transition matrix. In Figure 5, I represents a pointer to the card column of the source statement being scanned. Because FORTRAN statements do not have a termin-
<table>
<thead>
<tr>
<th>IX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>SCAN</td>
<td>SYM</td>
<td>SCAN</td>
<td>RT8</td>
<td>RT1</td>
<td>RT6</td>
<td>RT8</td>
<td>RT9</td>
<td>RT12</td>
<td>RT12</td>
</tr>
<tr>
<td>9</td>
<td>RT1</td>
<td>INT</td>
<td>OUT3</td>
<td>OUT3</td>
<td>SCAN</td>
<td>RT2</td>
<td>OUT3</td>
<td>OUT3</td>
<td>OUT3</td>
<td>OUT3</td>
</tr>
<tr>
<td>18</td>
<td>RT2</td>
<td>REAL</td>
<td>OUT2</td>
<td>OUT2</td>
<td>SCAN</td>
<td>OUT2</td>
<td>RT3</td>
<td>OUT2</td>
<td>OUT2</td>
<td>OUT2</td>
</tr>
<tr>
<td>27</td>
<td>RT3</td>
<td>EXP1</td>
<td>RT4</td>
<td>ERROR</td>
<td>RT5</td>
<td>ERROR</td>
<td>ERROR</td>
<td>ERROR</td>
<td>RT4</td>
<td>ERROR</td>
</tr>
<tr>
<td>36</td>
<td>RT4</td>
<td>EXP2</td>
<td>ERROR</td>
<td>ERROR</td>
<td>RT5</td>
<td>ERROR</td>
<td>ERROR</td>
<td>ERROR</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>45</td>
<td>RT5</td>
<td>REEXP</td>
<td>OUT1</td>
<td>OUT1</td>
<td>SCAN</td>
<td>OUT1</td>
<td>OUT1</td>
<td>OUT1</td>
<td>OUT1</td>
<td>ERROR</td>
</tr>
<tr>
<td>54</td>
<td>RT6</td>
<td>UND</td>
<td>ERROR</td>
<td>RT7</td>
<td>RT2</td>
<td>ERROR</td>
<td>RT7</td>
<td>ERROR</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>63</td>
<td>RT7</td>
<td>RB1</td>
<td>ERROR</td>
<td>SCAN</td>
<td>ERROR</td>
<td>RT10</td>
<td>SCAN</td>
<td>ERROR</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>72</td>
<td>RT8</td>
<td>ID</td>
<td>OUT5</td>
<td>SCAN</td>
<td>SCAN</td>
<td>OUT5</td>
<td>SCAN</td>
<td>OUT5</td>
<td>OUT5</td>
<td>OUT5</td>
</tr>
<tr>
<td>81</td>
<td>RT9</td>
<td>MUL</td>
<td>OUT7</td>
<td>OUT7</td>
<td>OUT7</td>
<td>OUT7</td>
<td>RT11</td>
<td>OUT7</td>
<td>OUT7</td>
<td>ERROR</td>
</tr>
<tr>
<td>90</td>
<td>RT10</td>
<td>RB</td>
<td>OUT4</td>
<td>OUT4</td>
<td>OUT4</td>
<td>OUT4</td>
<td>OUT4</td>
<td>OUT4</td>
<td>OUT4</td>
<td>ERROR</td>
</tr>
<tr>
<td>99</td>
<td>RT11</td>
<td>EX</td>
<td>OUT6</td>
<td>OUT6</td>
<td>OUT6</td>
<td>OUT6</td>
<td>ERROR</td>
<td>OUT6</td>
<td>OUT6</td>
<td>ERROR</td>
</tr>
<tr>
<td>108</td>
<td>RT12</td>
<td>DELS</td>
<td>OUT8</td>
<td>OUT8</td>
<td>OUT8</td>
<td>OUT8</td>
<td>OUT8</td>
<td>OUT8</td>
<td>OUT8</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

**Figure 4.** Transition matrix for FORTRAN source elements
SCAN:  I = I + 1;
          IF I > 72 THEN RETURN (STOPS)
          IY = UNSPEC (C(I));
          IX = VEC(IY);
          GO TO LABV(IX+IS);
          RT1:  IS = 9;  K1 = I;  GO TO SCAN;
          RT2:  IS = 18;  GO TO SCAN;
          RT3:  IS = 27;  GO TO SCAN;
          RT4:  IS = 36;  GO TO SCAN;
          RT5:  IS = 45;  GO TO SCAN;
          RT6:  IS = 54;  K1 = I;  GO TO SCAN;
          RT7:  IS = 63;  GO TO SCAN;
          RT8:  IS = 72;  K1 = I;  GO TO SCAN;
          RT9:  IS = 81;  GO TO SCAN;
          RT10: IS = 90;  GO TO SCAN;
          RT11: IS = 99;  GO TO SCAN;
          RT12: IS = 108; IF = IY;  GO TO SCAN;

Figure 5. Control routines for scanner
ating symbol, an integer representing a terminating symbol is returned to the recognizer at the end of the card.

The whole scanner is shown in the Appendix as the PL/1 procedure SCANR in the Ten Statement FORTRAN compiler. The reader may wish to look at SCANR's implementation of the OUT routines. In addition to the OUT routines passing an integer value to the recognizer, it also may be required to store information that will be used by semantic routines. In routines OUT1, OUT2, and OUT3, the constants and their types recognized by the scanner in the source stream are stored for use by semantic routines. In OUT8 single character symbols, which are also source elements, are converted to an integer for the recognizer by also using their encoded bit pattern as an integer subscript to a vector. The scanner never returns to the recognizer until it is one character symbol past the source element it has recognized. This requires the storage of the single symbols in variable IP.

The scanner only needs to be changed if new source elements are added to the language. If new elements are added, the following steps should be followed.

Step 1. Draw new horizontal lines on state diagram for elements.

Step 2. Add new rows and columns to transition matrix as needed.

Step 3. Implement new control and OUT routines as needed.
Recognizer

The recognizer analyzes the incoming strings of source elements to determine if they are permissible sentences in the language. If the sentence is not legal then the recognizer branches to an error routine. The error routine must analyze the problem and recover from the error. Error messages must also be printed to help the programmer correct his program. As the sentence is analyzed the recognizer branches to semantic routines that generate code, depending on the sequence of the source elements recognized.

An example of a transition matrix is given here for the grammar shown in Figure 6.

\[
\begin{align*}
A &= \#ID '=' E ';' \\
E &= E '+' T \\
E &= E '-' T \\
E &= T \\
T &= T '*' P \\
T &= P \\
P &= '(' E ')' \\
P &= \#ID
\end{align*}
\]

Figure 6. Grammar for arithmetic assignment statement

The $\#$ in Figure 6 represents the fact that the source element is recognized by the scanner, in this case an identifier.
A transition matrix for the grammar in Figure 6 is shown in Figure 7 along with its control routines in Figure 8. The parsing or recognizing of a sentence in the source program may be analyzed by a transition matrix where the rows represent all possible control states and the columns represent all possible incoming source elements. The recognizer uses a stack called STACK with a pointer S along with the incoming source element variable J. The integer values for J are provided by the scanner and correspond to the columns of the transition matrix. The transition matrix controls the process by using the top element in the state STACK and J as row and column subscripts to branch to the routine named in the matrix at that location. Initially, the first location in STACK and S are set equal to one and J set equal to the first incoming source element. The routines in Figure 8 are written in PL/1 and are used to add and delete states on the state STACK along with obtaining the next incoming source element. Figure 9 shows a trace of a parse of A = B+C*(D-E);.

Each element of the matrix was filled in by determining from the grammar what the next incoming source elements could be in that state and what corresponding actions should be performed. The actions to be performed are based on the operator priorities discussed by Samelson and Bauer (20). Many of the elements of the transition matrix are empty, since the corresponding combinations of state and incoming source elements cannot occur during the processing of a valid sentence in the
Incoming Source Element

<table>
<thead>
<tr>
<th>State</th>
<th>ID</th>
<th>=</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td>R9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
<td>R6</td>
<td>R7</td>
<td>R8</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>7</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>8</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>9</td>
<td>R6</td>
<td>R7</td>
<td>R8</td>
<td>R5</td>
<td>R2</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7. Transition matrix for assignment statements**

**CONTROL:**  \( I = \text{STACK}(S); \)  
Go to LABM(1, J);  
R1: \( \text{STACK}(S)=4; S=S+1; \text{STACK}(S)=2; J=\text{SCANR}; \)  
Go to CONTROL;  
R2: \( S=S+1; J=\text{SCANR}; \)  
Go to CONTROL;  
R3: \( \text{STACK}(S)=5; S=S+1; \text{STACK}(S)=3; J=\text{SCANR}; \)  
Go to CONTROL;  
R4: \( S=1; \text{STACK}(S)=1; \)  
Go to FINISHED;  
R5: \( S=S+1; \)  
Go to CONTROL;  
R6: \( S=S+1; \text{STACK}(S)=6; S=S+1; \text{STACK}(S)=3; J=\text{SCANR}; \)  
Go to CONTROL;  
R7: \( S=S+1; \text{STACK}(S)=7; S=S+1; \text{STACK}(S)=3; J=\text{SCANR}; \)  
Go to CONTROL;  
R8: \( S=S+1; \text{STACK}(S)=8; S=S+1; \text{STACK}(S)=3; J=\text{SCANR}; \)  
Go to CONTROL;  
R9: \( \text{STACK}(S)=9; S=S+1; \text{STACK}(S)=3; J=\text{SCANR}; \)  
Go to CONTROL;  

**Figure 8. Routines for transition matrix**
<table>
<thead>
<tr>
<th>SOURCE</th>
<th>STATE</th>
<th>ROUTINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT</td>
<td>J</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>(</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>)</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>)</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>;</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 9. Trace of the compilation of A = B+C*(D-E);
language. Error subroutines may be included for as many of the empty elements as desired.

An advantage of using the transition matrix is speed, because no searching is required and each subroutine specifies exactly what is to be done. Another advantage is that error recovery can be incorporated very easily. With other parsing techniques, error recovery has proven to be a difficult problem. The one disadvantage with this technique is the space used. For large languages the matrix and the number of routines becomes large. Examples of transition matrix compilers for large languages can be found in works by Campbell (2) and Grau (7).

Semantic Processor

The function of the semantic processor is to generate code in an appropriate language, usually machine instructions. Some compilers have the processor produce an intermediate form such as, Polish strings, quadruples, or triples, which are then used to generate code. The use of an intermediate form allows the compiler writer to perform optimization on the output code. Code optimization refers to the process of rearranging and changing operations in the program being compiled in order to produce a more efficient object program. Cheathem and Standish (3) list some 26 optimization techniques. Gries (10) has a good section on intermediate forms and code optimization with references to earlier works.
The semantic processor is made up of a number of semantic routines which use common data structures, such as tables, vectors, stacks and variables. Each routine may contain two parts, one to manipulate the data structures at compile time and the second to produce code for run time actions. One of the major tasks for the compiler's semantic routines is to manage run time storage. A good description of code production and run time storage can be found for FORTRAN in a book by Lee (14) and for ALGOL in a book by Grau, Hill and Langmaack (8). Gries (9) gives a good general description of compile time data structures and run time storage organization.

As an example, the semantic routines STACK, ADD, SUB, MUL, and STORE, shown in the Ten Statement FORTRAN compiler of the Appendix can be used with the recognizer of Figures 8 and 9 to show the compilation process. The semantic routine STACK places the name and type of identifiers in stacks STKID and STKTYPE. The routine ADD generates code that adds the two top elements of the stack STKID, if they are the same type, and places the result in a temporary location. The top two elements are deleted from the stack and the name of the temporary location is placed on top of the stack. The routines SUB and MUL are similar to ADD. The routine STORE generates code that places the value in the location specified by the name at the top of the stack in the location specified by the name, one location down from the top of the stack. The top two elements of the stack are then deleted. Figure 10 shows
the same transition matrix as Figure 7 with the semantic routines added. The following is a description of how the compilation process would work.

The compiler would first branch to the semantic routine shown in Figure 10, depending on the value at the top of the state stack and incoming source element. All of the semantic routines return to the first statement of the control routine in Figure 8 which is then executed.

Figure 11 shows a trace of the compilation of \( A = B + C \times (D - E) \); which is the same as Figure 9, except that it shows the contents of the ID Stack and the semantic routines used. The function \( \text{SUB}(D,E,T_1) \) in Figure 11 means subtract \( E \) from \( D \) and store the result in \( T_1 \). \( \text{MUL} \) and \( \text{ADD} \) are similar to \( \text{SUB} \). Store \( (A,T_1) \) means store the value of \( T_1 \) in \( A \).

The compiler writer using a transition matrix design will find the use of stacks most useful. If information is found in the incoming source elements which is needed at a later point in the process it must be stored. A stack is used if information is needed in the reverse order of its appearance. This saves the compiler from the problem of having to search a table for the information. The ID stack shown in Figure 11 is an example of using a stack.
### Incoming Source Element

<table>
<thead>
<tr>
<th>STATE</th>
<th>ID</th>
<th>=</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>(</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R9</td>
</tr>
<tr>
<td></td>
<td>STACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>R5</td>
<td></td>
<td>R6</td>
<td>R7</td>
<td>R8</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td></td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>SUB</td>
</tr>
<tr>
<td>6</td>
<td>R5</td>
<td></td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td></td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>SUB</td>
</tr>
<tr>
<td>7</td>
<td>R5</td>
<td></td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td></td>
<td>SUB</td>
<td>SUB</td>
<td>SUB</td>
<td>SUB</td>
<td>MUL</td>
</tr>
<tr>
<td>8</td>
<td>R5</td>
<td></td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R2</td>
</tr>
<tr>
<td></td>
<td>MUL</td>
<td></td>
<td>MUL</td>
<td>MUL</td>
<td>MUL</td>
<td>MUL</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>R6</td>
<td>R7</td>
<td>R8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10. Transition matrix with semantic routines**

<table>
<thead>
<tr>
<th>Source Element</th>
<th>State Stack</th>
<th>ID Stack</th>
<th>Semantic Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>EMPTY</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>1,2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>4,5,3</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>4,5</td>
<td>A,B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4,5,6,3</td>
<td>A,B</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>4,5,6</td>
<td>A,B,G</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>4,5,6,8,3</td>
<td>A,B,C</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>4,5,6,8,9,3</td>
<td>A,B,C,D</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>4,5,6,8,9</td>
<td>A,B,C,D</td>
<td>SUB(D,E,T1)</td>
</tr>
<tr>
<td>ADD</td>
<td>4,5,6,8,9,7</td>
<td>A,B,C,D</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>4,5,6,8,9,7</td>
<td>A,B,C,D</td>
<td></td>
</tr>
<tr>
<td>ST0RE</td>
<td>4,5,6,8,9</td>
<td>A,B,C,D</td>
<td>ADD(B,T1,T1)</td>
</tr>
<tr>
<td></td>
<td>4,5</td>
<td>A,T1</td>
<td>STORE(A,T1)</td>
</tr>
<tr>
<td>;</td>
<td>1</td>
<td>EMPTY</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>4</td>
<td>EMPTY</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11. Trace of compilation showing semantic routines**
CHAPTER III. RECOGNIZER FOR THE EXPANDABLE COMPILER

The design of the Expandable Compiler (EC) is similar to the design of the transition matrix compiler (TMC) discussed in Chapter II. The EC design uses the same scanner and semantic section as the TMC design but uses a different approach for the construction of the recognizer. The construction of the transition matrix for the EC recognizer is based on an algorithm for deriving the transition matrix from productions defining the language. The advantage of this approach is that the algorithm for deriving the transition matrix can be automated, making it easy to modify the recognizer to accept new additions to a language.

The earlier transition matrix compilers were created in an ad hoc manner as in Grau (7) and Campbell (2). Gries (9) described an algorithm for deriving the transition matrix from productions defining a language. Gries's algorithm expanded the productions until the maximum number of symbols on the right side of a production were three. He used the first and last symbol on the right of a production in the transition matrix and had the matrix control routines check the center nonterminal symbol. An algorithm is given in this chapter to expand the production until the maximum number of symbols on the right side of a production equals two. This simplifies the recognizer and allows a one to one relationship to be set up between states of the transition
matrix and the symbols on the right of a production.

A simple example of expanding a grammar is presented to illustrate the approach for creating a transition matrix from a grammar. In the production of the grammar in Figure 12 the U's represent nonterminal symbols and the T's represent terminal symbols.

\[ U_1 \rightarrow U_2T_1T_2 \]
\[ U_2 \rightarrow U_3T_3T_4 \]
\[ U_3 \rightarrow T_5T_6 \]

Figure 12. Simple grammar for transition matrix

New nonterminal symbols \( Y_1 \) are added to the grammar and the productions below are created.

\[ Y_1 \rightarrow U_2T_1 \]
\[ Y_2 \rightarrow U_3T_3 \]
\[ Y_3 \rightarrow T_5 \]

These new productions are added to the original grammar and the expanded grammar is rewritten in Figure 13.
1. $U_1 \rightarrow Y_1 T_2$
2. $U_2 \rightarrow Y_2 T_4$
3. $U_3 \rightarrow Y_3 T_6$
4. $Y_1 \rightarrow U_2 T_1$
5. $Y_2 \rightarrow U_3 T_3$
6. $Y_3 \rightarrow T_5$

Figure 13. Expanded grammar for transition matrix

Figure 14 shows the transition matrix created for the grammar in Figure 13. The state $Y_0$ must be added in the matrix for initializing the state stack to be used in applying the productions of Figure 13.

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$T_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>$Y_1$</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Y_2$</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Y_3$</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U_2$</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U_3$</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14. Transition matrix for expanded grammar

The numbers in the transition matrix refer to the productions of Figure 13 to be applied in the parsing of a sentence. The numbers are also used to refer to control
routines to be used in the application of the particular production shown in Figure 15. The control routines use a state variable $S$ and $J$ equals the next incoming source element. LABM is a label matrix which can use names as subscripts.

CONTROL: GO TO LABM (S,J);

1. $S = U_1$; GO TO FINISHED;
2. $S = U_2$; $J = $ NEXTELEMENT; GO TO CONTROL;
3. $S = U_3$; $J = $ NEXTELEMENT; GO TO CONTROL;
4. $S = Y_1$; $J = $ NEXTELEMENT; GO TO CONTROL;
5. $S = Y_2$; $J = $ NEXTELEMENT; GO TO CONTROL;
6. $S = Y_3$; $J = $ NEXTELEMENT; GO TO CONTROL;

Figure 15. Control routines for expanded grammar

Figure 16 shows a parse of the sentence $T_5^T_6T_3^T_4T_1^T_2$ described by the grammar in Figure 12. $S$ is set to $Y_0$ initially and $J = $ NEXTELEMENT.

<table>
<thead>
<tr>
<th>$S$</th>
<th>$J$</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_0$</td>
<td>$T_5$</td>
<td>6</td>
</tr>
<tr>
<td>$Y_3$</td>
<td>$T_6$</td>
<td>3</td>
</tr>
<tr>
<td>$U_3$</td>
<td>$T_3$</td>
<td>5</td>
</tr>
<tr>
<td>$Y_2$</td>
<td>$T_4$</td>
<td>2</td>
</tr>
<tr>
<td>$U_2$</td>
<td>$T_1$</td>
<td>4</td>
</tr>
<tr>
<td>$Y_1$</td>
<td>$T_2$</td>
<td>1</td>
</tr>
<tr>
<td>$U_1$</td>
<td></td>
<td>Finished</td>
</tr>
</tbody>
</table>

Figure 16. Parse of sentence $T_5^T_6T_3^T_4T_1^T_2$
Construction of an Expanded Operator Grammar

In developing an algorithm for obtaining an expanded grammar suitable for use with a transition matrix, the original grammar is restricted here to be an operator grammar. An operator grammar is a phrase structure grammar in which no production takes the form \( U \rightarrow XU_1U_2Y \) for some (possibly empty) strings \( X, Y \) and nonterminals \( U_1 \) and \( U_2 \).

Transition matrices can be used for grammars which are more general than operator grammars. However, the method being presented is based on an algorithm to be applied to an operator grammar to permit automatic construction of a transition matrix. No attempt is being made to do the same for more general grammars. Gries (9) states that the usual ALGOL (16) type languages can be represented by an operator grammar.

Steps are now given for taking the productions of an operator grammar (OG) and creating a new grammar called an expanded operator grammar (EOG). U's will be used to represent nonterminal symbols and T's will represent terminal symbols in the OG and EOG. The nonterminal symbols \( X_{hj} \) and \( Y_{hj} \) are generated as necessary to expand the OG into an EOG.

The OG will be expanded until the EOG will have productions with at the most two symbols in their right part. The productions with two symbols in their right part will
have either a Z or Y as the left symbol of the right part and
either a U, T or X as the right symbol of the right part.
Hence the transition matrix will have all of the Z's and Y's
as row headings and all of the U's, T's and X's as column
headings. Actually, a method is shown later that allows the
X's to be removed from the transition matrix.

The steps are described in a pseudo programming language
using nested IF-THEN-ELSE, GO-TO, assignment and plain English
statements. All the productions are searched for a condition
which calls for the application of a step. If a step is to
be applied to one or more productions, then all the produc­
tions whose right parts start with the identical symbols
(one or two, depending on the step) are modified in exactly
the same way. \( s_n \) is a nonempty and \( s_e \) a possible empty
string of U's and T's with no U's adjacent. \( h, i, j, k, m \) are
integer subscripts and have a single value at the time a
step is applied. The variable \( r \) which is created as a sub­
script for Z in Steps 3 and 4, represents a particular set of
integer values. The set of values for \( r \) depends on which \( U_i \)
is on the left side of the production in Steps 3 or 4. The
actual set of values for \( r \) is computed after all the steps
have been applied. The values for \( r \) are the subscripts of
the Y's which can appear adjacent and to the left of the \( U_i \)
in a sentential form. In terms of Definition 12 of Chapter I,
\[
    r = \{ j | Y_j \in \Sigma(U_i) \}.
\]
The algorithm for expanding an operator grammar consists of nineteen steps with loops. FLAG1, FLAG2, and FLAG3 are used to control the loops and are initially set to false. k is used as a subscript for Y and is initially set to zero. The steps for constructing the EOG are as follows:

**Step1:** IF there is a production $U_i \rightarrow T_m s_n$

THEN, $k = k+1$.

Replace each production $U_i \rightarrow T_m s_n$

(each production whose right part begins with $T_m$)

by the production $U_i \rightarrow Y_k s_n$.

Insert the production $Y_k \rightarrow T_m$.

GO TO Step1.

ELSE, GO TO Step2.

**Step2:** IF there is a production $U_i \rightarrow Y_j T_m s_n$

THEN, $k = k+1$.

Replace each production $U_i \rightarrow Y_j T_m s_n$

by the production $U_i \rightarrow Y_k s_n$.

Insert the production $Y_k \rightarrow Y_j T_m$.

GO TO Step2.

ELSE, GO TO Step3.

**Step3:** IF there is a production $U_i \rightarrow U_j T_m$

THEN, replace the production $U_i \rightarrow U_j T_m$

by the production $U_i \rightarrow Z_{rj} T_m$.

Insert the production $Z_{rj} \rightarrow U_j$. GO TO Step3.

ELSE, GO TO Step4.
Step4: IF there is a production $U_i \rightarrow U_j^T m^n$

THEN, replace each production $U_i \rightarrow U_j^T m^n$
by the production $U_i \rightarrow Z_{rj}^T m^n$.
Insert the production $Z_{rj} \rightarrow U_j$.

FLAG1 = TRUE.
GO TO Step4.

ELSE, IF FLAG1 = TRUE

THEN, GO TO Step5.

ELSE, GO TO Step7.

Step5: IF there is a production $U_i \rightarrow Z_{rj}^T m^n$

THEN, replace each production $U_i \rightarrow Z_{rj}^T m^n$
by the production $U_i \rightarrow Y_k^T m^n$.
Insert the production $Y_k \rightarrow Z_{rj}^T m^n$.
GO TO Step 5.

ELSE, GO TO Step 6.

/*Comment Step6 same as Step2 except for control*/

Step6: IF there is a production $U_i \rightarrow Y_j^T m^n$

THEN, $k = k+1$.

Replace each production $U_i \rightarrow Y_j^T m^n$
by the production $U_i \rightarrow Y_k^T m^n$.
Insert the production $Y_k \rightarrow Y_j^T m^n$.
GO TO Step6.

ELSE, GO TO Step7.
Step7: IF there is a production $U_i \rightarrow Y_h U_j T_m^s e$

    THEN, replace each production $U_i \rightarrow Y_h U_j T_m^s e$
    by the production $U_i \rightarrow Y_h Z_{hj} T_m^s e$.
    Insert the production $Z_{hj} \rightarrow U_j$.
    FLAG2 = TRUE, GO TO Step7.

ELSE, IF FLAG2 = TRUE

    THEN, GO TO Step8.

ELSE, GO TO Step11.

Step8: IF there is a production $U_i \rightarrow Y_h Z_{hj} T_m^s e$

    THEN, replace each production $U_i \rightarrow Y_h Z_{hj} T_m^s e$
    by the production $U_i \rightarrow Y_h X_{hjm}^s e$.
    Insert the production $X_{hjm} \rightarrow Z_{hj} T_m^s$.
    GO TO Step8.

ELSE, GO TO Step9.

Step9: IF there is a production $U_i \rightarrow Y_h X_{hjm}^s n$

    THEN, $k = k+1$.

    Replace each production $U_i \rightarrow Y_h X_{hjm}^s n$
    by the production $U_i \rightarrow Y_k X_{hjm}^s$.
    Insert the production $Y_k \rightarrow Y_h X_{hjm}^s$.
    GO TO Step9.

ELSE, GO TO Step10.
/*Comment Stepl0 is same as Step2 except for control*/

Step10: IF there is a production $U_i \to Y_h T_m a_n$

THEN, $k = k+1$.

Replace each production $U_i \to Y_h T_m a_n$
by the production $U_i \to Y_k s_n$.

Insert the production $Y_k \to Y_h T_m$.

GO TO Stepl0.

ELSE, FLAG2 = FALSE.

GO TO Step7.

Step11: IF there is a production $U_i \to Y_h U_j$

THEN, replace each production $U_i \to Y_h U_j$
by the production $U_i \to Y_h Z_{hj}$.

Insert the production $Z_{hj} \to U_j$.

FLAG3 = TRUE.

GO TO Step11.

ELSE, IF FLAG3 = TRUE

THEN, GO TO Step12.

ELSE, GO TO Step13.

Step12: IF there is a production $U_i \to Y_h Z_{hj}$

THEN, replace each production $U_i \to Y_h Z_{hj}$
by the production $U_i \to Y_h X_{hj0}$.

Insert the production $X_{hj0} \to Z_{hj}$.

GO TO Step12.

ELSE, GO TO Step13.
**Step13:** IF there is a production $Z_{ri} \rightarrow U_i$
THEN,

**Step13a:** IF there are productions $Z_{ri} \rightarrow U_i$ and $U_i \rightarrow U_j$
THEN, insert the production $Z_{rj} \rightarrow U_j$.

GO TO Step13a.

ELSE, GO TO Step13.

ELSE, GO TO Step14.

**Step14:** IF there is a production $Y_k \rightarrow Z_{ri}^T_m$
THEN,

**Step14a:** IF there are productions $Y_k \rightarrow Z_{ri}^T_m$ and $U_i \rightarrow U_j$
THEN, insert the production $Y_k \rightarrow Z_{rj}^T_m$.

GO TO Step14a.

ELSE, GO TO Step14.

ELSE, GO TO Step15.

**Step15:** IF there is a production $U_h \rightarrow Z_{ri}^T_m$
THEN,

**Step15a:** IF there are productions $U_h \rightarrow Z_{ri}^T_m$ and $U_i \rightarrow U_j$
THEN, insert the production $U_h \rightarrow Z_{rj}^T_m$.

GO TO Step15a.

ELSE, GO TO Step15.

ELSE, GO TO Step16.
Step16: IF there are productions $Z_{h_1} \rightarrow U_j$ and $U_i \rightarrow U_j$
THEN, insert the production $Z_{h_j} \rightarrow U_j$.
GO TO Step16.
ELSE, GO TO Step17.

Step17: IF there are productions $X_{hio} \rightarrow Z_{h_1}$ and $U_i \rightarrow U_j$
THEN, insert the production $X_{hio} \rightarrow Z_{h_j}$.
GO TO Step17.
ELSE, GO TO Step18.

Step18: IF there are productions $X_{him} \rightarrow Z_{h_1T_m}$ and $U_i \rightarrow U_j$
THEN, insert the production $X_{him} \rightarrow Z_{h_jT_m}$.
GO TO Step18.
ELSE, GO TO Step19.

Step19: Delete all the productions $U_i \rightarrow U_j$.
STOP.

After the application of the 19 steps to the OG the productions of the EGG will be in one of the eleven forms shown in Figure 17.
Figure 17. Eleven forms of the productions in an EOG

Once the values of \( r \) have been determined the two productions of the forms 1, 10 and 11 will be the same. In forms 2 and 3, \( X_{hnj0} \) is just a special case of \( X_{hnjm} \) where \( m \) takes on the value of zero.

The transition matrix with all the \( Y \)'s and \( Z \)'s as row headings and \( U \)'s, \( T \)'s and \( X \)'s as column headings has a cell corresponding to each production in Figure 17 with two symbols in the right part. In Figure 17, the EOG productions with two symbols in the right part are 2, 3, 6, 7, 8, 10 and 11.

To use the productions 1, 4, 5 and 9 in the transition matrix, additional information is needed as to which symbols may appear adjacent in a sentential form. In general, if the
single symbol on the right of a production is a U or T, then
the information sought is on Y's and Z's to go with it. If the
single symbol on the right of a production is a Z, then the in-
formation sought is on the U's, T's and X's to go with it.

In the productions of form 1 the h subscript of \( Z_{nj} \)
represents the \( Y_h \) that exists to the left of \( U_j \) for \( Z_{nj} \rightarrow U_j \).
This condition is added to the productions of form 1.

In the productions of form 4 it is necessary to know all
the Y's and Z's that can exist adjacent and to the left of \( U_i \)
in all the sentential forms of the grammar. Note that for
the production \( U_i \rightarrow T_m \) the Y's and Z's are to be determined
that are adjacent and to the left of \( U_i \) (not \( T_m \)) in the sent-
tential forms. Let \( s_j \) be the sentential form equal to \( eBT_mB \)
where \( e \) and \( b \) are strings (possibly empty) of symbols and \( B \)
is a single Y or Z. If the production \( U_i \rightarrow T_m \) is applied to
\( s_j \) to get \( s_{j+1} \), then \( s_{j+1} \) is equal to \( eBU_ib \). The symbol to
the left of \( T_m \) in \( s_j \) is the same symbol to the left of \( U_i \) in
\( s_{j+1} \). Therefore, before the \( U_i \rightarrow T_m \) can be applied to \( s_j \) the
B must be a symbol that can exist adjacent and to the left of
\( U_i \) in some sentential form.

In the productions of form 5 it is necessary to know
all the Y's and Z's that can exist adjacent and to the left
of \( Y_i \) in all the sentential forms of the grammar. It will
be shown later that there are no sentential forms of the
grammar with Z's on the left and adjacent to a U or a Y.
In the productions of form 9 it is necessary to know all the U's and T's that can exist to the right of $X_{hj0}$ in all the sentential forms of the grammar. There are no sentential forms with U's adjacent to the right of a Z, so it is only necessary to determine the T's for the productions of form 9.

Let $Y$ be a certain set of $Y_i$'s and $T$ be a certain set of $T_i$'s. $s(A)$ is defined as the set of symbols which are adjacent and to the left of $A$ in some sentential form. Mathematically $s(A)$ is defined as follows:

$$s(A) = \left\{ B | U_i \not\in sBAb, (e,b) \in \mathcal{V}^*, (B,A) \in \mathcal{V} \right\}$$

Using this definition of $s$, the eleven forms of the EOG production are shown in Figure 18 with the additional conditions stated.

The productions are now in a form from which a transition matrix can be built, with U's, T's and X's as columns and Y's and Z's as rows. However, the need for columns for the X's can be avoided by applying two productions at once. This can be done because, if a production of form 8 or 9 is applied, the next production to be applied is a unique production of form 2 or 3 with the same $X_{hjm}$. Since this reduces $Y_nZ_{hj}T_m$ or $Y_nZ_{hj}$ to a $U_i$ or $Y_i$, there is no need for X's in the transition matrix.
1. \( Z_{hj} \rightarrow U_j \) for \( Y = Y_h \)
2. \( U_i \rightarrow Y_h X_{hjm} \)
3. \( Y_i \rightarrow Y_h X_{hjm} \)
4. \( U_i \rightarrow T_m \) for \( Y \) where \( Y = \{ Y_j \mid Y_j \in \mathcal{L}(U_i) \} \)
5. \( Y_i \rightarrow T_m \) for \( Y \) where \( Y = \{ Y_j \mid Y_j \in \mathcal{L}(Y_i) \} \)
6. \( Y_i \rightarrow Y_j T_m \)
7. \( U_i \rightarrow Y_j T_m \)
8. \( X_{hjm} \rightarrow Z_{hj} T_m \)
9. \( X_{hj0} \rightarrow Z_{hj} \) for \( T \) where \( T = \{ T_i \mid X_{hj0} \in \mathcal{L}(T_i) \} \)
10. \( Y_i \rightarrow Z_{hj} T_m \)
11. \( U_i \rightarrow Z_{hj} T_m \)

**Figure 18. Eleven forms of the EOG production with added conditions**

**Finding the \( \mathcal{L} \) Function**

Two simple computer programs are presented in this section to find the \( \mathcal{L} \) function, using the LEFT and RIGHT functions defined in Chapter I. The LEFT(\( U_i \)) is the set of symbols that can occur on the left end of a derivation of \( U_i \) and \( U_i \) itself. The LEFT(\( T_i \)) is \( T_i \). The RIGHT(\( U_i \)) is the set of symbols that can occur on the right end of a derivation of \( U_i \) and \( U_i \) itself. The RIGHT(\( T_i \)) is \( T_i \). An example using the grammar in Figure 19 is used to illustrate the procedure.
The $L$ and $R$ matrices are constructed in the following manner. For the $L$ matrix use the symbol on the left part of the production and the left most symbol of the right side of the production to place a 1 in the $L$ matrix. For the $R$ matrix use the symbol on the left part of the production and the right most symbol on the right side of the production to place a 1 in the $R$ matrix. Figure 20 shows the $L$ and $R$ matrix for the grammar of Figure 19.
L and R are assumed to be Boolean matrices for finding the LEFT and RIGHT functions. Two bit string vectors, LV
and RV, of length six with elements of 6 bits each, are overlayed on the matrices L and R. The size of the vector is
made to match the dimensions of the matrix. This allows rows to be logically ORed together. The program in PL/1 of
Figure 21 is based on Warshall's (21) algorithm and is used to obtain the LEFT and RIGHT functions from the L and R
matrices respectively. The LEFT and RIGHT functions are shown in Figure 22. To find the function a matrix two
columns wide is built to hold all the adjacent pairs of symbols on the right of all the productions in the grammar.
The symbols are coded with unique integer values and the results are shown in Figure 23 in the matrix MAT2. MAT2 is
used along with the LEFT and RIGHT functions to create a matrix LL, from which the function may be obtained. The
LL matrix is initially all zeros and is Boolean with an overlayed bit-string vector LLV of length six with elements
of six bits each. LLV again is used for logically ORing rows together. The Boolean matrices L and R now contain the
LEFT and RIGHT functions. L and R are used in the PL/1 program of Figure 24 to compute the LL matrix. The re-
sulting LL matrix is shown in Figure 25.

To find the function of any symbol in the grammar, look at the column of the LL matrix under that symbol. All
DCL LV(6) BIT(6) STATIC;
DCL L(6,6) BIT(1) DEF LV;
DCL RV(6) BIT(6) STATIC;
DCL R(6,6) BIT(1) DEF RV;
N = 6;

LAB1: DO I = 1 BY 1 TO N;
LAB2: DO J = 1 BY 1 TO N;
IF L(J,I) THEN LV(J) = LV(J) | LV(I);
IF R(J,I) THEN RV(J) = RV(J) | RV(I);
END LAB1;
END LAB2;

LAB3: DO I = 1 BY 1 TO N;
L(I,I) = '1'B;
R(I,I) = '1'B;
END LAB3;

Figure 21. PL/1 program for computing LEFT and RIGHT function

<table>
<thead>
<tr>
<th></th>
<th>U1</th>
<th>U2</th>
<th>U3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th></th>
<th>U1</th>
<th>U2</th>
<th>U3</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>U2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>U3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 22. LEFT and RIGHT functions for grammar example
<table>
<thead>
<tr>
<th>PAIRS</th>
<th>PRODUCTION</th>
<th>MAT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>U₂, U₃</td>
<td>1</td>
<td>2 3</td>
</tr>
<tr>
<td>U₃, T₁</td>
<td>1</td>
<td>3 4</td>
</tr>
<tr>
<td>U₁, T₂</td>
<td>2</td>
<td>1 5</td>
</tr>
<tr>
<td>T₁, U₁</td>
<td>4</td>
<td>4 1</td>
</tr>
<tr>
<td>T₁, T₁</td>
<td>5</td>
<td>4 4</td>
</tr>
<tr>
<td>U₁, U₁</td>
<td>6</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Figure 23. Two column matrix for adjacent symbols

DCL LLV(6) BIT(6) STATIC INIT ((6)(6)'0'B);
DCL LL(6,6) BIT(1) DEF LLV;
N = 6;
M = 6;
LAB4: DO I = 1 BY 1 TO M;
L1 = MAT2(I,1);
L2 = MAT2(I,2);
LAB5: DO J = 1 BY 1 TO N;
IF R(L1,J) THEN LLV(J) = LLV(J) | LV(L2);
END LAB5;
END LAB4;

Figure 24. PL/1 program for computing LL matrix
the I's in that column represent the set of symbols that are the $f$ function. For example, $f(T_2) = (U_1, T_1, T_2)$. The LL matrix is most useful in constructing recognizers because it tells which symbols may appear adjacent to each other. This same information would be most useful in constructing the recognizer of Chapter II.

```
<table>
<thead>
<tr>
<th></th>
<th>U_1</th>
<th>U_2</th>
<th>U_3</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>U_1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>U_2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>U_3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T_1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T_2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T_3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Figure 25. LL matrix for grammar example

Example of Constructing an EOG

An example of construction of an EOG is given in this section. The nineteen steps of the previous section are applied to the grammar in Figure 26.
The grammar is shown before and after each step as it is being transformed from an OG to an EOG. The productions that are to be changed in a step are underlined. After the application of all the steps, the conditions specified in Figure 18 are obtained. Also the set of values the $r$'s can take on in the productions of the form $Z_{rj} \rightarrow U_j$ are determined.

**Step 1**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow T_1 U_2 T_3 U_2 T_4$</td>
<td>$U_1 \rightarrow Y_1 U_2 T_3 U_2 T_4$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_2 T_6 U_3$</td>
<td>$U_2 \rightarrow U_2 T_6 U_3$</td>
</tr>
<tr>
<td>$U_3 \rightarrow T_7 U_2 T_8$</td>
<td>$U_3 \rightarrow Y_2 U_2 T_8$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>$U_2 \rightarrow U_3$</td>
</tr>
<tr>
<td>$U_3 \rightarrow T_1$</td>
<td>$U_3 \rightarrow T_1$</td>
</tr>
</tbody>
</table>

The production $U_3 \rightarrow T_1$ is never altered, therefore, it is not carried along but is included at the final step.
Step 2

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_1T_2U_2T_3U_2T_4$</td>
<td>$U_1 \rightarrow Y_3U_2T_3U_2T_4$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow T_1$</td>
<td>$Y_1 \rightarrow T_1$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_2^6U_2$</td>
<td>$U_2 \rightarrow U_2^6U_3$</td>
</tr>
<tr>
<td>$U_3 \rightarrow Y_2U_2T_8$</td>
<td>$U_3 \rightarrow Y_2U_2T_8$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>$U_2 \rightarrow U_3$</td>
</tr>
</tbody>
</table>

Step 3 does not apply to any of the productions.

Step 4

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_3U_2T_3U_2T_4$</td>
<td>$U_1 \rightarrow Y_3U_2T_3U_2T_4$</td>
</tr>
<tr>
<td>$Y_3 \rightarrow Y_1T_2$</td>
<td>$Y_3 \rightarrow Y_1T_2$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow T_1$</td>
<td>$Y_1 \rightarrow T_1$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_2^6U_3$</td>
<td>$U_2 \rightarrow Z_{r_2}^6U_3$</td>
</tr>
<tr>
<td>$U_3 \rightarrow Y_2U_2T_8$</td>
<td>$U_3 \rightarrow Y_2U_2T_8$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>$U_2 \rightarrow U_3$</td>
</tr>
</tbody>
</table>

where $r = \{j | Y_j \in \xi(U_2)\}$
### Step 5

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_3 U_2 T_3 U_2 T_4$</td>
<td>$U_1 \rightarrow Y_3 U_2 T_3 U_2 T_4$</td>
</tr>
<tr>
<td>$Y_3 \rightarrow Y_1 T_2$</td>
<td>$Y_3 \rightarrow Y_1 T_2$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow T_1$</td>
<td>$Y_1 \rightarrow T_1$</td>
</tr>
<tr>
<td>$U_2 \rightarrow Z_{r2} T_6 U_3$</td>
<td>$U_2 \rightarrow Y_4 U_3$</td>
</tr>
<tr>
<td>$Z_{r2} \rightarrow U_2$</td>
<td>$Z_{r2} \rightarrow U_2$</td>
</tr>
<tr>
<td>$U_3 \rightarrow Y_2 U_2 T_8$</td>
<td>$U_3 \rightarrow Y_2 U_2 T_8$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>$U_2 \rightarrow U_3$</td>
</tr>
</tbody>
</table>

**Step 6** does not apply to any of the productions

### Step 7

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_3 U_2 T_3 U_2 T_4$</td>
<td>$U_1 \rightarrow Y_3 Z_{32} T_3 U_2 T_4$</td>
</tr>
<tr>
<td>$Y_3 \rightarrow Y_1 T_2$</td>
<td>$Y_3 \rightarrow Y_1 T_2$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow T_1$</td>
<td>$Y_1 \rightarrow T_1$</td>
</tr>
<tr>
<td>$U_2 \rightarrow Y_4 U_3$</td>
<td>$U_2 \rightarrow Y_4 U_3$</td>
</tr>
<tr>
<td>$Y_4 \rightarrow Z_{r2} T_6$</td>
<td>$Y_4 \rightarrow Z_{r2} T_6$</td>
</tr>
<tr>
<td>$Z_{r2} \rightarrow U_2$</td>
<td>$Z_{r2} \rightarrow U_2$</td>
</tr>
<tr>
<td>$U_3 \rightarrow Y_2 U_2 T_8$</td>
<td>$U_3 \rightarrow Y_2 Z_{22} T_8$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>$U_2 \rightarrow U_3$</td>
</tr>
</tbody>
</table>
### Step 8

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_3 Z_2 T_3 U_2 T_4$</td>
<td>$U_1 \rightarrow Y_3 X_3 Z_2 T_3 U_2 T_4$</td>
</tr>
<tr>
<td>$Z_2 \rightarrow U_2$</td>
<td>$Z_3 \rightarrow U_2$</td>
</tr>
<tr>
<td>$Y_3 \rightarrow Y_1 T_2$</td>
<td>$Y_3 \rightarrow Y_1 T_2$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow T_1$</td>
<td>$Y_1 \rightarrow T_1$</td>
</tr>
<tr>
<td>$U_2 \rightarrow Y_4 U_3$</td>
<td>$U_2 \rightarrow Y_4 U_3$</td>
</tr>
<tr>
<td>$Y_4 \rightarrow Z r_2 T_6$</td>
<td>$Y_4 \rightarrow Z r_2 T_6$</td>
</tr>
<tr>
<td>$Z r_2 \rightarrow U_2$</td>
<td>$Z r_2 \rightarrow U_2$</td>
</tr>
<tr>
<td>$U_3 \rightarrow Y_2 Z_{22} T_8$</td>
<td>$U_3 \rightarrow Y_2 X_{228}$</td>
</tr>
<tr>
<td>$Z_{22} \rightarrow U_2$</td>
<td>$Z_{22} \rightarrow U_2$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>$U_2 \rightarrow U_3$</td>
</tr>
</tbody>
</table>

The next set of Steps (9, 7, and 8) only affects the production $U_1 \rightarrow Y_3 X_3 Z_2 T_3 U_2 T_4$.

### Step 9

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_3 X_3 Z_2 T_3 U_2 T_4$</td>
<td>$U_1 \rightarrow Y_5 U_2 T_4$</td>
</tr>
<tr>
<td>$Y_5 \rightarrow Y_3 X_3 Z_2$</td>
<td>$Y_5 \rightarrow Y_3 X_3 Z_2$</td>
</tr>
</tbody>
</table>

Step 10 does not apply to any of the productions.
Step 7

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_5U_2T_4$</td>
<td>$U_1 \rightarrow Y_5Z_52T_4$</td>
</tr>
<tr>
<td>$Y_5 \rightarrow Y_3X_323$</td>
<td>$Y_5 \rightarrow Y_3X_323$</td>
</tr>
</tbody>
</table>

$Z_{52} \rightarrow U_2$

Step 8

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_5Z_52T_4$</td>
<td>$U_1 \rightarrow Y_5X_524$</td>
</tr>
<tr>
<td>$Z_{52} \rightarrow U_2$</td>
<td>$Z_{52} \rightarrow U_2$</td>
</tr>
<tr>
<td>$Y_5 \rightarrow Y_3X_323$</td>
<td>$Y_5 \rightarrow Y_3X_323$</td>
</tr>
</tbody>
</table>

$X_{524} \rightarrow Z_{52T4}$

Steps 9 and 10 do not apply to any of the productions.
<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U_1 \rightarrow Y_5 X_{524} )</td>
<td>( \bar{U}<em>1 \rightarrow Y_5 X</em>{524} )</td>
</tr>
<tr>
<td>( X_{524} \rightarrow Z_{52T_4} )</td>
<td>( X_{524} \rightarrow Z_{52T_4} )</td>
</tr>
<tr>
<td>( Z_{52} \rightarrow U_2 )</td>
<td>( Z_{52} \rightarrow U_2 )</td>
</tr>
<tr>
<td>( Y_5 \rightarrow Y_3 X_{323} )</td>
<td>( Y_5 \rightarrow Y_3 X_{323} )</td>
</tr>
<tr>
<td>( X_{323} \rightarrow Z_{32T_3} )</td>
<td>( X_{323} \rightarrow Z_{32T_3} )</td>
</tr>
<tr>
<td>( Z_{32} \rightarrow U_2 )</td>
<td>( Z_{32} \rightarrow U_2 )</td>
</tr>
<tr>
<td>( Y_3 \rightarrow Y_1 T_2 )</td>
<td>( Y_3 \rightarrow Y_1 T_2 )</td>
</tr>
<tr>
<td>( Y_1 \rightarrow T_1 )</td>
<td>( Y_1 \rightarrow T_1 )</td>
</tr>
<tr>
<td>( U_2 \rightarrow Y_4 U_3 )</td>
<td>( U_2 \rightarrow Y_4 Z_{43} )</td>
</tr>
<tr>
<td>( Y_4 \rightarrow Z_{r2T_6} )</td>
<td>( Y_4 \rightarrow Z_{r2T_6} )</td>
</tr>
<tr>
<td>( Z_{r2} \rightarrow U_2 )</td>
<td>( Z_{r2} \rightarrow U_2 )</td>
</tr>
<tr>
<td>( U_3 \rightarrow Y_2 X_{228} )</td>
<td>( U_3 \rightarrow Y_2 X_{228} )</td>
</tr>
<tr>
<td>( X_{228} \rightarrow Z_{22T_8} )</td>
<td>( X_{228} \rightarrow Z_{22T_8} )</td>
</tr>
<tr>
<td>( Z_{22} \rightarrow U_2 )</td>
<td>( Z_{22} \rightarrow U_2 )</td>
</tr>
<tr>
<td>( Y_2 \rightarrow T_7 )</td>
<td>( Y_2 \rightarrow T_7 )</td>
</tr>
<tr>
<td>( U_2 \rightarrow U_3 )</td>
<td>( U_2 \rightarrow U_3 )</td>
</tr>
</tbody>
</table>
Steps 13, 14, 15, 16, 17 and 18 can be applied simultaneously because they involve mutually exclusive productions.
<table>
<thead>
<tr>
<th>Before</th>
<th>Step No.</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \rightarrow Y_5 X_{524}$</td>
<td>1</td>
<td>$U_1 \rightarrow Y_5 X_{524}$</td>
</tr>
<tr>
<td>$X_{524} \rightarrow Z_{52}^T$</td>
<td>2</td>
<td>$X_{524} \rightarrow Z_{52}^T$</td>
</tr>
<tr>
<td>$Z_{52} \rightarrow U_2$</td>
<td>18</td>
<td>$Z_{52} \rightarrow U_2$</td>
</tr>
<tr>
<td>$Y_5 \rightarrow Y_3 X_3$</td>
<td>16</td>
<td>$Y_5 \rightarrow Y_3 X_3$</td>
</tr>
<tr>
<td>$X_3 \rightarrow Z_{32}^T$</td>
<td>6</td>
<td>$X_3 \rightarrow Z_{32}^T$</td>
</tr>
<tr>
<td>$Z_{32} \rightarrow U_2$</td>
<td>18</td>
<td>$Z_{32} \rightarrow U_2$</td>
</tr>
<tr>
<td>$Y_3 \rightarrow Y_1 X_1$</td>
<td>16</td>
<td>$Y_3 \rightarrow Y_1 X_1$</td>
</tr>
<tr>
<td>$Y_1 \rightarrow T_1$</td>
<td>11</td>
<td>$Y_1 \rightarrow T_1$</td>
</tr>
<tr>
<td>$U_2 \rightarrow Y_4 X_4$</td>
<td>13</td>
<td>$U_2 \rightarrow Y_4 X_4$</td>
</tr>
<tr>
<td>$X_4 \rightarrow Z_4$</td>
<td>14</td>
<td>$X_4 \rightarrow Z_4$</td>
</tr>
<tr>
<td>$Z_4 \rightarrow U_3$</td>
<td>15</td>
<td>$Z_4 \rightarrow U_3$</td>
</tr>
<tr>
<td>$Y_4 \rightarrow Z_x X_{228}$</td>
<td>16</td>
<td>$Y_4 \rightarrow Z_x X_{228}$</td>
</tr>
<tr>
<td>$Z_22 \rightarrow U_2$</td>
<td>18</td>
<td>$Z_22 \rightarrow U_2$</td>
</tr>
<tr>
<td>$U_3 \rightarrow Y_2 X_{228}$</td>
<td>19</td>
<td>$U_3 \rightarrow Y_2 X_{228}$</td>
</tr>
<tr>
<td>$X_{228} \rightarrow Z_{22}^T$</td>
<td>20</td>
<td>$X_{228} \rightarrow Z_{22}^T$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>21</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_2 \rightarrow U_3$</td>
<td>22</td>
<td>$U_3 \rightarrow T_3$</td>
</tr>
<tr>
<td>$Z_22 \rightarrow U_2$</td>
<td>23</td>
<td>$Z_22 \rightarrow U_2$</td>
</tr>
<tr>
<td>$Y_2 \rightarrow T_7$</td>
<td>24</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>deleted</td>
<td>25</td>
<td>$Y_2 \rightarrow T_7$</td>
</tr>
<tr>
<td>$U_3 \rightarrow T_1$</td>
<td>26</td>
<td>$U_3 \rightarrow T_1$</td>
</tr>
</tbody>
</table>
To find the set of values for \( r \) to be used in productions 16, 17, 18 and 19, it is necessary to trace back to Step 4 to find out which \( U_4 \) is used for defining \( r \). \( r \) is found to be based on \( U_2 \) because it was the left part of the production when Step 4 was applied. \( (r = \{ j \mid Y_j \in \mathcal{L}(U_2) \}) \). In general, rules 3 and 4 may be applied to more than one production creating more than one \( r \). An easy way to keep track of the \( r \)'s and which \( U \)'s they are defined by, is to replace the \( r \) by the negative subscript of \( U \). Productions 16, 17, 18 and 19 would appear as follows:

\[
\begin{align*}
16 & \quad Y_4 \to Z_{-22}^T \, 6 \\
17 & \quad Y_4 \to Z_{-23}^T \, 6 \\
18 & \quad Y_{-22} \to U_2 \\
19 & \quad Z_{-23} \to U_3
\end{align*}
\]

The use of the negative subscript at Steps 3 and 4 avoids the problem of tracing back and the problem of multiple \( r \)'s. Since the procedure for finding the \( \mathcal{L} \) function was shown in a previous section, only the results are shown here for \( r \) and the productions of forms 4, 5 and 9 in Figure 18.

\( r \) is found to have the values 2, 3 and 5. The replacing of \( r \) in productions 16, 17, 18 and 19 results in the creation of twelve productions. As in the previous transition matrix of Figure 14, the state \( Y_0 \) is added to the transition matrix for initializing the state stack. If the \( \mathcal{L}(Y_1) \) is an empty set, then it is necessary to let \( Y = Y_0 \) because the state
stack is always initialized with $Y_0$. Figure 27 shows the results of applying the conditions shown in Figure 18 for the productions in forms 4, 5 and 9. Figure 28 shows the final form of the grammar ready for creating a transition matrix.

<table>
<thead>
<tr>
<th>Form</th>
<th>Production</th>
<th>Set</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>12</td>
<td>$Y = {Y_j</td>
<td>Y_j \in \Sigma(Y_1)}$</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>$T = {T_1</td>
<td>X_{430} \in \Sigma(T_1)}$</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>$Y = {Y_j</td>
<td>Y_j \in \Sigma(Y_2)}$</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>$Y = {Y_j</td>
<td>Y_j \in \Sigma(U_3)}$</td>
</tr>
</tbody>
</table>

Figure 27. Results of applying $\Sigma$ functions

Construction of the Recognizer

The recognizer consists of a set of routines for making reductions and a transition matrix. The transition matrix contains a set of control routine numbers and is used to select the correct routine to execute the reduction. A row of the matrix is assigned to each $Y$ and $Z$ and a column of the matrix is assigned to each $U$ and $T$.

The process uses a stack called STACK with a pointer $S$ to store $Y$'s and $Z$'s, and a table called TABLE with a pointer $T$ to store the $U$'s and $T$'s. The top element of STACK and the element of TABLE indicated by $T$ are used as row and column subscripts of the transition matrix to determine the control routine to be executed. The control routine makes a reduction
Figure 28. Grammar example for transition matrix
by performing operations on the STACK and TABLE, such as adding or deleting symbols. TABLE is initialized with the incoming string of source elements with the first element in the first location of TABLE.

In the development of the control routine consider the sentential form as being represented by the contents of STACK concatenated with the contents of TABLE from location T to the end of the source element string. The production to be applied to the sentential form is determined by the symbol at the top of STACK and the symbol in location T of TABLE. In the sentential form they are actually adjacent symbols. The STACK symbol is a $Y_h$ or $Z_{hj}$ and the TABLE symbol is a $U_i$ or $T_m$. If the production $Y_i \rightarrow T_m$ is to be applied to a sentential form $s_i$ then $S = S+1$; STACK($S$) = $Y_i$; $T = T+1$; would create the new sentential form $s_{i+1}$ resulting from the application of the production $Y_i \rightarrow T_m$. $Y_i \rightarrow Z_{hj}T_m$ can be applied by STACK($S$) = $Y_i$; $T = T+1$; $Z_{hj}$ is already on top of STACK so it is replaced in the sentential form with $Y_i$. Letting $T = T+1$ takes the $T_m$ out of the sentential form. $U_i \rightarrow Z_{hj}T_m$ can be applied by $S = S-1$; TABLE($T$) = $U_i$; $S = S-1$ takes $Z_{hj}$ out of the sentential form and TABLE($T$) = $U_i$ replaces $T_m$ with $U_i$ in the sentential form. The two productions $X_{njm} \rightarrow Z_{njm}T_m$ and $Y_i \rightarrow Y_hX_{njm}$ can be applied by $S = S-1$; STACK($S$) = $Y_i$; $T = T+1$; which is equivalent to replacing $Y_hZ_{hj}T_m$ in the sentential form by $Y_i$. 
Figure 29 shows a set of control routines for the productions of Figure 18. The control routines RT6, RT7, RT8 and RT9 apply a unique sequence of two productions at one time. This saves having X in the transition matrix and reduces the number of steps in a parse. The control routine RT2 applies a unique sequence of two productions to save steps in a parse.

If the productions of Figure 28 are to be used in a recognizer, each production would require its own control routine similar to one of the routines shown in Figure 29. In the routines of Figure 29, each routine stores only one symbol (the left part of the appropriate production) in TABLE or STACK. The control routines are the same for productions of the same form except for the symbol that is stored. To save constructing a large number of control routines, this symbol is stored in the transition matrix along with the control routine number.

Figures 30 and 31 show the transition matrix and control routines for the set of productions in Figure 28. INFO is a matrix used to reference the symbols in the transition matrix. In the implementation INFO contains an integer corresponding to each symbol in the transition matrix of Figure 30. The following is a description of how the integer values are assigned in INFO. If the symbol is a U or T the number of the column in the transition matrix corresponding to that
Figure 29. Control routines for productions of Figure 18
Figure 30. Transition matrix for productions of Figure 28
CONTROL:  
I = STACK(S);  
J = TABLE(T);  
GO TO LABM(M(I,J));  

RT1:  
S = S+1; STACK(S) = INFO(I,J);  
T = T+1; GO TO CONTROL;  

RT2:  
STACK(S) = INFO(I,J); T = T+1;  
GO TO CONTROL;  

RT3:  
S = S-1; TABLE(T) = INFO(I,J);  
GO TO CONTROL;  

RT4:  
S = S-2; TABLE(T) = INFO(I,J);  
GO TO CONTROL;  

RT5:  
S = S-1; STACK(S) = INFO(I,J);  
T = T+1; GO TO CONTROL;  

RT6:  
S = S-2; T = T-1;  
TABLE(T) = INFO(I,J); GO TO CONTROL;  

RT7:  
S = S-1; STACK(S) = INFO(I,J);  
GO TO CONTROL;  

RT8:  SUCCESSFUL PARSE;  

Figure 31. Control routines for productions of Figure 28

<table>
<thead>
<tr>
<th>Production(s)</th>
<th>Routine</th>
<th>Production(s)</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>8,3</td>
<td>5</td>
</tr>
<tr>
<td>4,1</td>
<td>1</td>
<td>9,2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>9,3</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8,2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 32. Production forms and control routines
symbol is used. For example, the integer 5 is used for $T_2$ because it heads the 5th column. If the symbol is a $Y$ or $Z$, the number of the row in the transition matrix corresponding to that symbol is used.

The control routine numbers are in a matrix $M$ which is used as a subscript for a PL/1 label vector. It has been stated that EOG productions of the same form can use the same control routine if the symbol to be stored is placed in INFO. However, on examining Figure 29 some of the control routines for different EOG productions are the same except for the symbol they store. For example, RT1, RT2 and RT3 are the same except for the symbol they store. The routines that are the same in Figure 29, except for the symbol to be stored, are combined and the results are shown in Figure 31. The list in Figure 32 shows the relationship between the eleven forms of the productions of Figure 18 and the routines of Figure 31.

Figure 33 shows the parse of the sentence $T_1T_2T_3T_1T_6T_1T_8T_3T_1T_4$ in the EOG. $Y_0$ is placed in STACK and $S$ and $T$ are set equal to one initially. If the contents of STACK and TABLE shown in Figure 33 are concatenated, the resulting strings would represent the sentential forms produced in reducing the sentence to the starting symbol $U_1$. At each point the left most prime phrase and the symbol to which it is to be reduced is determined by the top of STACK and the entry in TABLE.
<table>
<thead>
<tr>
<th>Step</th>
<th>Routine</th>
<th>S</th>
<th>T</th>
<th>Stack Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( Y_0 )</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>( Y_0Y_1 )</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>( Y_0Y_3 )</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>( Y_0Y_3Y_2 )</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>( Y_0Y_3Y_2Z_{23} )</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>( Y_0Y_3Y_2Y_4 )</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>( Y_0Y_3Y_2Y_4Z_{43} )</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>( Y_0Y_3Y_2 )</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>( Y_0Y_3Y_2Z_{22} )</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>( Y_0Y_3 )</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>( Y_0Y_3Z_{33} )</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>( Y_0Y_5 )</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>( Y_0Y_5Z_{53} )</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>( Y_0 )</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>BRANCH TO SUCCESSFUL PARSE</td>
</tr>
</tbody>
</table>

Figure 33. Parse of sentence \( T_1T_2T_7T_1T_8T_3T_1T_4 \)
To perform semantic operations using the recognizer of Figures 30 and 31, it is necessary to add to the transition matrix semantic routine numbers. The control section would first branch to the semantic routine indicated in the transition matrix and then branch back to one of the control routines in Figure 31.

Because of the empty spaces in the transition matrix of Figure 30 the nonempty spaces are stored in a list structure. Figure 31 shows the list structure storage for the transition matrix in Figure 30. The elements of the matrix are stored in the list by row. The elements within a row are not necessarily stored in the order in which they appeared in the matrix. Integer values are assigned to the U's and T's as shown below:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>U T T T T T T T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
</tbody>
</table>

The following is a description of how the location of an element within a row is obtained. The value associated with U or T of that column is divided by the total number of nonempty entries in that row. The remainder is used as a displacement from the beginning of the row in the list. The number of locations reserved for a row in the list equals the number of nonempty entries in that row. If two or more elements in a row have the same remainder the additional elements are placed in the leftover locations and chained to the
<table>
<thead>
<tr>
<th>LINE NO.</th>
<th>TV</th>
<th>N</th>
<th>CHN</th>
<th>BRNR</th>
<th>INFOS</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$y_0$</td>
<td>$T_1$</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$u_1$</td>
<td>$T_1$</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>$y_1$</td>
<td>$T_2$</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$y_2$</td>
<td>$T_1$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$T_7$</td>
<td>$2$</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>$u_2$</td>
<td>$2$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$z_{23}$</td>
</tr>
<tr>
<td>7</td>
<td>$u_3$</td>
<td>$3$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$z_{33}$</td>
</tr>
<tr>
<td>8</td>
<td>$y_3$</td>
<td>$T_1$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>$T_7$</td>
<td>$9$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>$u_2$</td>
<td>$2$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$z_{32}$</td>
</tr>
<tr>
<td>11</td>
<td>$u_3$</td>
<td>$3$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$z_{33}$</td>
</tr>
<tr>
<td>12</td>
<td>$y_4$</td>
<td>$u_3$</td>
<td>3</td>
<td>3</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>$T_1$</td>
<td>$4$</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>$z_{43}$</td>
</tr>
<tr>
<td>14</td>
<td>$T_7$</td>
<td>$9$</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>$y_2$</td>
</tr>
<tr>
<td>15</td>
<td>$y_5$</td>
<td>$T_1$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>$T_7$</td>
<td>$9$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$y_2$</td>
</tr>
<tr>
<td>17</td>
<td>$u_2$</td>
<td>$2$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$z_{52}$</td>
</tr>
<tr>
<td>18</td>
<td>$u_3$</td>
<td>$3$</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>$z_{53}$</td>
</tr>
</tbody>
</table>

Figure 34. List structure...
<table>
<thead>
<tr>
<th>LINE NO.</th>
<th>TV</th>
<th>N</th>
<th>CHN</th>
<th>BRNR</th>
<th>INFOS</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>$Z_{22}$</td>
<td>$T_6$</td>
<td>8</td>
<td>2</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>$T_8$</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>$U_3$</td>
</tr>
<tr>
<td>21</td>
<td>$Z_{23}$</td>
<td>$T_6$</td>
<td>8</td>
<td>2</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>$T_8$</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>$U_3$</td>
</tr>
<tr>
<td>23</td>
<td>$Z_{32}$</td>
<td>$T_3$</td>
<td>6</td>
<td>2</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>$T_6$</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>$Y_4$</td>
</tr>
<tr>
<td>25</td>
<td>$Z_{33}$</td>
<td>$T_3$</td>
<td>6</td>
<td>2</td>
<td>26</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>$T_6$</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>$Y_4$</td>
</tr>
<tr>
<td>27</td>
<td>$Z_{43}$</td>
<td>$T_6$</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>28</td>
<td>$T_8$</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>$U_2$</td>
</tr>
<tr>
<td>29</td>
<td>$T_3$</td>
<td>6</td>
<td>4</td>
<td>29</td>
<td>6</td>
<td>$U_2$</td>
</tr>
<tr>
<td>30</td>
<td>$T_4$</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>$U_2$</td>
</tr>
<tr>
<td>31</td>
<td>$Z_{52}$</td>
<td>$T_6$</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>$T_4$</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>$U_1$</td>
</tr>
<tr>
<td>33</td>
<td>$Z_{53}$</td>
<td>$T_6$</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>$T_4$</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>$U_1$</td>
</tr>
</tbody>
</table>

Figure 34 (continued)
original location, using the vector CHN. In Figure 34, the vector TV contains the integer value corresponding to a particular U or T. N contains the number of nonempty elements in the row, BRNR contains the number of the control routine from the transition matrix and INFOS contains the symbols from the transition matrix to be used in the control routines. The integer numbers in INFO are put in after all the other entries in the list are completed. The following is a description of how the integer values for INFO are determined.

If the symbol in INFOS is a Y or Z then put the line number corresponding to that symbol in INFO. If the symbol is a U, put the integer value associated with U in INFO.

Figure 35 shows the control section using the vectors described in Figure 34. The list structure shown in Figure 34 also contains (but not shown) a vector of numbers (BRNS) referring to semantic routines that are to be executed. BRANCHS and BRNCHR are PL/1 label vectors containing the names of the semantic and control routines. Each semantic routine has at the end a GO TO BRNCHR(BRNR(JZ)) statement for branching to the correct control routine. The control routines in Figure 35 are the same as those in Figure 31, only rewritten to share some common instructions. In Figure 35, TABLE is not initialized with the string of incoming source elements, but the elements are placed in the TABLE one at a time as they are needed. The variable FLAG is used to control this process.
CONTROL:  JZ = STACK(S) + MOD(TABLE(T),N(STACK(S)));
LOOPL:  IF TV(JZ) = TABLE(T) THEN GO TO BRANCHS(BRNS(JZ));
        JZ = CHN(JZ);
        IF JZ = 0 THEN GO TO ERRORX;
        GO TO LOOPL
RXT1:  S = S+1;
RXT2:  STACK(S) = INFO(JZ)
        T = T+1;
        IF FLAG = 1 THEN TABLE(T) = SCAR;
        ELSE FLAG = 1;
        GO TO CONTROL;
RXT3:  S = S-1;
        TABLE(T) = INFO(JZ);
        GO TO CONTROL;
RXT4:  T = T+1;
        TABLE(T) = SCAR;
RXT5:  S = S-1;
        STACK(S) = INFO(JZ)
        GO TO CONTROL;
RXT6:  T = T-1;
        FLAG = 0;
RXT7:  S = S-2;
        TABLE(T) = INFO(JZ);
        GO TO CONTROL;

Figure 35. Control routine using list structure

The control section of Figure 35 is used in the recognizer of the Ten Statement FORTRAN compiler shown in Appendix A. This control section is fixed and will be the same for any compiler using the Expandable Compiler (EC) design.

The EC design presented here has been improved over the TMC design in that it has reduced the number of control routines and the storage size for the transition matrix. This has been done at the expense of having to do some searching for the correct control routine. In the example the average number of search is about 1.2, assuming each line has an equal probability of being selected. The EC design
also can be easily changed because a system of computer pro-
grams have been written that will build the following TV, N,
CH, INFO, BRNR, and BRNS from a description of the language.
The following is a descriptive summary of the six vectors to be used in the expanded compiler design.

<table>
<thead>
<tr>
<th>Vector Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV</td>
<td>TV contains the integer values corresponding to particular U's and T's to be compared against present value of TABLE for a match</td>
</tr>
<tr>
<td>N</td>
<td>N contains the number of nonempty elements in a row.</td>
</tr>
<tr>
<td>CHN</td>
<td>CHN is used to chain entries together with a common remainder</td>
</tr>
<tr>
<td>BRNR</td>
<td>BRNR contains the number of the control routine to be branched to.</td>
</tr>
<tr>
<td>INFO</td>
<td>INFO contains the integer values to be stored in TABLE or STACK by the control routines.</td>
</tr>
<tr>
<td>BRNS</td>
<td>BRNS contains the number of the semantic routine to be branched to.</td>
</tr>
</tbody>
</table>
In this chapter it is shown that any production in the expanded operator grammar must be one of eleven forms. The eleven forms of the EOG are those generated by the nineteen steps in Chapter III and they are shown in Figure 18. The symbols used in the forms are the U's and T's of the original OG and the Y's, Z's and X's generated for the EOG. The conditions are established to guarantee an unambiguous parse of a sentence in the EOG. The sentential forms of the EOG that can occur in the canonical parse of a sentence are described. It is also shown that all the sentences of the language that can be parsed in the original OG can also be parsed in the EOG, provided the EOG is unambiguous.

The Eleven Forms of a Production in the EOG

In this section it will be shown that if the steps for constructing an EOG are applied to an arbitrary production in an operator grammar the resulting productions in the EOG must be one of the forms shown in Figure 36b. Any production from the original OG must be in one of the forms of Figure 36a.

1.1 $U_i \rightarrow T_m$
1.2 $U_i \rightarrow T_m^s n$
1.3 $U_i \rightarrow U_j$
1.4 $U_i \rightarrow U_j T_m$
1.5 $U_i \rightarrow U_j T_m^s n$

Figure 36a. Production forms in original OG
The approach is to start out with the forms in Figure 36a and state which steps are applicable to each form. After the application of the steps, new forms are created. Steps are again applied to the new forms. The process is repeated until all steps have been applied for the procedure given. The remaining forms should all be the EOG forms described in Figure 36b.

Form 1.1 is already one of the eleven forms and no steps apply to it.

Form 1.3 is deleted in step 19 but is required in steps 13, 14, 15, 16, 17 and 18 before it is deleted. Steps 13 through 18 create productions from productions already in the EOG form. The new production is in the same form as the
original EOG form, therefore, it is not necessary to be concerned with form 1.3 or steps 13 through 18.

To aid in the discussion, the intermediate forms a production may take are denoted by the step number applicable to it. These are called non-EOG forms and are shown in Figure 37.

\[
\begin{align*}
U_i & \rightarrow T_m \text{sn} & (1) \\
U_i & \rightarrow Y_h T_m \text{sn} & (2,6,10) \\
U_i & \rightarrow U_j^m & (3) \\
U_i & \rightarrow U_j T_m \text{sn} & (4) \\
U_i & \rightarrow Z_r T_m \text{sn} & (5) \\
U_i & \rightarrow Y_h U_j T_m \text{sn} & (7) \\
U_i & \rightarrow Y_h Z_h j T_m \text{sn} & (8) \\
U_i & \rightarrow Y_h X_h j \text{sn} & (9) \\
U_i & \rightarrow Y_h U_j & (11) \\
U_i & \rightarrow Y_h Z_h j & (12)
\end{align*}
\]

Figure 37. Non-EOG forms

The forms in Figure 36b are called EOG forms. Figure 38 shows the possible results when steps 1 and 2 are applied to the form 1.2 in Figure 36a. An (*) denotes that the form is an EOG form. Since step 2 is applied until there are no more productions of that form, there are only two non-EOG forms created, (11) and (7). Figure 39a shows that the application of step 3 to the form 1.4 in Figure 36a creates two EOG forms. Figure 39b shows that the application of steps 4 and 5 to
Step 1

\[ U_1 \rightarrow Y_h T_m s_n \] (2)

\[ U_1 \rightarrow Y_h T_m \] (\#)

\[ U_1 \rightarrow Y_h U_j \] (11)

\[ U_1 \rightarrow Y_h U_j T_m s_e \] (7)

\[ Y_h \rightarrow T_m \] (\#)

Step 2

\[ U_1 \rightarrow Y_h T_m s_n \] (2)

\[ U_1 \rightarrow Y_h T_m \] (\#)

\[ U_1 \rightarrow Y_h T_m s_n \] (2)

\[ U_1 \rightarrow Y_h U_j \] (11)

\[ U_1 \rightarrow Y_h U_j T_m s_e \] (7)

\[ Y_h \rightarrow Y_j T_m \] (\#)

Figure 38. Steps 1 and 2 applied
Step 3

\[ U_1 \rightarrow U_j T_m (3) \rightarrow Z_{rj} \rightarrow U_j \quad (*) \]

\[ U_1 \rightarrow Z_{rj} T_m \quad (*) \]

Figure 39a. Step 3 applied

Step 4

\[ U_1 \rightarrow U_j T_{m n} (4) \rightarrow Z_{rj} \rightarrow U_j \quad (*) \]

\[ U_1 \rightarrow Z_{rj} T_{m n} \quad (5) \]

Step 5

\[ Y_n \rightarrow Z_{rj} T_m \quad (*) \]

\[ U_1 \rightarrow Y_n T_{m n} \quad (6) \]

\[ U_1 \rightarrow Z_{rj} T_{m n} (5) \rightarrow U_1 \rightarrow Y_n T_m \quad (*) \]

\[ U_1 \rightarrow Y_n U_j \quad (11) \]

\[ U_1 \rightarrow Y_n U_j T_{m n e} \quad (7) \]

Figure 39b. Steps 3, 4 and 5 applied
form 1.5 creates forms (6), (7) and (11). The rule for step 6 is the same as step 2 shown in Figure 38 which creates forms (7) and (11). Therefore, after the application of the first six steps the only non-EOG productions are in the form of (7) and (11).

Figure 40 shows the application of steps 7, 8, 9 to form (7). The results are non-EOG forms (10), (11) and (7). Since step 10 is applied until there are no more productions of form 10, there are only two non-EOG forms remaining, forms (11) and (7). The sequence of steps 7, 8, 9, 10 are repeated until there are no more productions of form (7).

Figure 41 shows the application of steps 11 and 12 which results in the creation of three EOG forms. There are no more non-EOG forms remaining after step 12.
Figure 40. Steps 7, 8 and 9 applied
Step 11

\[ U_i \rightarrow Y_{h}U_{j}(11) \rightarrow Z_{hj} \rightarrow U_j \quad (*) \]
\[ U_i \rightarrow Y_{h}Z_{hj} \quad (12) \]

Step 12

\[ U_i \rightarrow Y_{h}Z_{hj}(12) \quad X_{hj0} \rightarrow Z_{hj} \quad (*) \]
\[ U_i \rightarrow Y_{h}X_{hj0} \quad (*) \]

Figure 41. Steps 11 and 12 applied
Conditions for an Unambiguous EOG

Sufficient conditions are given for an EOG to be unambiguous. A proof is given that the conditions suffice for parsing or recognizing sentences of the EOG. The recognizer is of the left-to-right bottom-up variety which consists of reducing a sentence to the symbol $U_1$.

The EOG has productions of one of the forms given in Figure 42.

1. $Z_{hj} \to U_j$
2. $U_i \to Y_{h}X_{hj}m$
3. $Y_i \to Y_{h}X_{hj}m$
4. $U_i \to T_m$
5. $Y_i \to T_m$
6. $Y_i \to Y_{j}T_m$
7. $U_i \to Y_{j}T_m$
8. $X_{hj}m \to Z_{hj}T_m$
9. $X_{hj0} \to Z_{hj}$
10. $Y_i \to Z_{hj}T_m$
11. $U_i \to Z_{hj}T_m$

Figure 42. Eleven forms of the EOG productions

From Figure 42 any possible prime phrase of the EOG must have one of the forms in Figure 43.

$$U_i \quad Y_{j}T_m$$
$$Y_{h}X_{hj}m \quad Z_{hj}T_m$$
$$T_m \quad Z_{hj}$$

Figure 43. Possible prime phrases for EOG

Let $V$ represent the set of $(U, X, Y, Z)$ which are non-terminal symbols of the EOG. Let $s_0$ be the sentence to be
analyzed. If $s_0$ contains one or more prime phrases, replace the left most prime phrase by the appropriate $V_j$, yielding the sentential form $s_1$. Iterate the procedure at each step replacing the left most prime phrase of $s_1$ by some $V_j$ to obtain $s_{i+1}$. For an unambiguous parse the process must in a finite number of steps result in $s_m = U_1$.

To reduce a sentence of $N$ symbols to $s_m = U_1$ would imply that productions with two symbols in their right part must be applied $(N-1)$ times. To make the reduction process finite it is necessary to show that the reduction of a single symbol to a single symbol can take place only a fixed number of times.

\[ \text{Figure } 44. \text{ Reduction of single symbols in EOG} \]

\[ \text{Figure } 44 \text{ is derived from the productions in Figure 42 to show the paths a single symbol reduction to a single symbol can take. The paths are labeled with the production numbers. The bottom line shows the symbols in a circle to which the top symbols are reduced and each of these must be combined with the symbol in the box to make further reduction. The longest} \]
path of single symbol reductions before a two symbol reduction
must be made is \( T_m = U_i = Z_{hj} = X_{hj0} \) of length three. The
next step must reduce \( Y_{hj0} \) which is a reduction of one symbol
from \( s_i \) to \( s_{i+1} \) in four steps. Therefore, if a sentence of \( N \)
symbols is to be reduced to \( s_m = U_1 \), the maximum number of
steps is \( 4 \, (N-1) \).

A grammar is unambiguous grammar if it can be shown that
the three requirements of Figure 45 are true.

1. The parse must in a finite number of
   steps result in \( s_m = U_1 \).
2. There exists a unique left most prime
   phrase in \( s_i \) at each step of the parse.
3. The \( V_i \) that replaces the left most prime
   phrase is also unique at each step of
   the parse.

Figure 45. Requirements for an unambiguous grammar

It has already been shown that the EOG must parse in a
finite number of steps. Conditions are now stated for the
EOG, which will allow the EOG to meet the requirements 2 and
3 of Figure 45. If an EOG is generated from an OG and meets
the conditions given, it will be shown that the conditions
are sufficient to guarantee the EOG is unambiguous.

Condition 1. For each pair of \( Y_j \) and \( T_m \), only one of
the four following EOG productions may exist in the EOG.
\[ (1.1) \quad U_i \rightarrow T_m \text{ such that } Y_j \in f(U_i) \]
\[ (1.2) \quad Y_i \rightarrow T_m \text{ such that } Y_j \in f(Y_1) \]
\[ (1.3) \quad Y_i \rightarrow Y_j T_m \]
\[ (1.4) \quad U_i \rightarrow Y_j T_m \]

Furthermore, if the one production that exists in the EOG is (1.1) or (1.4) then there must be only one such \( U_i \). If the one production that exists in the EOG is (1.3) or (1.2) there must be only one such \( Y_i \).

Condition 2. For each pair of \( Z_{hj} \) and \( T_m \), only one of the following four EOG productions may exist in the EOG.

\[ (2.1) \quad X_{hj0} \rightarrow Z_{hj} \text{ such that } X_{hj0} \in f(T_m) \]
\[ (2.2) \quad X_{hjm} \rightarrow Z_{hj} T_m \]
\[ (2.3) \quad Y_i \rightarrow Z_{hj} T_m \]
\[ (2.4) \quad U_i \rightarrow Z_{hj} T_m \]

Furthermore, if the one production that exists in the EOG is (2.1) or (2.2) there must be only one such \( X_{hj0} \) or \( X_{hjm} \).

This is true by construction because the subscripts of the \( X \) reflect the subscripts of the right part of the production. If the one production that exists in the EOG is (2.3) or (2.4), there may be only one such \( Y_i \) or \( U_i \).

Condition 3. For each pair of \( Y_h \) and \( X_{hjm} \), only one of the two EOG productions listed below may exist in the EOG.

\[ (3.1) \quad U_i \rightarrow Y_h X_{hjm} \]
\[ (3.2) \quad Y_i \rightarrow Y_h X_{hjm} \]
Furthermore, if the one production that exists in the EOG
is (3.1), there must be only one $U_i$ and if the one production
is (3.2), there must be only one $Y_i$.

Condition 4. For each pair of $Y_h$ and $U_j$ only one of the
single EOG productions listed below may exist in the EOG.

\[(4.1) \quad Z_{hj} \rightarrow U_j \text{ such that } Y_h \in \mathcal{L}(U_j)\]

Furthermore, if the production \((4.1)\) exists in the EOG, there
must be only one $Z_{hj}$. This is true by construction of the
EOG because the subscripts of the $Z$ matches the subscripts
of the $Y$ and $U$ pair. Condition 4 is true by construction
because there are no productions to conflict with $Z_{hj} \rightarrow U_j$.
Condition 4 is stated here only so the fact can be used later.

Some of the productions in the EOG may not meet the four
conditions stated. This means that the reduction to be made
is not uniquely determined and not that the original OG is
ambiguous. Some methods of circumventing these difficulties
are presented in a later section.

It will now be shown that if conditions 1, 2, 3 and 4
hold in the EOG, the EOG is unambiguous. First, $s_1$ (a sen­tential form) is assumed to be one of the forms shown in
Figure 46. The possible prime phrases for each sentential
form are also shown in Figure 46.
91

Sentential Forms Possible Prime Phrases

1. \( Y_1 Y_2 \ldots Y_{h+1} T_2 \ldots T_m \) \( Y_{h+1} T_1 \)
2. \( Y_1 Y_2 \ldots Y_h Z_{h+1} T_2 \ldots T_m \) \( Z_{h+1} T_1 \)
3. \( Y_1 Y_2 \ldots Y_h X_{h+1} T_2 \ldots T_m \) \( Y_h X_{h+1} T_1 \)
4. \( Y_1 Y_2 \ldots Y_{h+1} U_2 \ldots T_m \) \( U_1 \)

Figure 46. Sentential forms in a canonical parse in the E0G

All the productions that are applicable in Figure 42 are applied to \( s_i \) to obtain all possible forms of \( s_{i+1} \).

Figure 47 shows forms for \( s_{i+1} \).

<table>
<thead>
<tr>
<th>Sentential Forms</th>
<th>Production Applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.1) ( Y_1 Y_2 \ldots Y_{h+1} T_2 \ldots T_m )</td>
<td>5</td>
</tr>
<tr>
<td>(1.2) ( Y_1 Y_2 \ldots Y_h U_2 \ldots T_m )</td>
<td>4</td>
</tr>
<tr>
<td>(1.3) ( Y_1 Y_2 \ldots Y_{h+1} T_2 \ldots T_m )</td>
<td>6</td>
</tr>
<tr>
<td>(1.4) ( Y_1 Y_2 \ldots Y_{h-1} Y_{h+1} T_2 \ldots T_m )</td>
<td>7</td>
</tr>
<tr>
<td>(2.1) ( Y_1 Y_2 \ldots Y_h X_{h+1} T_2 \ldots T_m )</td>
<td>8</td>
</tr>
<tr>
<td>(2.2) ( Y_1 Y_2 \ldots Y_h X_{h} T_1 \ldots T_m )</td>
<td>9</td>
</tr>
<tr>
<td>(2.3) ( Y_1 Y_2 \ldots Y_{h+1} Y_{h+1} T_2 \ldots T_m )</td>
<td>10</td>
</tr>
<tr>
<td>(2.4) ( Y_1 Y_2 \ldots Y_h U_2 \ldots T_m )</td>
<td>11</td>
</tr>
<tr>
<td>(3.1) ( Y_1 Y_2 \ldots Y_{h-1} U_2 \ldots T_m )</td>
<td>2</td>
</tr>
<tr>
<td>(3.2) ( Y_1 Y_2 \ldots Y_h Y_1 \ldots T_m )</td>
<td>3</td>
</tr>
<tr>
<td>(4.1) ( Y_1 Y_2 \ldots Y_{h+1} T_2 \ldots T_m )</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 47. Sentential forms after application of productions
By renumbering the subscripts the forms arrived at for \( s_{i+1} \) are the same as for \( s_i \).

In the recognizer, \( Y_0 \) is placed at the start of each incoming symbol string before beginning the parse.

For the sentence

\[
s = T_1 T_2 T_3 \ldots T_m
\]

\[
s_0 = Y_0 T_1 T_2 \ldots T_m
\]

\( s_0 \) has one of the forms shown in Figure 46, therefore, it has been shown by induction that all the \( s_i \)'s are in the forms shown in Figure 46.

The four sentential forms of Figure 46 are the only ones which can occur during the parsing process. If each condition is applied to its corresponding sentential form in Figure 46, that is, condition 1 to sentential form 1, condition 2 to sentential form 2, etc., each condition guarantees for its corresponding sentential form that the left most prime phrase can be uniquely determined and also that the symbol to which that phrase is reduced is unique. Therefore, the EOG is unambiguous, if it meets conditions 1, 2, 3 and 4.

The sentential forms of Figure 46 are of interest because they show which symbols may appear pair wise in a sentential form. They show that no \( Z \)'s appear to the left of \( U \)'s or \( Y \)'s and that no \( U \)'s appear to the right of \( Z \)'s. These facts were used earlier to establish additional conditions for productions with only one symbol on the right part of a production. The sentential forms show that there may be only
one Z, U, or X in a sentential form.

Since the sentential forms of Figure 46 represent the concatenation of the STACK and part of the TABLE, it shows that the Z's can only exist on top of the STACK, and the U's can only exist as the first element of TABLE. Because the X's are never used in the recognizer of Chapter III, sentential form 3 of Figure 46 never exists in a computer parse. The X's are avoided by applying a unique sequence of two productions at one time.

The sentential forms of Figure 49 are based strictly on a canonical parse in the EOG of a sentence. The recognizer of Chapter III reduces sentences using a canonical parse in the EOG, therefore, the canonical parse is the only parse of interest here.

Relationship Between EOG and Original OG

It is important in the development of the EOG that the EOG is derived such that all the sentences defined by the original OG can also be parsed in the EOG. The above statement is shown in this section to be true, if the EOG is unambiguous. Also it is shown in this section that if the EOG is unambiguous, then the corresponding OG is also unambiguous.

The following statements are used to explain the relationship between the OG and the corresponding EOG. The
94

statements are called lemmas and follow from the construction of the EOG and from the fact that the EOG is unambiguous.

Lemma 1. For each production of the OG $U_k \rightarrow s_n$ where $s_n \neq U_j$ there exists a unique set of productions $(P_1, P_2, \ldots, P_m)$ in the EOG such that $s_n$ reduces into $U_k$ ($U_k \overset{*}{\rightarrow} s_n$) if the EOG is unambiguous.

The $s_n$ is an arbitrary string of $U$'s and $T$'s with no $U$'s adjacent. A parse to reduce $s_n$ to the symbol $U_k$ is performed by applying a set of EOG productions $(P_1, P_2, \ldots, P_m)$. To apply a production $P_j$ at the $i$th step means to replace the right part of $P_j$ in $s_i$ by the left part of $P_j$ to give a new form $s_{i+1}$. The $P_j$'s are productions of the form shown in Figure 42 and their right parts are one of the six phrases shown in Figure 43.

The flow chart in Figure 48 is designed to show for any arbitrary $s_n$, except $s_n = U_j$, what the set of EOG productions will be for reducing $s_n$ to $U_k$. The flow chart scans the string of $U$'s and $T$'s of $s_n$ from left to right, applying EOG productions to the intermediate $s_i$'s until the last $s_i$ is $U_k$. If the first symbol in $s_n$ is a $T$, the flow chart starts at the top left side, and if it is a $U$, the flow chart starts at the bottom left side. The boxes in the flow chart show the intermediate $s_i$'s before and after the application of the EOG production along with the EOG production applied. The
Figure 48. Flow chart for EOG parse
flow chart insures that the parses of $s_n$ are canonical parses. $s_e$ is an arbitrary (possibly empty) string of U's and T's with no U's adjacent.

The step numbers shown are those from Chapter III for the EOG production or productions are created. The flow chart can also be viewed as a method of creating the EOG productions corresponding to a single production in an operator grammar. The word END is used in the flow chart to indicate there are no more U's or T's in $s_n$. Figure 4.9 shows the results of applying the flow chart to the OG production $U_2 \rightarrow T_2 U_3 T_2 T_4 U_5 T_9$ to obtain the equivalent set of EOG productions.

Although Figure 4.8 shows a set of EOG productions for reducing $s_n$ into $U_k$, the question can be asked, is it the only set of productions in the total EOG that will do so? The total EOG is all the EOG productions created for all the productions in the OG. If there was a second set of EOG productions for reducing $s_n$ into $U_k$ then some sentences in the language could be parsed in more than one way with respect to the EOG. This violates the fact that the EOG is unambiguous, therefore, the set of EOG productions created in Figure 4.8 is unique.

Suppose an OG had productions of the form $U_k \rightarrow U_j$. Since the EOG has no productions of the form $U_k \rightarrow U_j$, how can the EOG parse sentences generated from the OG? Let
<table>
<thead>
<tr>
<th>Next Symbol</th>
<th>Step(s)</th>
<th>$s_i$</th>
<th>Production(s) To Be Applied</th>
<th>$s_{i+1}$</th>
<th>$i$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_5$</td>
<td>Start</td>
<td>$T_5s_e$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U_3$</td>
<td>1</td>
<td>$T_5U_3s_e$</td>
<td>$Y_1 \rightarrow T_5$</td>
<td>$Y_1U_3s_e$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$T_2$</td>
<td>7,8</td>
<td>$Y_1U_3T_2s_e$</td>
<td>$Z_{13} \rightarrow U_3$</td>
<td>$Y_1X_{132}s_e$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$T_4$</td>
<td>9</td>
<td>$Y_1X_{132}T_4s_e$</td>
<td>$Y_2 \rightarrow Y_1X_{132}$</td>
<td>$Y_2T_4s_e$</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$U_5$</td>
<td>2</td>
<td>$Y_3T_4U_5s_e$</td>
<td>$Y_3 \rightarrow Y_2T_4$</td>
<td>$Y_3U_5s_e$</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>$T_9$</td>
<td>7,8</td>
<td>$Y_3U_5T_9s_e$</td>
<td>$Z_{35} \rightarrow U_5$</td>
<td>$Y_3X_{359}s_e$</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
<td>$Y_3X_{359}$</td>
<td>$U_2 \rightarrow Y_3X_{359}$</td>
<td>$U_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 49. EOG productions equivalent to $U_2 \rightarrow T_5U_3T_2T_4U_5T_9$
OG1 represent the original OG and let OG2 be a new OG created from OG1. For every production in OG1 with a $U_j$ in it and a production $U_k \rightarrow U_j$ add a new production of the same form with $U_j$ replaced by $U_k$. For example, if OG1 is as follows:

$$U_1 \rightarrow U_2T_3U_3$$
$$U_2 \rightarrow U_4$$
$$U_3 \rightarrow U_5$$

then OG2 would be as follows:

$$U_1 \rightarrow U_2T_3U_3$$
$$U_1 \rightarrow U_4T_3U_3$$
$$U_1 \rightarrow U_2T_3U_5$$
$$U_1 \rightarrow U_4T_3U_5$$

The new OG accepts the same set of sentences in the language as the original OG. The EOG reflects the productions of the new OG. In actual implementation the additions by substitution are made in the EOG and not in the original OG.
Lemma 2. If the EOG is unambiguous, the corresponding OG must also be unambiguous.

It is assumed that the OG is ambiguous, therefore, there exists at least one sentence \( s \) in the language for which there are two different sets of productions \( (0_1, 0_2, \ldots, 0_n) \) and \( (L_1, L_2, \ldots, L_m) \) in the OG that represent a canonical parse of \( s \). Because of Lemma 1, a unique set of EOG productions can replace each of the 0 and L productions, giving two canonical parses of \( s \) with respect to the EOG. This violates the condition that the EOG is unambiguous, therefore, the OG must be unambiguous.

Lemma 3. Every sentence that can be parsed in the original OG can also be parsed in the EOG, if the EOG is unambiguous.

As shown in Lemma 2, if the EOG is unambiguous, then the OG is unambiguous. If the OG is unambiguous then there is a unique set of productions \( (0_1, 0_2, \ldots, 0_n) \) in the OG that performs a canonical parse of every sentence defined by the OG. Because of Lemma 1, a unique set of EOG productions can be substituted for each 0 giving a unique canonical parse in the EOG of every sentence defined by the original OG.
CHAPTER V. IMPLEMENTATION

The functional units of the Expandable Compiler (EC) are shown in Figure 50. The EC design is that of a table driven compiler because it uses a table of vectors and a fixed set of control routines in the recognizer.

<table>
<thead>
<tr>
<th>RECOGNIZER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Routine</td>
</tr>
<tr>
<td>Table of Vectors</td>
</tr>
</tbody>
</table>

| SCANNER |
| SEPANTIC |
| ROUTINES |

Figure 50. Functional units of expandable compiler

The recognizer acts as a main program calling the scanner and semantic routines as they are needed. The scanner and semantic routines are hand coded by the compiler writer as described in Chapter II. The control routine for the recognizer is that described in Figure 35. The table of vectors for the recognizer is created by a system of computer programs called XPAND. XPAND builds the six vectors, TV, N, CHN, BRNR, INFO, and BRNS described in Chapter III. These vectors are used by the control routine in the recognizer to control the parsing process.
Construction of the Tables

XPAND accepts as input a description of the language for which a compiler is to be built or expanded. The description is in the form of productions of an operator grammar along with names of semantic routines associated with particular elements of a production.

A metalanguage is a language which is used to describe (in any way) a language. The name METAGG is given to the metalanguage used here for describing the OG productions along with their semantic names. The basic approach employed by METAGG is similar to Backus Normal Form (BNF).

Metalinguistic names are denoted by a letter followed by a sequence of seven letters or digits. Source elements of the language, such as key words, operators, and punctuation are represented by a non-null sequence of characters enclosed with apostrophes ('). There are other source elements which are recognized by the scanner which are not placed in apostrophes, such as identifiers and constants. These source elements use the symbol (#) as follows:

1. Identifier - represented by #ID
2. Constant - represented by #CONST

Figure 53 shows a description of the statements in Ten Statement FORTRAN described in METAGG. The productions are those of an operator grammar with nonterminal symbols as meta names and terminal symbols as character strings enclosed in
apostrophes ('), or as #ID or as #CONST.

The list of names between the $'s represent the names of semantic routines. The semantic names correspond in sequence to the elements on the right part of the production. The first semantic name in the list corresponds to the first element on the right of a production, etc. At compile time these semantic routines will be executed at the time the corresponding element is used in the parse of a sentence in the language. The names corresponding to different elements are separated by commas. If no semantic function is to be performed for an element, the word NONE should be placed in the corresponding semantic list.

The first input to XPAND is a list of names and symbols used as nonterminal names, terminal names and semantic names. The names are coded as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Nonterminal or meta name</td>
</tr>
<tr>
<td>1</td>
<td>Terminal name or source element</td>
</tr>
<tr>
<td>2</td>
<td>Semantic name</td>
</tr>
</tbody>
</table>

An example of a list is obtained by combining the data of Figures 51 and 52. The -1 at the bottom of Figure 52 in the code column marks the end of the list.

The numbers in the right column are the corresponding integer values used by the recognizer at compile time. The number assigned to the source elements are those which the
Figure 51. Terminal and nonterminal names
Figure 52. Semantic names

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NONE</td>
</tr>
<tr>
<td>2</td>
<td>TYPEP</td>
</tr>
<tr>
<td>3</td>
<td>TYPEI</td>
</tr>
<tr>
<td>4</td>
<td>TABLE</td>
</tr>
<tr>
<td>5</td>
<td>RRED</td>
</tr>
<tr>
<td>6</td>
<td>WRITE</td>
</tr>
<tr>
<td>7</td>
<td>RETURN</td>
</tr>
<tr>
<td>8</td>
<td>STOP</td>
</tr>
<tr>
<td>9</td>
<td>BRANCH</td>
</tr>
<tr>
<td>10</td>
<td>LABEL</td>
</tr>
<tr>
<td>11</td>
<td>STACK</td>
</tr>
<tr>
<td>12</td>
<td>STORE</td>
</tr>
<tr>
<td>13</td>
<td>TEST</td>
</tr>
<tr>
<td>14</td>
<td>TEQ</td>
</tr>
<tr>
<td>15</td>
<td>THE</td>
</tr>
<tr>
<td>16</td>
<td>TLE</td>
</tr>
<tr>
<td>17</td>
<td>TLT</td>
</tr>
<tr>
<td>18</td>
<td>TGE</td>
</tr>
<tr>
<td>19</td>
<td>TGT</td>
</tr>
<tr>
<td>20</td>
<td>BRANCHT</td>
</tr>
<tr>
<td>21</td>
<td>ADD</td>
</tr>
<tr>
<td>22</td>
<td>SUB</td>
</tr>
<tr>
<td>23</td>
<td>NGE</td>
</tr>
<tr>
<td>24</td>
<td>MUL</td>
</tr>
<tr>
<td>25</td>
<td>DIV</td>
</tr>
<tr>
<td>26</td>
<td>CONST</td>
</tr>
</tbody>
</table>

-1 0
Figure 53. Productions for Ten Statement FORTRAN
scanner must return to the recognizer on detecting the source element. The numbers associated with the semantic name are used as a subscript to a PL/1 label vector. The names in the label vector are the names in Figure 52 and their subscripts are the numbers in the right hand column of Figure 52. This label vector must be put in the compiler by the compiler writer.

The description of the language as shown in Figure 53 follows the list of names as input to the XPAND system. The XPAND system, using the procedures described in Chapter III, builds the six vectors used for controlling the recognizer.

Expanding the Compiler

Suppose the compiler for Ten Statement FORTRAN has been built and the compiler writer wishes to add the new instruction below.

\[
\text{ST} = \text{'JUMP'} \ ((\text{EXP} \cdot \text{CP} \cdot \text{EXP} \cdot) \ #\text{CONST} \cdot, \ #\text{CONST} \cdot, \ #\text{CONST} \\
\text{$NONE, NONE, NONE, COMPARE, NONE, NONE, BRANCHL, BRANCHE, BRANCHH$}
\]

The semantic meaning of this statement is compare the two expressions and jump to one of the three labels at the end if the first expression is low, equal, or high respectively.

This section describes the changes that must be made in the previous compiler for adding the above statement. First, the additional input necessary for changing the control vectors


are described as follows and added to the list in Figure 51.

1  JUMP  33
1  .CP.  34

The following is added to the list in Figure 52.

2  COMPARE  27
2  BRANCHL  28
2  BRANCHE  29
2  BRANCHH  30

The statement in metalanguage form is added to the list in Figure 53. All the data in Figures 51, 52 and 53, along with the additions, are used as input to XPAND to create a new set of control vectors for the recognizer.

The scanner must now return 33 or 34 to the recognizer on detecting JUMP and .CP., respectively. Therefore, the names JUMP and .CP. must be placed in the name table, along with their corresponding numbers, so when the scanner searches the name table, they will be found.

The semantic names COMPARE, BRANCHL, BRANCHE; and BRANCHH, must be added to the end of the name list of the label vector used for semantic names. The compiler writer must then write the four semantic routines and label them with COMPARE, BRANCHL, BRANCHE and BRANCHH, for the recognizer to branch to. Each semantic routine should end with a .

GO TO BRANCHR(BRNR(JZ));.
Error detection and recovery can be handled in two ways. The first approach is to put pseudo productions into XPAND to indicate common error conditions and use the semantic routines associated with these productions as error recovery routines. The second approach is to use the contents at the top of the stack and the next incoming source element to tell what the error was. The compiler writer can use the transition matrix of Chapter III to determine what the set of incoming source elements should be for the symbols that can appear on top of the stack.

**Stretching the Conditions**

In Chapter IV conditions were stated for an EOG to be unambiguous. This section discusses some approaches for solving the problem when these conditions are not met. The XPAND system indicates to the user when the conditions are not met.

The first approach would be to rewrite the original productions causing the problem and rerun the productions through XPAND.

The second approach is to provide a special routine to test the next incoming source element. Suppose, for example, that there are two productions as follows:

\[ X_7 \rightarrow Z_3 \rightarrow T_8 \]
\[ U_5 \rightarrow Z_3 \rightarrow T_8 \]
The problem is that the system does not know which reduction to make when $Z_{32}^T Y_8$ is encountered in a sentential form. Because the recognizer branches to a semantic routine prior to a control routine, a semantic routine called FIXUP1 is associated with $T_8$ in the original productions. FIXUP1 would in actuality be a special control routine. Suppose that after looking at the LL matrix discussed in Chapter III, it is determined that only a $T_3$ can appear to the right of a $Y_7$ and only a $T_4$ can appear to the right of a $U_5$ in a sentential form, FIXUP1 can then use the scanner to test the next source element and tell which reduction to make. FIXUP1 should store and reset the pointer on the scanner. Once FIXUP1 knows which reduction to make, it can perform the proper semantics and then with the correct information in INFO, branch to the proper control routine.

A third and somewhat more elaborate approach, can be used when one or more of the incoming source elements are required to make the correct parsing decision. Again, a special routine called FIXUP2 would be used for making the parsing decision. FIXUP2 would contain two sets of control routines as described in Figure 35. It would use STACK1 and TABLE1 in one routine and STACK2 and TABLE2 in the second routine. STACK1, STACK2, TABLE1 and TABLE2, plus their pointers, would be initialized with the present values in STACK and TABLE.
Each of the two productions in conflict uses its own control routine to perform a reduction. The two control routines are used to continue parsing the two different sentential forms in parallel, using the incoming source elements, until one routine detects an error. Semantic routines would not be called and the pointer on the scanner would be stored and reset on entering and leaving FIXUP2.

The same FIXUP2 routine could be used to settle more than one conflict of two productions in the grammar. The contents of INFO, with the same subscript as the BRNS used for the semantic branch, would be used as a pointer to a list. The list would contain the information normally found in INFO and BRNR for all the sets of productions in conflict. Once the conflict was decided, FIXUP2 would branch back to the normal control routines. The grammar must be checked to be sure that a fixup condition does not arise while the program is in the FIXUP2 routine.
CHAPTER VI. CONCLUSIONS

Through the use of the XPAND system the Expandable Compiler design does produce a compiler that allows a language to be easily changed or expanded. This is done with some restrictions on the language. The language must be expressable in an operator grammar. For the most efficient parsing the EOG generated from the original OG should meet the four conditions given in Chapter IV.

The EC design leaves to the compiler writer the freedom to select the level of code optimization and to develop error recovery facilities as he desires. Since the EC design uses an efficient parsing technique and much of the compiler is hand coded, there is no reason to believe that a carefully hand coded version, (say in assembly language), using the EC design, would not compare favorably in compile time with other hand coded compilers.

Although the size of a compiler using the EC design would be smaller than a transition matrix compiler, it is likely to be larger than other hand coded compilers. An area for further research would be to develop methods for reducing the size of the control vectors. The control vectors are obtained from a transition matrix which can be expressed as a state diagram. No efforts have been made to minimize the number of states in the state diagram. The problem is complicated by the fact that elements that look the same in the
transition matrix may have different semantic routines associated with them.

In the recognizer the average number of searches to find the correct element in the list structure is a function of the integer values assigned to the terminal and non-terminal names as shown in Figure 51. The overall compile time could be improved if an algorithm were developed that would reduce the average search time of the list structure. The compiler writer can work on this by hand by rearranging the columns in the transition matrix associated with the control vector. He can then rerun XPAND with a new set of integer values associated with terminal and nonterminal names.

An advantage of the EC design is the simplicity of the overall design. The compiler can be easily broken down into parts which for a large compiler could be assigned to different people to write.
BIBLIOGRAPHY


I want to express my deep appreciation to Dr. Dale Grosvenor for his many hours of helpful discussion and guidance in the research for and the preparation of this dissertation. Special thanks to Dr. Robert J. Lambert for his guidance in my graduate work and his assistance in obtaining a graduate assistantship on the Themis Project.

I am deeply grateful to my wife, Mary Kay, for the many hours she devoted in typing this dissertation, her endurance and patience.

Finally I want to thank the Iowa State University thesis staff for their assistance in thesis editing and final acceptance.
APPENDIX: TEN STATEMENT FORTRAN COMPILER

The Appendix contains a Ten Statement FORTRAN compiler. The compiler accepts as input the ten statements listed below.

REAL variable
INTEGER variable
READ, variable
variable = expression
IF (expression reop expression) GO TO stno
GO TO stno
stno CONTINUE
PRINT, variable
RETURN
END

The output of the compiler is IBM 360 assembly language instructions and some special macros. The special macros are READ, PRINT, and RESTORE.
/* TEN STATEMENT FORTRAN COMPILER */

COMP: PROC OPTIONS(MAIN);
  DCL C(72) CHAR(1) STATIC;
  DCL CARD CHAR(80) STATIC;
  DCL SNAME CHAR(8) STATIC;
  DCL BLANK8 CHAR(8) STATIC INIT(' ');
  DCL BLANK20 CHAR(20) STATIC INIT('* * *');
  DCL ANAME CHAR(8) STATIC;
  DCL CNUM CHAR(20) STATIC;
  DCL (I, IRT, ICT, IFD, ILP, IL, IR, IT, IX, IS, IY, IP, K1, K2, LNG, OLDIL) FIXED BIN(15) STATIC;
  DCL SYMBOL CHAR(46) STATIC INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZOl23456789.*+-/(=),');
  DCL VEC(46) FIXED BIN(15) STATIC INIT(1, (4) 2, 5, (21) 2, (10) 1, 3, 4, 6, 7, 7, (5) 8);
  DCL VECR(46) FIXED BIN(15) STATIC INIT((38) 0, 11, 9, 10, 12, 14, 8, 15, 16);
  DCL NAME(300) CHAR(8) STATIC INIT('REAL', 'INTEGER', 'READ', 'IF', 'GO', 'TO', 'CONTINUE', 'PRINT', 'RETURN', 'END', 'EQ', 'NE', 'LE', 'LT', 'GE', 'GT');
  DCL (TYPMOOOI, LI) FIXED BIN(15) STATIC;
  DCL NUMB (300) FIXED BIN(15) STATIC INIT(17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, (28) 0);
  DCL STOPS FIXED BIN(15) STATIC INIT(33);
  DCL LABN CHAR(4) STATIC;
  DCL LABNE CHAR(8) STATIC;
  DCL BINST CHAR(6) STATIC;
  DCL BRANCHS(0:26) LABEL INIT(NONE, NONE, TYPR, TYPI, TABEL, READ,
WRITE, RETURN, STUP, BRANCH, LABEL, STACKID, STORE, TEST, TEQ, TNE, TLE, TLT, TGE, TGT, BRNCHT, ADD, SUB, NEG, MUL, DIV, CONST;
DCL BRNCHR(7) LABEL INIT(RXT1, RXT2, RXT3, RXT4, RXT5, RXT6, RXT7);
DCL CT(2) CHAR(2) STATIC INIT("'F',"'E'");
DCL HC CHAR(1) STATIC INIT(****);
DCL (CNAME, NTEMP) CHAR(8) STATIC;
DCL (FLAG, ITP, IKC, JZ, JS, JT, KIS, MAXT, S, T, TYPE) FIXED BIN(15) STATIC;
DCL (STACK(20), TABLE(20)) FIXED BIN(15) STATIC;
DCL STKID(8) STATIC;
DCL TABLEC(30) CHAR(20) STATIC;
DCL TEMP(30) BIT(1) STATIC;
DCL (BS, BR) FIXED BIN(15) STATIC;
DCL STKTYPE(30) FIXED BIN(15) STATIC;

/* THE VECTORS TV, N, CHN, INFO, BRNS AND BRNR ARE CREATED BY XPAND */
/* BECAUSE OF THE LARGE NUMBER OF VALUES WITH WHICH THEY ARE TO */
/* BE INITIALIZED, THEY HAVE BEEN OMITTED TO SAVE SPACE */
/* THE VECTORS ARE DECLARED FIXED BINARY(15) */

ON ENDFILE(SYSIN) GO TO ENDT;
ILP=16;
KIS=0;
ITP=0;
IKC=0;
LI=0;
LAST:
DO I=1 BY 1 TO 30;
TEMP(I)="0"B;
END LAST;

/* RECOGNIZER */

READX:
GET EDIT(CARD) (COL(1), A(80));
PUT EDIT(CARD) (COL(1), A(80));
GET STRING(CARD) EDIT((C(I) DO I=1 BY 1 TO 72)) (72 (A(1)));
IX=1; I=1; IS=0;
T=1; S=1; FLAG=1;
STACK(1)=1;
TABLE(1)=SCANR;

CONTRL: JZ=STACK(S) + MOD(TABLE(T),N(STACK(S)));

LOOP1: IF TV(JZ)=TABLE(T) THEN GO TO BRANCHS(BRNS(JZ));
JZ=CHN(JZ);
IF JZ=0 THEN GO TO ERRORX;
GO TO LOOP1;

/***** CONTROL ROUTINES FOR RECOGNIZER ******/

RXT1: S=S+1;
RXT2: STACK(S)=INFO(JZ);
T=T+1;
IF FLAG=1 THEN TABLE(T)=SCANR;
    ELSE FLAG=1;
    GO TO CONTRL;

RXT3: S=S-1;
TABLE(T)=INFO(JZ);
GO TO CONTRL;

RXT4: T=T+1;
TABLE(T)=SCANR;

RXT5: S=S-1;
STACK(S)=INFO(JZ);
GO TO CONTRL;

RXT6: T=T-1;
FLAG=0;

RXT7: S=S-2;
TABLE(T)=INFO(JZ);
GO TO CONTRL;

ERRORX: PUT EDIT('$\$') (SKIP, COL(I), A);
PUT EDIT(* ERROR IN LAST SOURCE ELEMENT BEFORE THE $')
    (COL(I), A);
GO TO READX;

/***** SCANNER ******/
SCANNR: PROCEDURE RETURNS (FIXED BINARY (15));
DCL LABV(117) LABEL INIT(SCAN,RT8,RT1,RT6,RT8,RT9,(2)RT12,
ERROR,(2)OUT3,SCAN,RT2,(4)OUT3,ERROR,(2)OUT2,SCAN,OUT2,RT3,
(3)OUT2,ERROR,RT4,ERROR,RT5,(3)ERROR,RT4,(2)ERROR,(2)ERROR,
RT5,(6)ERROR,(2)OUT1,SCAN,(5)OUT1,ERROR,ERROR,RT7,RT2,ERROR,
RT7,(4)ERROR,ERROR,SCAN,ERROR,RT10,SCAN,(4)ERROR,OUT5,(2)SCAN,
OUT5,SCAN,(3)OUT5,ERROR,(5)OUT7,RT11,(2)OUT7,ERROR,(8)OUT4,
ERROR,(5)OUT6,ERROR,(2)OUT6,ERROR,(8)OUT8,ERROR);
GO TO LABV(I+IS);
SCAN: I=I+1;
IF I>72 THEN RETURN(STOPS);
IY=INDEX(SYMBOL,C(I));
IX=VEC(IY);
GO TO LABV(I+IS);
RT1: IS=9; K1=I; GO TO SCAN;
RT2: IS=18; GO TO SCAN;
RT3: IS=27; GO TO SCAN;
RT4: IS=36; GO TO SCAN;
RT5: IS=45; GO TO SCAN;
RT6: IS=54; K1=I; GO TO SCAN;
RT7: IS=63; GO TO SCAN;
RT8: IS=72; K1=I; GO TO SCAN;
RT9: IS=81; GO TO SCAN;
RT10: IS=90; GO TO SCAN;
RT11: IS=99; GO TO SCAN;
RT12: IS=108; IP=IY; GO TO SCAN;
OUT1: K2=I-1; LNG=K2-K1+1;
CNUM=BLANK20; IT=7; ICT=2;
SUBSTR(CNUM,1,LNG)=SUBSTR(CARD,K1,LNG);
RETURN(IT);
OUT2: K2=I-1; LNG=K2-K1+1;
CNUM=BLANK20; IT=7; ICT=2;
SUBSTR(CNUM,1,LNG)=SUBSTR(CARD,K1,LNG);
RETURN(IT);
OUT3: K2=I-1; LNG=K2-K1+1;
CNUM=BLANK20; IT=7; ICT=1;
SUBSTR(CNUM,1,LNG)=SUBSTR(CARD,K1,LNG);
RETURN(IT);

OUT4: OUT5: K2=I-1; LNG=K2-K1+1;
    IF LNG>8 THEN LNG=8;
    SNAME=BLANK8;
    ANAME=BLANK8;
    SUBSTR(ANAME,1,LNG)=SUBSTR(CARD,K1,LNG);
    SUBSTR(SNAME,9-LNG,LNG)=SUBSTR(CARD,K1,LNG);
    CALL ISEARCH;
    RETURN(IT);

OUT6: IT=13; RETURN(IT);
OUT7: IT=11; RETURN(IT);
OUT8: IT=VECR(IP); RETURN(IT);
ERROR: PUT SKIP;
    PUT EDIT('*$' COL(I),A);
    GO TO SCAN;
    END SCANR;

ISEARCH: PROC;
    DCL I FIXED BIN(15) STATIC;
LAB1: DO I=1 BY 1 TO ILP;
    IF NAME(I)=SNAME THEN DC; IT=NUMB(I);
        IFD=1; TYPE=TYPM(I);
        RETURN; END;
    END LAB1;
    ILP=ILP+1; NAME(ILP)=SNAME; NUMB(I)=6;
    TYPM(I)=2; IT=6; IFD=0;
    RETURN;
    END ISEARCH;

/***** SEMANTIC ROUTINES  *****/
NONE: GO TO BRNCHR(BRNR(JZ));
TYPR: TYPE=1;
    GO TO BRNCHR(BRNR(JZ));
TYPI: TYPE=2;
    GO TO BRNCHR(BRNR(JZ));
TABEL: ILP=ILP+1;
NAME(ILP)=SNAME;
NUMB(ILP)=6;
TYPM(ILP)=TYPE;
GO TO BRNCHR(BRNR(JZ));

READ: PUT EDIT('READ ',SNAME,,TYPE,'')
      (COL(10),A,A(8),A,F(1),A);
GO TO BRNCHR(BRNR(JZ));

WRITE: PUT EDIT('PRINT ',SNAME,,TYPE,'')
      (COL(10),A,A(8),A,F(1),A);
GO TO BRNCHR(BRNR(JZ));

RETURN: PUT EDIT('RESTORE') (COL(10),A);
GO TO BRNCHR(BRNR(JZ));

STUP: DO J=1 BY 1 TO IKC;
   CNAME=*CONTR INTEGER(J);
   PUT EDIT(CNAME,*DC*,TABLEC(J))
      (COL(10),A,A(8),X(1),A,X(4),A(20));
END STUP;

STOP1: DO J=1 BY 1 TO MAXT;
   CNAME=*TEMP* INTEGER(J);
   PUT EDIT(CNAME,*DS*,F)
      (COL(10),A,A(8),X(1),A,X(4),A);
END STOP1;
GO TO ENDT;

TEST: IF STKTYPE(KIS-1) STKTYPE(KIS) THEN GO TO DEED;
   TYPE=STKTYPE(KIS);
IF TYPE=2 THEN GO TO TEST1;
   PUT EDIT('L 4*,STKID(KIS-1),
   'L 5*,STKID(KIS),'CR 4,5')
      (COL(10),A,A(8),COL(10),A,A(8),COL(10),A);
   KIS=KIS-2;
GO TO BRNCHR(BRNR(JZ));

TEST1: PUT EDIT('LE 4*,STKID(KIS-1),
   'LE 2*,STKID(KIS),'CDR 4,2')
      (COL(10),A,A(8),COL(10),A,A(8),COL(10),A);
   KIS=KIS-2;
GO TO BRNCHR(BRNR(JZ));
TEQ: BINST='BE ";
GO TO BRNCHR(BRNR(JZ));
TNE: BINST='BNE ";
GO TO BRNCHR(BRNR(JZ));
TLE: BINST='BNH ";
GO TO BRNCHR(BRNR(JZ));
TLT: BINST='BL ";
GO TO BRNCHR(BRNR(JZ));
TGE: BINST='BNL ";
GO TO BRNCHR(BRNR(JZ));
TGT: BINST='BH ";
GO TO BRNCHR(BRNR(JZ));
LABEL: LABN=SUBSTR(CNUM,1,4);
CALL SERLAB;
    PUT EDIT(LABNE,'EQU') (COL(1),A(8),X(2),A);
    GO TO BRNCHR(BRNR(JZ));
BRANCH: BINST='B ";
BRNCHT: LABN=SUBSTR(CNUM,1,4);
CALL SERLAB;
    PUT EDIT(BINST,LABNE) (COL(10),A(6),X(2),A(8));
    GO TO BRNCHR(BRNR(JZ));
SERLAB: PROC;
    DCL I FIXED BIN(15) STATIC;
    DCL LABNM(50) CHAR(8) STATIC;
    DCL LNAME(50) CHAR(4) STATIC;
LAB1: DO I=1 BY 1 TO LI;
    IF LABN=LNAME(I) THEN DO; LABNE=LABNM(I);
        RETURN; END;
END LAB1;
    LI=LI+1;
    LNAME(LI)=LABN;
    LABNE='LAB'||INTEGER(LI);
    LABNM(I)=LABNE;
    RETURN;
END SERLAB;
STACKID: KIS=KIS+1;
STKID(KIS)=ANAME;
STKTP(KIS)=2;
GO TO BRNCHR(BRNR(JZ));

STORE:  IF STKTP(KIS-1)=STKTP(KIS) THEN GO TO ERRORT;
        IF TEMP(KIS) THEN DO; ITP=ITP-1;
                          TEMP(KIS)='0'B;
                          END;
        PUT EDIT(*MVC ',STKID(KIS-1),' ,STKID(KIS))
                       (COL(10),A,A(8),A,A(8));
        GO TO BRNCHR(BRNR(JZ));

ADD:    IF STKTP(KIS-1)=STKTP(KIS) THEN GO TO ERRORT;
        TYPE=STKTP(KIS);
        IF TEMP(KIS) THEN DO; ITP=ITP-1; TEMP(KIS)='0'B; END;
        IF TEMP(KIS-1) THEN GO TO ADD1;
        ITP=ITP+1; MAXT=ITP;
        TEMP(KIS-1)='0'B;

ADD1:   NTEMP=*TEMP||INTEGER(ITP);
        IF TYPE=2 THEN GO TO ADD2;
        PUT EDIT(*L 4,*STKID(KIS-1),'A 4,*STKID(KIS),
                       *ST 4,*NTEMP)
                       (COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));

RET1:   KIS=KIS-1;
        STKID(KIS)=NTEMP;
        STKTP(KIS)=1;
        GO TO BRNCHR(BRNR(JZ));

ADD2:   PUT EDIT(*LE 4,*STKID(KIS-1),'AE 4,*STKID(KIS),
                       *STE 4,*NTEMP)
                       (COL(10),A,A(8),COL(40),A,A(8),COL(70),A,A(8));

RET2:   KIS=KIS-1;
        STKID(KIS)=NTEMP;
        STKTP(KIS)=2;
        GO TO BRNCHR(BRNR(JZ));

SUB:    IF STKTP(KIS-1)=STKTP(KIS) THEN GO TO ERRORT;
        TYPE=STKTP(KIS);
        IF TEMP(KIS) THEN DO; ITP=ITP-1; TEMP(KIS)='0'B; END;
        IF TEMP(KIS-1) THEN GO TO SUB1;
ITP=ITP+1; MAXT=ITP;
TEMP(KIS-1)="1'B;
NTEMP=*TEMP||INTEGER(ITP);
SUB1: IF TYPE=2 THEN GO TO SUB2;
    PUT EDIT('L 4,*,STKID(KIS-1),*S 4,*,STKID(KIS),
            'ST 4,*NTEMP)
            (COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));
    GO TO RET1;
SUB2: PUT EDIT('LE 4,*,STKID(KIS-1),*SE 4,*,STKID(KIS),
            'ST 4,*NTEMP)
            (COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));
    GO TO RET2;
MUL: IF STKTYPE(KIS-1)=STKTYPE(KIS) THEN GO TO ERRORT;
    TYPE=STKTYPE(KIS);
    IF TEMP(KIS) THEN DO; ITP=ITP-1; TEMP(KIS)="0'B; END;
    IF TEMP(KIS-1) THEN GO TO MUL1;
    ITP=ITP+1; MAXT=ITP;
    TEMP(KIS-1)="1'B;
MUL1: NTEMP=*TEMP||INTEGER(ITP);
    IF TYPE=2 THEN GO TO MUL2;
    PUT EDIT('L 4,*,STKID(KIS-1),*M 4,*,STKID(KIS),
            'ST 5,*NTEMP)
            (COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));
    GO TO RET1;
MUL2: PUT EDIT('LE 4,*,STKID(KIS-1),*ME 4,*,STKID(KIS),
            'ST 5,*NTEMP)
            (COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));
    GO TO RET2;
DIV: IF STKTYPE(KIS-1)=STKTYPE(KIS) THEN GO TO ERRORT;
    TYPE=STKTYPE(KIS);
    IF TEMP(KIS) THEN DO; ITP=ITP-1; TEMP(KIS)="0'B; END;
    IF TEMP(KIS-1) THEN GO TO DIV1;
    ITP=ITP+1; MAXT=ITP;
    TEMP(KIS-1)="1'B;
DIV1: NTEMP=*TEMP||INTEGER(ITP);
IF TYPE=2 THEN GO TO DIV2;
PUT EDIT('L 4,*,STKID(KIS-1),*M 4,*STKID(KIS),
'SE 5,*NTEMP)
(COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));
GO TO RET1;

DIV2: PUT EDIT('LE 4,*,STKID(KIS-1),*DE 4,*STKID(KIS),
'STE 4,*NTEMP)
(COL(10),A,A(8),COL(10),A,A(8),COL(10),A,A(8));
GO TO RET2;

NEG: TYPE=STKYP(KIS);
IF TEMP(KIS) THEN GO TO NEG1;
ITP=ITP+1; MAXT=ITP;
TEMP(KIS)=1*8;

NEG1: NTEMP=TEMP||INTEGER(ITP);
IF TYPE=2 THEN GO TO NEG2;
PUT EDIT('L 5,*,STKID(KIS-1),*LCR 4,5',
'S 4,*NTEMP)
(COL(10),A,A(8),COL(10),A,COL(10),A,A(8));
GO TO RET1;

NEG2: PUT EDIT('LE 4,*,STKID(KIS-1),*LCER 6,4',
'SE 6,*NTEMP)
(COL(10),A,A(8),COL(10),A,COL(10),A,A(8));
GO TO RET2;

ERROR: PUT EDIT(STKID(KS),STKID(KIS-1),* DO NOT AGREE IN TYPE*)
(COL(1),A(8),X(4),A(8),X(4),A);
GO TO READX;

CONST: IKC=IKC+1;
TABLEC(IKC)=CT(ICT)||SUBSTR(CNUM,1,LNG)||HC;
KIS=KIS+1;
STKID(KIS)="CONT"||INTEGER(IKC);
STKYP(KIS)=ICT;
GO TO BRNCR(BRNR(JZ));

ENDT: END COMP;