THE USE OF TRANSITION MATRICES IN COMPILING

BY

D. GRIES

TECHNICAL REPORT CS 57
MARCH 17, 1967

Supported in part by the Atomic Energy Commission.

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY
THE USE OF TRANSITION MATRICES IN COMPILING

by

D. Gries

Supported in part by the Atomic Energy Commission.
The Use of Transition Matrices in Compiling

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Notation, terminology, basic definitions</td>
<td>3</td>
</tr>
<tr>
<td>3. Operator and augmented operator grammars and languages</td>
<td>5</td>
</tr>
<tr>
<td>4. Parsing a string using an AOG</td>
<td>14</td>
</tr>
<tr>
<td>5. Sufficient conditions for a unique canonical parse</td>
<td>16</td>
</tr>
<tr>
<td>6. The transition matrix and stack</td>
<td>18</td>
</tr>
<tr>
<td>7. An example of a parse</td>
<td>25</td>
</tr>
<tr>
<td>8. Representation of nonterminals in the stack</td>
<td>27</td>
</tr>
<tr>
<td>9. Other uses of transition matrices</td>
<td>28</td>
</tr>
<tr>
<td>10. Summary</td>
<td>39</td>
</tr>
<tr>
<td>References</td>
<td>41</td>
</tr>
<tr>
<td>Appendix A. An ALGOL-like grammar and associated matrix</td>
<td>42</td>
</tr>
</tbody>
</table>
The Use of a Transition Matrix in Compiling

1. **Introduction.**

The construction of efficient parsing algorithms for programming languages has been the subject of many papers in the last few years. Techniques for efficient parsing and algorithms which generate the parser from a grammar or phrase structure system have been derived. Some of the well-known methods are the precedence techniques of Floyd [4] and Wirth and Weber [10], and the production language of Feldman [3]. Perhaps the first such discussion was by Samelson and Bauer [9]. There the concept of the push-down stack was introduced, along with the idea of a **transition matrix**. A transition matrix is just a switching table which lets one determine from the top element of the stack (denoting a row of the table) and the next symbol of the program to be processed (represented by a column of the table) exactly what should be done. Either a reduction is made in the stack, or the incoming symbol is pushed onto the stack.

Considering its efficiency, the transition matrix technique does not seem to have achieved much attention, probably because it was not sufficiently well-defined. The purpose of this paper is to define the concept more formally, to illustrate that the technique is very efficient, and to describe an algorithm which generates a transition matrix from a suitable grammar. We will also describe other uses of transition matrices besides the usual ones of syntax checking and compiling.

We will require that the set of productions \( \{ U_i ::= x_i \} \) form an **operator grammar** (Floyd [4]), which means that no production has
the form $U ::= xV_1V_2y$ for strings $x,y$ and nonterminal symbols $V_1$ and $V_2$. This restriction is not necessary in order to use a transition matrix. One may also describe suitable conditions for the general phrase structure grammar \{U_1 ::= x_1\} which allow the use of a transition matrix. The restriction to operator grammars is a rather natural way to reduce the size of storage necessary to implement the technique. The syntax of the usual ALGOL-like languages can easily be represented by such a grammar.

We emphasize that the use of a transition matrix is just another technique, though a very efficient one, for parsing sentences of a suitable (programming) language.

Section 2 introduces the notation and terminology. Sections 3 through 5 are devoted to discussing sufficient conditions for a unique canonical parse which enable us to use a transition matrix. These conditions are of course closely related to those derived by Floyd [4], Wirth and Weber [10], and Eickel et al [2]. Sections 6 and 7 explain the technique and go through an example in detail. In Sections 8 and 9 practical examples and applications are discussed. Appendix A gives Floyd's ALGOL-like grammar ([4]) and the associated matrix and subroutines. All examples were produced by an algorithm, written in Extended ALGOL [11], on the B5500 at Stanford.

The author is indebted to Jerome Feldman and Niklaus Wirth for their critical comments on this manuscript. This work was partially supported by the U.S. Atomic Energy Commission.
2. Notation, terminology, basic definitions.

Let \( V \) be a given set: the vocabulary. Elements of \( V \) are called symbols and are denoted here by capital Latin letters, \( S \), \( T \), \( U \), etc. Finite sequences of symbols - including the empty sequence (\( \Lambda \)) - are called strings and are denoted by small Latin letters \( u \), \( v \), \( y \), \( z \), etc. The set of all strings over \( V \) is denoted by \( V^* \).

If \( z = xy \) is a string, \( x \) is a head and \( y \) a tail of \( z \).

A production or syntactic rule \( \varphi : U ::= x \), is an ordered pair consisting of a symbol \( U \) and a nonempty string \( x \). \( U \) is called the left part and \( x \) the right part of \( \varphi \). We assume that \( U \neq x \).

Let \( \mathcal{G} \) be a finite set of productions \( \varphi_1, \ldots, \varphi_n \). \( y \) directly -produces \( z \) (\( y \rightarrow z \)) and conversely \( z \) directly reduces into \( y \), if and only if there exist strings \( u, v \) such that \( y = uuv \), \( z = uvv \), and the production \( U ::= x \) is an element of \( \mathcal{G} \).

\( y \) produces \( z \) (\( y \not\rightarrow z \)) and conversely \( z \) reduces into \( y \), if and only if there exist strings \( x_0, \ldots, x_n \) such that \( y = x_0 \), \( x_n = z \) and

\[ x_{i-1} \rightarrow x_i \quad (i = 1, \ldots, n; \quad n \geq 1) \]

\( z \) is also said to be a derivation of \( y \).

Let \( \mathcal{G} \) be a set of productions \( \varphi_1, \ldots, \varphi_n \). If \( V \), the vocabulary, contains exactly one symbol \( A \) which occurs in no right part of a production, and a non empty set \( \mathcal{B} \) of symbols which appear only in the right part of productions, then \( \mathcal{G} \) is a phrase structure grammar. The symbols of \( \mathcal{B} \) are called terminal or basic symbols and are denoted...
by capital letters $T, T_1, T_2$. The letters $U, V$ always denote symbols in $\mathcal{V}-\mathcal{S}$ and are called nonterminal symbols.

$x \in \mathcal{V}^*$ is called a sentential form of $\mathcal{S}$ if either $A = x$ or $A \Rightarrow x$. The set of sentences $x$ - i.e., the set of sentential forms consisting only of terminal symbols - constitutes the phrase structure language $L_\mathcal{S}$, that is:

$$(2.1) \quad L_\mathcal{S} := \{ x \mid A \Rightarrow x \land x \in \mathcal{S}^* \}$$

Without restricting the set of phrase structure languages we shall assume that

$$(2.2) \quad U \Rightarrow U \quad \text{for any} \quad U \in \mathcal{V}-\mathcal{S};$$

$$(2.3) \quad \text{if} \quad U_1 \Rightarrow U_k, \quad \text{then the sequence}$$

$$U_1 ::= U_2, \ U_2 ::= U_3, \ldots, U_n ::= U_k$$

is unique; and

$$(2.4) \quad \text{every symbol may be used in deriving some sentence: for each symbol} \quad X \in \mathcal{V} \text{ there exists strings} \quad x, z, t \quad \text{such that} \quad A \Rightarrow xXz \quad \text{and either} \quad X \in \mathcal{S} \quad \text{or} \quad X \Rightarrow t \quad \text{where} \quad t \in \mathcal{S}^*.$$
the parse \( \varphi_1, \varphi_2, \ldots, \varphi_n \) of \( z_o \) into \( U \) is canonical if and only if for \( j = 1, \ldots, n \) \( x_k \) is not contained in \( u_j \) for all \( k > j \).

Every parse has a unique canonical form (simply rearrange the productions to form a canonical parse), but in an ambiguous grammar there exists more than one canonical parse for some sentence. An unambiguous grammar is a phrase structure grammar such that for every string \( x \in L_g \) there exists exactly one canonical parse of \( x \) into the symbol \( A \).

It has been shown that there exists no algorithm which decides whether an arbitrary grammar is unambiguous. However, a sufficient condition for a grammar to be unambiguous is subsequently derived, and a method is explained which determines whether a given grammar satisfies this condition.

3. Operator and augmented operator grammars and languages.

If no production \( \varphi_1 \) of the phrase structure grammar \( \mathcal{G} \) takes the form \( U ::= xV_1V_2y \) for some (possibly empty) strings \( x, y \) and nonterminal symbols \( V_1 \) and \( V_2 \), then (Floyd [4]) \( \mathcal{G} \) is called an operator grammar (OG). The phrase structure language \( L_g \) generated by an OG is then called an operator language.

Floyd proved that in an operator grammar no sentential form contains two adjacent non-terminal symbols - i.e., if \( A \Rightarrow x \) then there exists no strings \( x_1 \) and \( x_2 \) and no nonterminals \( V_1 \) and \( V_2 \) such that \( x = x_1V_1V_2x_2 \). The grammar in Figure 1 is not an operator grammar, since \(<\text{IF CLAUSE}>\) and \(<\text{STATEMENT}>\) are both nonterminal.
The grammar in Figure 2, which is equivalent to (generates the same language as) the grammar in Figure 1, is an operator grammar.

When parsing a sentence, at each step the leftmost right part $x$ of a production $U ::= x$ must be detected. Then $x$ is replaced by $U$ and the process is repeated. In order to reduce the number of symbols to be checked at each step, we introduce intermediate reductions. For
instance, although the string

\[
\text{IF} \ <\text{EXPR}> \ \text{THEN} \ <\text{STATE}> \ \text{ELSE} \ <\text{STATE}>
\]

can be reduced directly to \(<\text{STATE}>>\) by the grammar of Figure 2, we want to parse it as is shown below:

This can be achieved by constructing an augmented operator grammar (AOG) \(\mathcal{A}\) corresponding to \(\mathcal{J}\). The augmented operator grammar is useful for describing theoretically the mechanism of the matrix technique to be introduced later. However, so as not to complicate the process too much, one can give mnemonic names to the introduced symbols needed for the intermediate reductions. For instance, in the above diagram \(U_1^*\) may be named "\(<\text{IF}*>\)" , \(U_2^*\) "\(<\text{IF expr THEN}*>\)" , and \(U_3^*\) "\(<\text{IF expr THEN state ELSE}*>\)" . That is, each new symbol is just a representation of the head of the right part of some production. \(\mathcal{A}\) is constructed from \(\mathcal{J}\) by repeating the following step 1 until no longer applicable, then step 2 until no longer applicable, and finally steps 3a
and \(3b\) alternately until no longer applicable. New nonterminal symbols will be introduced into \(\mathcal{V}\) and \(\mathcal{V-\mathcal{B}}\) (but not \(\mathcal{B}\)). Note that all newly introduced symbols are distinguished from the original nonterminals by an asterisk "*".

**step 1:** If there is a production \(U_i ::= T_2 y_1\) (\(y_1\) may be empty), and if \(k\) new symbols \(U_1^*, \ldots, U_k^*\) have been created so far, create a new symbol \(U_{k+1}^*\), replace each production \(U_i ::= T_2 y_i\) (each production whose right part begins with \(T_2\)) by the production \(U_i ::= U^*_{k+1} y_i\), and insert the production \(U_{k+1}^* ::= T_2\) into the grammar.

After step 1 all productions have one of the forms

\[
U_1 ::= U_2, \quad U_1 ::= U_2 T y, \quad U_1 ::= U^* y, \quad U ::= T.
\]

where \(y\) contains no introduced symbol \(U^*\).

**step 2:** If there is a production \(U_1 ::= U_2 T_2 y_1\) (note that \(U_2\) must be one of the original nonterminals of the \(A\)), and if \(k\) new symbols have been created so far, create a new symbol \(U^*_{k+1}\), replace each production \(U_i ::= U_2 T_2 y_i\) (each production whose right part begins with \(U_2 T_2\)) by the production \(U_i ::= U^*_{k+1} y_i\), and insert the production \(U^*_{k+1} ::= U_2 T_2\).

After step 2 all productions have one of the forms

\[
U_1 ::= U_2, \quad U_1 ::= U^* y, \quad U^* ::= U T, \quad U ::= T
\]

where \(y\) contains no introduced symbol \(U^*\).
step 3a: If there is a production \( U_1 ::= U'T_2y \), and if \( k \) new symbols have been created so far, create a new symbol \( U_{k+1}^* \), replace each production \( U_1 ::= U'T_2y_1 \) by \( U_1 ::= U_{k+1}^*y_1 \), and insert the new production \( U_{k+1}^* ::= U_{2}T_2 \).

step 3b: If there is a production \( U_1 ::= U_2U_2T_2y \), and if \( k \) new symbols have been created so far, create a new symbol \( U_{k+1}^* \), replace each production \( U_1 ::= U_2U_2T_2y_1 \) by \( U_1 ::= U_{k+1}^*y_1 \), and insert the new production \( U_{k+1}^* ::= U_2U_2T_2 \).

An AOG has, therefore, only productions of one of the following forms:

\[
U_1 ::= U_2, \quad U_1 ::= U^*, \quad U_1 ::= U^*U_2 \\
U^* ::= T, \quad U^* ::= UT, \quad U_2^* ::= U_1^*T, \quad U_2^* ::= U_1^*U_2T
\]

Figure 3

Note that we differentiate between the original Unstarred NonTerminal Symbols (called UNTS) and the newly created Starred NonTerminal Symbols' (SNTS), which for reasons explained later are also called stack nonterminals.

Again, we have introduced augmented operator grammars and stack nonterminals in order to be able to make intermediate reductions. A stack non-terminal can also be thought of as a representation for the head \( xT \) (ending in a terminal symbol \( T \)) of the right part of a production \( U ::= xTy \) of the original OG. As another example, if \( U_1 ::= T_1T_2T_3T_4T_5 \) is a production of the OG, then the string \( T_1T_2T_3T_4T_5 \)
will be parsed as a sentence of the ACG as follows:

Consider the ACG in Figure 2. After step 1 it will have been changed to

\[
\begin{align*}
\text{Productions in ACG} \\
U_1 & ::= T_1 \\
U_2 & ::= U_1 T_1 \\
U_3 & ::= U_2 U_3 T_4 \\
U_4 & ::= U_3 T_5 \\
U_1 & ::= U_4
\end{align*}
\]

\[
\begin{align*}
\text{Consider the ACG in Figure 2. After step 1 it will have been changed to}
\end{align*}
\]

\[
\begin{align*}
\langle \text{PROC} \rangle & ::= \langle \text{STATEMENT} \rangle \\
\langle \text{IF}\ast \rangle & ::= \text{IF} \\
\langle \text{VARIABLE}\ast \rangle & ::= \text{VARIABLE} \\
\langle \text{PROC} \rangle & ::= \langle \text{IF}\ast \rangle \langle \text{EXPRESSION} \rangle \text{ THEN } \langle \text{STATEMENT} \rangle \\
\langle \text{STATEMENT} \rangle & ::= \langle \text{IF}\ast \rangle \langle \text{EXPRESSION} \rangle \text{ THEN } \langle \text{STATEMENT} \rangle \text{ ELSE } \langle \text{STATEMENT} \rangle \\
\langle \text{STATEMENT} \rangle & ::= \langle \text{VARIABLE}\ast \rangle \text{ := } \langle \text{EXPRESSION} \rangle \\
\langle \text{EXPRESSION} \rangle & ::= \langle \text{EXPRESSION} \rangle \text{ OR VARIABLE} \\
\langle \text{EXPRESSION} \rangle & ::= \langle \text{VARIABLE}\ast \rangle
\end{align*}
\]

Step 2 changes it to
Finally, after step 3 we have the tree of Figure 4:

```
<**VAR**> ::= <**EXPRESS**>
<**EXP**> ::= <**VAR**-
<**EXP**-**VAR**> ::= <**EXP**-
<**VAR**> ::= <**VAR**-
<**VAR**> ::= <**VAR**-
<**IF**-**ELSE*** **IF**-**THEN** ::= <**IF**-
<**IF**-**THEN** ::= <**IF**-
<**IF**-**THEN** ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
<**VAR**> ::= <**VAR**-
<**IF**> ::= <**IF**-
```
We will need the following definition: A string \( y \) is a phrase if

(3.1) \( y \) contains at least one terminal or SNTS; and

(3.2) there exists a production \( U ::= y_1 \) or \( U^* ::= y_1 \) of the ACG where \( y_1 = y \) or \( y_1 = uU_1v \), \( y = uU_2v \) and \( U_1 \Rightarrow U_2 \), for some \( u, v \).

Thus, \( y \) is a phrase if it is the right part of some production of the AOG (except a production of the form \( \cdot U_1 ::= U_2 \)), or if it can be reduced to the right part of some production by a sequence of reductions \( U_1 ::= U_2 \). Given a sentential form \( x = x_1yx_2 \), \( y \) is called a reducible phrase (of \( x \)), if

(3.3) \( y \) is a phrase; and

(3.4) for some \( U \) (or \( U^* \)) as defined in (3.2), the string resulting from replacing the string \( y \) by \( U \) (or \( U^* \)) is a sentential form.

The problem for the compiler, then, is to find the leftmost reducible phrase and to make the correct replacement (reduction).

The following statements, which help to explain the relationship between an OG and the corresponding AOG, follow directly from the construction of the AOG. For lack of a better name, we call them lemmas.

Lemma 1. Each SNTS \( U^* \) appears as the left part of only one production \( U^* ::= x \). The corresponding right part \( x \) appears as the right part of no other production.
Lemma 2. If the SNTS's $U^*$ are numbered in the order in which they were introduced, $U_1^*, U_2^*, \ldots, U_n^*$, and if a production $U_i^* := U_j^* y$ exists in the AOG, then $i > j$.

Lemma 3. For each production $U ::= y$ with $y \notin V-G$ of the AOG there exists a unique set of productions $U_1^* ::= y_1^*, U_2^* ::= U_1^* y_2^*, \ldots, U_n^* ::= U_{n-1}^* y_{n-1}^*, u ::= U_n^* y_n^*$ of the AOG such that $y = y_1^* y_2^* \ldots y_n^*$.

Lemma 3 follows directly from the construction and Lemmas 1 and 2.

Lemma 4 follows directly from Lemma 3.

Lemma 4. If $\varphi_1 y_1^*$ is a canonical parse of a string $x$ relative to the OG, then we get a parse of $x$ relative to the AOG by substituting for each $\varphi_i$ (which is not of the form $U_1 ::= U_2$) the unique set of productions defined in Lemma 3.

Since two different canonical parses $\{\varphi_n\}$ and $\{\lambda_n\}$ of a string must for some $i$ have $\varphi_i \neq \lambda_i$, we have also

Lemma 5. Different canonical parses of a string $x$ with respect to an OG, yield different canonical parses of $x$ with respect to the AOG.

Therefore we have finally

Lemma 6. If an AOG is unambiguous, the corresponding OG must also be unambiguous.

A sufficient condition for an OG to be unambiguous is therefore the unambiguousness of the corresponding AOG.
4. Parsing a string using an AOG.

In order to parse a sentence $x$ we first enclose $x$ in symbols $\Phi^*$ and $\Phi$ (where $\Phi^*$ is assumed to be a new SNTS and $\Phi$ a new terminal symbol), yielding $\Phi^*x\Phi$. Formally, we add to the AOG the productions $\langle\text{Program}\rangle ::= \Phi^*A\Phi$ and $\Phi^* ::= \Phi$, where $A$ is the symbol which appeared only in a left part. We show that after each reduction the string has one of the following two forms

$$(4.1) \quad U_1^*U_2^* \ldots U_l^*U_1^*T_1T_2^* \ldots T_m$$

or

$$(4.2) \quad U_1^*U_2^* \ldots U_l^*U_1^*U_1T_1 \ldots T_m,$$

where the $U_i^*$ are SNTS's, the $T_i$ are terminals and $U_1$ is an UNTS.

Note that the original string $\Phi^*x\Phi$ has form (4.1). We assume also that no reduction $U ::= U_i^*, i < l$, can be made such that the resulting string is still a sentential form, since such reductions will have already been made. It will be seen later that the sufficient conditions for a unique canonical parse fulfill this requirement.

From the form of productions in an AOG (Figure 3), any reducible phrase containing $U_i, i < l$, must be $U_i$ itself, and this by assumption is not the case. Now at some point of the parse a reducible phrase must contain $T_1$, and from the form of productions in an AOG, $T_1$ must be the last character of any reducible phrase containing it. Therefore, at this step, the leftmost reducible phrase may not contain $T_2, T_3, \ldots, T_m$. We have therefore, using again the possible forms of productions in an AOG,
Lemma 7. A leftmost reducible phrase at step \( k \), assuming the string is of form (4.1), must be either

\[
U_l^*, T_1, \text{ or } U_l^*T_1.
\]

Assuming a string of form (4.2) the leftmost reducible phrase is

\[
U^*_l U_1, \text{ or } U^*_l T_1, \text{ or } U_1 T_1.
\]

In the case (4.1), if we know which of the strings \( U_l^* \), \( T_1 \) or \( U_l^*T_1 \) is the leftmost reducible phrase, we make a reduction \( U ::= U^*_a \), \( U_l ::= T_1 \) resp. \( U^* ::= U_l^*T_1 \), yielding again a string of the form (4.1) or (4.2).

In case (4.2) we first make a sequence of reductions

\[
U_2 ::= U_1, \ldots, U_j ::= U_{j-1} \text{ for some } j \geq 1,
\]

and then execute a final reduction \( U ::= U_l^*U_j, \text{ or } U^* ::= U_l^*U_jT_1, \text{ or } U^* ::= U_l^*T_1 \), depending on which of the three possibilities is the leftmost reducible phrase.

Note that at each step not only the leftmost reducible phrase, but also the sequence of reductions to be made, should be unique. If this is the case, then of course there exists a unique canonical parse. The next section gives sufficient conditions for the uniqueness of the canonical parse. The reason for calling the \( U^* \) "stack nonterminal symbols" is now clear. They are the only symbols which get pushed into the stack.

Note that a reduction \( U ::= U_l^* \) or \( U^* ::= T_1 \) at this time would result in a string which is not a sentential form, since the grammar is an AOG (two original nonterminals of the OG would eventually appear adjacent).
5. Sufficient conditions for a unique canonical parse.

We will use the following set $L(S)$ where $S$ is a symbol:

$$L(S) = \{s_1 | xS_1Sy \text{ is a sentential form for some } x, y \}.$$ 

$L(S)$ is just the set of symbols which are adjacent and to the left of $S_1$ in some sentential form. The construction of $L$ or related sets has been discussed elsewhere (see for instance Wirth and Weber [10]). We therefore do not wish to discuss at length the construction of $L(S)$.

We just state that

$$S_1 \in L(S) \text{ if and only if } (S, S' \neq S \lor S_1 < S \lor S, * > S),$$

where $=, <, \text{ and } * >$ are the precedence relations defined by Wirth and Weber (page 18, [10]).

Now consider case (4.1). We have a sentential form

$$U_1^*U_2^* \ldots U_l^* \ T_1 \ T_2 \ldots \ T_m.$$ 

If $U_l^*$ is a leftmost reducible phrase, then obviously

$$(4.3) \ \exists \ \text{a production } U ::= U_l^* \text{ such that } U \in L(T_1).$$

Similarly, if $U_l^*T_1$ or $T_1$ is a leftmost reducible phrase, we have respectively

$$(4.4) \ \exists \ \text{a production } U^* ::= U_l^*T_1;$$

$$(4.5) \ \exists \ \text{a production } U^* ::= T_1 \text{ such that } U_l^* \in L(U^*).$$

Consider case (4.2). We have a sentential form

$$U_1^* \ldots U_l^*U_1^*T \ldots \ T_m.$$ 

Depending on whether $U_l^*U_1^*, U_l^*T_1$, or $U_1^*T$ is a leftmost reducible
phrase, we have respectively

\[(4.6)\] a production \( U ::= U^* a_2 \) where \( U \in \mathcal{L}(T_1) \), and either
\[ U_2 = U_1 \text{ or } U_2 \Rightarrow U_1 ; \]

\[(4.7)\] a production \( U^*: = U^*_l U^*_2 T_1 \) where \( U_2 = U_1 \) or \( U_2 \Rightarrow U_1 ; \)

\[(4.8)\] a production \( U^*: = U^*_2 T_1 \) where \( U^*_l \in \mathcal{L}(U^*) \), and either
\[ U_2 = U_1 \text{ or } U_2 \Rightarrow U_1 . \]

We may now state the main

**Result.** Let \( U^*_l \) be a SNTS, \( U_1 \) a UNTS and \( T_1 \) a terminal symbol.

Assume that for any such \( U^*_l, U_1 \) and \( T_1 \)

(a) At most one of the conditions \((4.3),(4.4),(4.5)\) holds;

(b) At most one of the conditions \((4.6),(4.7),(4.8)\) holds;

(c) If one of the conditions \((4.3) \rightarrow (4.8)\) holds, the production described therein is unique.

Then there exists a unique canonical parse for each sentence \( x \) of the language.

The result follows from the fact that at each step the leftmost reducible phrase is unique (from (a) and (b)), and the corresponding reduction or set of reductions is unique (remember the restriction on an \( \text{OG} \) that if \( U_j \Rightarrow U_1 \) then the reductions \( U_j \ ::= U(j+1) \cdots , 1 \cdot 1\) are unique).

The algorithm which generates the "compiler is then straightforward. We first check that if \( U_1 \Rightarrow U_j \), the sequence of productions
\[ U_1 ::= U_2, \ldots ) U_{j-1} ::= U_j \] is unique. Next the \( \text{AOG} \) is constructed.
$L(s)$ is then determined for all terminal and SNTS $S$. Then for each $U_i^*$ and $T_1$ the productions are searched to see whether conditions (4.3), (4.4), or (4.5) hold, and if so, the production number (or just the left part) together with the reducible phrase is recorded. If for some $U_i^*$ and $T_1$ two different reductions are found to be possible, then some sentence may not be parsed unambiguously with the technique given in the next section. Note that this does not mean that the grammar is unambiguous; it just has not satisfied our sufficiency conditions. Triples $U_i^*, U_1$ and $T_1$ are handled similarly.

6. The transition matrix and stack.

We have seen that, with sufficient restrictions on the grammar, at each step in the parsing of a sentence according to the augmented operator grammar AOG, the partially reduced string has the form

(6.1) $U_1^* U_2 \ldots U_i^* T_1 \ldots T_m$

or

(6.2) $U_1^* U_2^* \ldots U_i^* U_1 T_1 \ldots T_m$

where $U_1$ is a non-terminal symbol of the original OG, $T_i$ are terminal symbols, and $U_j^*$ are SNTs of the AOG, or can be thought of as

(6.3) a representation for the head $xT$ (ending in a terminal symbol $T$) of the right part of a production $U ::= xTy$

of the original OG.

Furthermore, in the case (6.1) the leftmost reducible phrase, either
$U^*_l$, $T_1$ or $U^*_l T_1$ is uniquely determined by $U^*_l$ and $T_1$. In the case (6.2), the leftmost reducible phrase is uniquely determined by $U^*_l$, $T_1$ and $U_1$ and is either $U^*_l U_1$, $U_1 T_1$ or $U^*_l U_1 T_1$.

In order to parse a sentence as quickly as possible, we construct a transition matrix $B$ — a rectangular matrix whose elements $b_{ij}$ are numbers of subroutines. Each column $j$ represents a terminal symbol $T$ (or a class of terminal symbols — for instance, one column could represent the class <type> consisting of boolean, and integer). For each stack symbol $U^*_l$ we designate two rows of the matrix — a basic row and a secondary row. Their uses are as follows:

Suppose at a step of the parse, the string has the form (6.1). Then the basic row $b_{U^*_l} = i$ corresponding to $U^*_l$ together with the column $j$ representing $T_1$ determine an element $b_{ij}$ of the matrix — the number of a subroutine which, when executed, will effect the unique reduction to be made.

If the string has form (6.2), the secondary row corresponding to $U^*_l$ together with the column representing $T_1$ determine an element of the matrix. When the corresponding subroutine is executed, $U_1$ will be checked and the appropriate unique reductions will be made.

For practical purposes we assume that the secondary row always follows the basic row. That is, if the basic row for $U^*_l$ is row number $b_{U^*_l}$, then the secondary row is number $b_{U^*_l} + 1$.

The pushdown (last-in-first-out) stack $ST$ consists of elements, each consisting of two parts. If $p$ is the pointer to the current top stack element, the two parts of the top stack element are labeled $ST1_p$ and $ST2_p$. The contents of each stack element are best illustrated by
a diagram. If the string has the form (6.1) or (6.2), the stack configuration is as illustrated in Figure 5a or Figure 5b respectively, where again $b_1^*$ is the index or row number of the basic row corresponding to $U_1^*$.

\[
\begin{array}{ll}
(ST1, ST2) & (ST1, ST2) \\
(b^*, 0) & (b^* + 1, 0) \\
(b_{l-1}^* + 1, 0) & (b_{l-1}^* + 1, 0) \\
(b^* + 1, 0) & (b^* + 1, 0)
\end{array}
\]

(a) (b)

Figure 5

Note that the first part $ST1$ of a stack element which is not the top stack element is always the index of a secondary row, since when it later becomes the top stack element, $ST2_p$ must contain a nonterminal symbol $U$. Later it will be shown how the second part of each element may be used systematically to hold semantic information.

To illustrate how efficient the technique is, we give an example of a typical implementation on the IBM 7090. Suppose that the stack element $ST1_p$ actually contains the instruction

\[
\text{TRA}^*\{\text{ADDRESS OF B}_{i^*}, 0\}, 4 \quad \text{or} \quad \text{TRA}^*\{\text{ADDRESS OF B}_{i^* + 1}, 0\}, 4
\]

that the matrix $B$ is stored rowwise in memory, that each matrix element consists of a single location containing the address of the corresponding subroutine, that there exists a vector COLUMN to map a terminal symbol into the corresponding column number, and that the stack pointer $p$ is
in index register 1. The following sequence then determines the subroutine to be executed:

- **LXA Tl,2** PUT THE INCOMING SYMBOL IN INDEX REGISTER 2.
- **CLA COLUMN,2** COLUMN NUMBER FOR Tl IN ACCUMULATOR.
- **PAC 0,4** COMPLEMENT OF COLUMN NUMBER IN XR4.
- **TRA STl,1** JUMP TO TOP STACK ELEMENT, WHICH IN TURN WILL JUMP TO THE SUBROUTINE

As an example of a matrix and subroutines, consider first the grammar in Figure 6, which is the same as the grammar in Figure 2 (Section 3) except for the introduction of the production

<PROGRAM> ::= φ <PROG> φ.

Both the matrix in Figure 7 and the associated subroutines of Figure 8 were produced exactly as they appear from the grammar of Figure 6 by the algorithm programmed in Extended ALGOL [11] on the B5500 (except for the numbering of the rows of the matrix).

The SNTSs of the AOG do not appear in the matrix or subroutines; we have labeled the basic rows of the matrix with the head of the right part of the production which they represent. Correspondingly, for example, subroutine 400 of Figure 8 contains the instruction

STl_p ← ROW( <EXPRESSION> OR ), which means that STl_p is to have as its value the index of the basic row corresponding to the SNTS which represents <EXPRESSION> OR . This is row 13. As can be seen, the AOG is actually not necessary practically, but is a convenient theoretical tool.
The zero elements of the matrix represent incorrect pairs, while the other matrix elements are numbers of the subroutines listed in Figure 8.

The individual statements of the subroutines are separated by a slash "/". The statement "NEW T1" means "SCAN", or use the symbol T2 as the next incoming symbol T1. If the statement "NEW T1" is not executed, the old T1 will be used again on the next cycle. After a subroutine has been executed, the next cycle is performed.

Some of the subroutines test ST2_p for the presence of a nonterminal symbol. If ST2_p is one of the nonterminals listed, a corresponding subroutine is executed. If not, a syntactic error has occurred — the original string is not a sentence of the grammar.

Note also that if a reduction to some non-terminal U is made (ST2_p \rightarrow U), the original production of the operator grammar corresponding to this reduction is also listed for reference.

The following simplification has been made. Suppose that xT is the right part of only one production U ::= xT of the OG, and that there is no production U_1 ::= xTy for some nonempty y. The AOG will contain among others two productions - U^* ::= x_1T and U ::= U^*. Obviously this is not necessary. In order to save the intermediate step, the two productions are replaced by the single production U ::= x_1T.
### Productions

1. `<PROGRAM>` ::= `<STATEMENT>` `<PROGRAM>`
2. `<PROGRAM>` ::= `<STATEMENT>`
3. `<STATEMENT>` ::= `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>`
4. `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>` ::= `<STATEMENT>`
5. `<STATEMENT>` ::= `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>`
6. `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>` ::= `<STATEMENT>`
7. `<STATEMENT>` ::= `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>`

### Nonterminal Symbols

1. `<PROGRAM>`
2. `<STATEMENT>`
3. `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>`
4. `<IF EXPRESSION THEN STATEMENT ELSE STATEMENT>`

### Terminal Symbols

5. `<PHI>`
6. `<IF>`
7. `<THEN>`
8. `<ELSE>`
9. `<VARIABLE>`
10. `<OR>`

### Matrix

**Figure 6**

<table>
<thead>
<tr>
<th>PHI</th>
<th>I</th>
<th>T</th>
<th>E</th>
<th>VAR</th>
<th>V</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>F</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PHI</th>
<th>SECONDARY ROW</th>
<th>I</th>
<th>T</th>
<th>E</th>
<th>VAR</th>
<th>V</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 1 0 0 0</td>
<td>I</td>
<td>F</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 2 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 4 0 0 0 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 7 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7**

<table>
<thead>
<tr>
<th>PHI</th>
<th>SECONDARY ROW</th>
<th>I</th>
<th>T</th>
<th>E</th>
<th>VAR</th>
<th>V</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0 1 0 0 0</td>
<td>I</td>
<td>F</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>0 0 10 10 0 9 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 2 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 10 0 5 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 12 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MATRIX SUBROUTINES, THOSE SUBS WHICH ARE ACTUALLY MATRIX ENTRIES HAVE NUMBERS LESS THAN 400

1. \( ST1P+ST1P+1 / P*P+1 / ST1P*ROW(T1) / NEW T1 \)
2. \( ST1P*ST1P+1 / ST2P*<EXPRESSION> FOR PRODUCTION *<EXPRESSION>**T1= VARIABLE / NEW T1 \)
3. \( USE SUB 401 IF ST2P IS <STATEMENT> \)
4. \( USE SUB 401 IF ST2P IS <EXPRESSION> \)
5. \( USE SUB 401 IF ST2P IS <EXPRESSION> \)
6. \( USE SUB 401 IF ST2P IS <STATEMENT> \)
7. \( USE SUB 401 IF ST2P IS <STATEMENT> \)
8. \( USE SUB 401 IF ST2P IS <STATEMENT> \)
9. \( ST1P*ROW( VARIABLE T1= ) / NEW T1 \)
10. \( P*P+1 / ST2P*<EXPRESSION> FOR PRODUCTION <EXPRESSION> **T1= VARIABLE / NEW T1 \)
11. \( USE SUB 406 IF ST2P IS <EXPRESSION> \)
12. \( P*P+1 / ST2P*<EXPRESSION> FOR PRODUCTION <EXPRESSION> **T1= <EXPRESSION> OR VARIABLE / NEW T1 \)

100. \( P*P+1 / ST1P*ROW( <EXPRESSION>OR ) / NEW T1 \)
101. \( P*P+1 / ST2P*<PROGRAM> FOR PRODUCTION <PROGRAM> II= PHI <PROG> PHI / NEW T1 \)
102. \( ST1P*ROW( IF <EXPRESSION>THEN ) / NEW T1 \)
103. \( P*P+1 / ST2P*<PROG> FOR PRODUCTION <PROG> II= IF <EXPRESSION>THEN <STATEMENT> \)
104. \( ST1P*ROW( IF <EXPRESSION>THEN <STATEMENT> ELSE ) / NEW T1 \)
105. \( P*P+1 / ST2P*<STATEMENT> FOR PRODUCTION <STATEMENT> II= IF <EXPRESSION>THEN <STATEMENT> \)
106. \( ELSE <STATEMENT> ELSE <STATEMENT> \)
107. \( P*P+1 / ST2P*<STATEMENT> FOR PRODUCTION <STATEMENT> II= VARIABLE <EXPRESSION> \)

Figure 8
7. An example of a parse.

Let us parse the sentence

\[ \text{IF VARIABLE THEN VARIABLE := VARIABLE} \]

of the \( \mathcal{G} \) in Figure 6. We start with the following configuration:

<table>
<thead>
<tr>
<th>Cycle P</th>
<th>STACK</th>
<th>( T_1 )</th>
<th>Rest of string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 ([row] 1 [PHI], 0)</td>
<td>IF VARIABLE THEN VARIABLE := VARIABLE PHI</td>
<td></td>
</tr>
</tbody>
</table>

The row labeled PHI in Figure 7 and \( T_1 = \text{"IF"} \) determine subroutine 1 of Figure 8. Execution of the first statement "\( \text{ST}_1 \leftarrow \text{ST}_1 + 1 \)" of subroutine 1 changes \( \text{ST}_1 \) to [row]2. The stack pointer is then increased by 1 and the index of the row corresponding to "IF" (since \( T_1 = \text{"IF"} \)), which is 3, is put in \( \text{ST}_2 \). "VARIABLE" is then scanned, yielding

<table>
<thead>
<tr>
<th>Cycle P</th>
<th>STACK</th>
<th>( T_1 )</th>
<th>Rest of string</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 ([row]3, 0)</td>
<td>VARIABLE THEN VARIABLE := VARIABLE PHI</td>
<td></td>
</tr>
</tbody>
</table>

([row]2, 0)

Row 3 and "VARIABLE" now determine subroutine number 2. Here we change \( \text{ST}_2 \) to [row]4, put "<EXPRESSION>" in \( \text{ST}_2 \), and indicate that the next symbol, "THEN", is to be scanned. This yields

<table>
<thead>
<tr>
<th>Cycle P</th>
<th>STACK</th>
<th>( T_1 )</th>
<th>Rest of string</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2 (4, &lt;EXPRESSION&gt;)</td>
<td>THEN VARIABLE := VARIABLE PHI</td>
<td>(2, 0)</td>
</tr>
</tbody>
</table>

25
Row 4 and "THEN" lead to subroutine number 4. There, ST2 is checked for "<EXPRESSION>". Since it is correct, subroutine 402 is executed yielding

<table>
<thead>
<tr>
<th>Cycle</th>
<th>P STACK</th>
<th>T1 Rest of string subroutine to execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2 (5, 0) VARIABLE := VARIABLE PHI 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2, 0)</td>
<td></td>
</tr>
</tbody>
</table>

Continuing in this manner gives the following configurations at the beginning of each cycle.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>P STACK</th>
<th>T1 Rest of string subroutine to execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3 (9, 0) := VARIABLE PHI 9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(6, 0) (2, 0)</td>
<td></td>
</tr>
</tbody>
</table>

| 6     | 3 (11, 0) VARIABLE PHI 2 |
|       | (6, 0) (2, 0)                                    |

| 7     | 3 (12, <EXPRESSION>) PHI 11 |
|       | (6, 0) (2, 0)                          |

| 8     | 2 (6, <STATEMENT>) PHI 6 |
|       | (2, 0) (2, 0)                        |

| 9     | 1 (2, <PROG>) PHI 3(STOP) |
|       | (2, 0) (2, 0)                        |
8. **Representation of non-terminals in the stack.**

Strictly speaking, one should insert the nonterminal symbol $U$ itself into $ST_{2p}$. This is however neither practical nor necessary. In practice, nonterminals fall into classes whose elements are the same semantically. For instance, in ALGOL the nonterminals `<primary>`, `<factor>`, `<term>`, `<simple arith expr>` are introduced only to help define the precedence of operations. In a compiler, they would all be represented by an address specifying a location which gives the type, location of the value during execution time (accumulate, register, storage location), etc. The determination of which $U$ is actually in $ST_{2p}$ turns out to be almost always a **semantic evaluation**, which would have to be done anyway. There is therefore very rarely any list searching to determine which $U$ is in $ST_{2p}$, but just a semantic evaluation of $ST_{2p}$. Accordingly, a reduction $U ::= x$ is accomplished by inserting into $ST_{2p}$ the **semantic meaning** of the symbol $U$ and not $U$ itself. Notice that we assume in the discussion of the method that productions $U_i ::= U_j$ have no "interpretation rule" associated with them, which is usually the case.

Note also that the part $ST_{2p}$ of the elements $p = 1, \ldots, n-1$ may also be used systematically to store semantic information. If we formally parse the ALGOL statement `BEGIN A := (B+E)+C*D END` there will be in the stack at some time the elements

\[
\begin{align*}
(b <\text{term*}> +1, \text{identifier}) \\
(b <\text{expr}+> +1, 0) \\
(b <\text{var}:= > +1, 0)
\end{align*}
\]
We can, though, use the second part of each stack element to contain semantic information:

\[(\text{\textless} \text{term}\textgreater + 1, \text{\text{semantics of D}})\]
\[(\text{\textless} \text{expr \triangleright} + 1, \text{\text{semantics of C}})\]
\[(\text{\textless} \text{VAR := \triangleright} + 1, \text{\text{semantics of B+E}})\]
\[(\text{\textless} \text{BEGIN} + 1, \text{\text{semantics of A}})\]
\[(\text{\textless} \text{\textgreater} + 1, \text{\text{any necessary information}})\]

9. **Other uses of transition matrices.**

Two other uses will be introduced here, both concerned with optimizing the calculation of addresses of subscripted variables within FOR-loops ([6],[9],[5]).

Provided that a FOR-loop meets certain conditions, calculation of the address of a subscripted variable \(A[E_1, \ldots, E_n]\) occurring in the statement of the FOR-loop may be optimized if the \(E_i\)s satisfy certain restrictions, some of which we list here:

1. \(E_i\) is **linear** in the loop variable of the FOR-loop, \(i = 1, \ldots, n\).
   That is, \(E_i\) may be put in the form \(C_1 \times I + C_2\), where \(C_1\) and \(C_2\) are expressions not containing the loop variable \(I\).

2. \(E_i\) contains only simple integer variables, integer constants, parentheses \((\text{ and })\), and the operators \(+, -, \text{ and } \ast\).
3. The variables appearing in the $E_i$ do not change within the
FOR-loop statement.

Restrictions 1 and 2 may be checked systematically using the
(operator) grammar in Figure 9. If $A[E_1, \ldots, E_n]$ is the subscripted
variable and $<$CONST $E_i$ $\Rightarrow [E_1, \ldots, E_n]$, then $A[E_1, \ldots, E_n]$ satisfies
restrictions 1 and 2 and moreover no $E_i$ contains the loop variable. If
$L<$LIN $E_i$ $\Rightarrow [E_1, \ldots, E_n]$, then similarly $A[E_1, \ldots, E_n]$ satisfies
restrictions 1 and 2 but at least one $E_i$ contains the loop variable.

From the grammar we generate the optimizable subscript checker —
the transition matrix and subroutines in Figure 10. Note that column 1
of the matrix contains only zeroes. We may map all terminal symbols
except for the ones listed in restriction 2 into column 1. If, when
parsing a subscripted variable according to the grammar in Figure 9, an
error occurs, then this subscripted variable is handled in the usual way.
Otherwise, it may be possible to optimize here and the variables occurring
in the $E_i$ should be stored in some list for further checking.

As a second example we look at the FOR-loop itself. We want it
to have the form

$$\text{FOR } I \leftarrow E_1 \text{ STEP } E_2 \text{ UNTIL } E_3 \text{ DO } S;$$

where the variables in $E_2$ do not change within the statement $S$, $E_2$
does not contain the loop variable $I$, and the $E_i$ are integer expres-
sions. The further restriction is again made, that the $E_i$ consist only
of integer simple variables, integer constants, $\,(,)\,,$ $+$, $-$, and $\ast\,.$
Note that $E_1$ and $E_3$ may contain the loop variable $I$, but $E_2$ may
not. The variables in $E_2$ should be listed for further checking.
PRODUCTIONS

1. \textbf{<CONST ELEM>} ::= [ \textbf{<CONST SUBS>}]
2. \textbf{<LIN ELEM>} ::= [ \textbf{<LIN SUBS>}
3. \textbf{<CONST SUBS>} ::= \textbf{<CONST EXPR>}
4. \textbf{<LIN SUBS>} ::= \textbf{<LIN EXPRESSION>}
5. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST SUBS>}, \textbf{<CONST EXPRESSION>}
6. \textbf{<CONST EXPRESSION>} ::= \textbf{<LIN SUBS>}, \textbf{<CONST EXPRESSION>}
7. \textbf{<CONST EXPRESSION>} ::= \textbf{<LIN SUBS>}, \textbf{<LIN EXPRESSION>}
8. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST SUBS>}, \textbf{<CONST EXPRESSION>}
9. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
10. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<LIN EXPRESSION>}
11. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
12. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
13. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
14. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
15. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
16. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
17. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
18. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
19. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}
20. \textbf{<CONST EXPRESSION>} ::= \textbf{<CONST EXPRESSION>}, \textbf{<CONST SUBS>}, \textbf{<LIN EXPRESSION>}

NONTERMINAL SYMBOLS

1. \textbf{<CONST ELEM>}
2. \textbf{<LIN ELEM>}
3. \textbf{<CONST SUBS>}
4. \textbf{<LIN SUBS>}
5. \textbf{<CONST EXPRESSION>}
6. \textbf{<CONST TERM>}
7. \textbf{<CONST FACT>}
8. \textbf{<LIN EXPRESSION>}
9. \textbf{<LIN TERM>}
10. \textbf{<LIN FACT>}

TERMINAL SYMBOLS

11. [ \textbf{\textgreater} ]
12. [ \textbf{\textless} ]
13. [ \textbf{\textgreater} ]
14. [ \textbf{\textless} ]
15. \textbf{+}
16. \textbf{-}
17. \textbf{INTEGER}
18. \textbf{LOOP}
19. \textbf{VAR}
20. \textbf{INTEGER}

MATRIX IS 18 × 10

Figure 9
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;CONST SUBS&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;LIN SUBS&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;CONSTEXPR&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>8</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>++ OR --&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>8</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;CONSTTERM&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;LINEXPR&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>&lt;LINTERM&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>SECONDARY ROW</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*Figure 10*
MATRIX SUBROUTINES, THOSE SUBS WHICH ARE ACTUALLY MATRIX ENTRIES HAVE NUMBERS LESS THAN 000

1. \[ SPI+SPI+1 \times SPI+1 \times SPI+RDN(T1) \times NEW \ T1 \]
2. \[ SPI+SPI+1 \times SPI+SPI+1 \times SPI+CONST \quad FACT \quad FOR \ PRODUCTION \quad <CONST \quad FACT \times tI = INTEGER \quad \times NEW \ T1 \]
3. \[ SPI+SPI+1 \times SPI+SPI+1 \times SPI+CONST \quad FACT, \quad FOR \ PRODUCTION <CONST \quad FACT \times tI = INTEGER \quad VAR \quad \times NEW \ T1 \]
4. \[ SPI+SPI+1 \times SPI+SPI+1 \times SPI+CONST \quad FACT \quad FOR \ PRODUCTION <LIN \quad FACT \times tI = LOOP \quad VAR \quad \times NEW \ T1 \]

5. EITHER
   - USE SUB 406 IF STOP IS <CONST SUBS><CONST EXPR><CONST TERM><CONST FACT>
   - USE SUB 407 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

6. EITHER
   - USE SUB 408 IF STOP IS <CONST SUBS><CONST EXPR><CONST TERM><CONST FACT>
   - USE SUB 409 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

7. EITHER
   - USE SUB 410 IF STOP IS <CONST EXPR><CONST TERM><CONST FACT>
   - USE SUB 411 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

8. EITHER
   - USE SUB 412 IF STOP IS <CONST EXPR><CONST TERM><CONST FACT>
   - USE SUB 413 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

9. EITHER
   - USE SUB 414 IF STOP IS <CONST EXPR><CONST TERM><CONST FACT>
   - USE SUB 415 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

10. EITHER
    - USE SUB 416 IF STOP IS <CONST EXPR><CONST TERM><CONST FACT>
    - USE SUB 417 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

11. EITHER
    - USE SUB 418 IF STOP IS <CONST EXPR><CONST TERM><CONST FACT>
    - USE SUB 419 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

12. EITHER
    - USE SUB 420 IF STOP IS <CONST EXPR><CONST TERM><CONST FACT>
    - USE SUB 421 IF STOP IS <LINSUBS><LINEXPR><LINTERM><LINFACT>

Figure 10 (continued)
16 USE SUB 422 IF ST2P IS <CONST FACT>

400 P*P+1 / ST1P*ROW( <CONST SUBS> ) / NEW Ti
401 P*P+1 / ST1P*ROW( <CONST EXPR><+ OR > ) / NEW Ti
402 P*P+1 / ST1P*ROW( <CONST TERM> ) / NEW Ti
403 P*P+1 / ST1P*ROW( <LIN SUBS> ) / NEW Ti
404 P*P+1 / ST1P*ROW( <LIN EXPR> <+ OR > ) / NEW Ti
405 P*P+1 / ST1P*ROW( <LIN TERM> ) / NEW Ti
406 P*P+1 / ST2P<CONST ELEM> FOR PRODUCTION <CONSTELEM> := [ <CONST SUBS> ] / NEW Ti
407 P*P+1 / ST2P<LIN ELEM> FOR PRODUCTION <LINELEM> := [ <LIN SUBS> ] / NEW Ti
408 P*P+1 / ST2P<CONST SUBS, FOR PRODUCTION <CONSTSUBS> := <CONSTSUBS> <CONST EXPR>
409 P*P+1 / ST2P<LIN SUBS> FOR PRODUCTION <LINSUBS> := <LINSUBS> <LIN EXPR>
410 P*P+1 / ST2P<LIN SUBS> FOR PRODUCTION <LINSUBS> := <LINSUBS> <LIN SUBS> <CONST EXPR>
411 P*P+1 / ST2P<CONST EXPR> FOR PRODUCTION <CONSTEXPR> := <CONSTEXPR><+ OR > <CONST TERM>
412 P*P+1 / ST2P<LIN EXPR> FOR PRODUCTION <LINEXPR> := <LINEXPR><+ OR > <LIN TERM>
413 P*P+1 / ST2P<CONST EXPR> FOR PRODUCTION <CONSTEXPR> := <CONSTEXPR><+ OR > <CONST TERM>
414 P*P+1 / ST2P<LIN EXPR> FOR PRODUCTION <LINEXPR> := <LINEXPR><+ OR > <LIN TERM>
415 P*P+1 / ST2P<CONST TERM> FOR PRODUCTION <CONSTTERM> := <CONST TERM> <CONST FACT>
416 P*P+1 / ST2P<LIN TERM> FOR PRODUCTION <LINTERM> := <LINTERM> <CONST TERM>
417 P*P+1 / ST2P<CONST FACT> FOR PRODUCTION <CONST FACT> := ( <CONST EXPR> ) / NEW Ti
418 P*P+1 / ST2P<LIN FACT> FOR PRODUCTION <LIN FACT> := ( <LIN EXPR> ) / NEW Ti
419 P*P+1 / ST2P<LIN EXPR> FOR PRODUCTION <LIN EXPR> := <LIN EXPR><+ OR > <LIN TERM>
420 P*P+1 / ST2P<LIN EXPR> FOR PRODUCTION <LIN EXPR> := <LIN EXPR><+ OR > <CONST TERM>
421 P*P+1 / ST2P<LIN TERM> FOR PRODUCTION <LIN TERM> := <LIN TERM> <CONST FACT>
The grammar of Figure 11 is then constructed. The terminal "INT CONS, VAR" represents the class of integer constants and simple integer variables (except the loop variable of this loop). The corresponding transition matrix and subroutines are in Figure 12. We call this the loop checker. Since we are not interested in the precedence of the operators +, -, *, we have simplified the productions for arithmetic expressions. Note also in Figure 11 that "*" is used as a unary operator. Since we assume that the subscripts have been or are being checked for syntactical errors by another part of the compiler, this will never happen. Grammars should always be constructed according to what they will be used for, and should be as simple as possible.

One can incorporate the loop checker and optimizable subscript checker into an existing syntax checker as follows. All three are put in memory together. The main syntax checker executes as it normally does, performing the usual "cycles" described already.

When a FOR is scanned, the syntax checker activates the loop checker. Thereafter both process in parallel. The syntax checker processes one symbol and then passes it on to the loop checker, which when finished returns to the syntax checker to process the next symbol:
PRODUCTIONS

1 <FOR LOOP> ::= FOR LOOP VAR + <EXP2> STEP <EXP1> UNTIL <EXP2>

2 <EXP1> ::= <EXP1> <+,-,x> <EXP1 FACT>

3 <EXP1> ::= <EXP1> <+,-,x> <EXP1 FACT>

4 <EXP1> ::= <EXP1 FACT> ( <EXP1> )

5 <EXP1> ::= <EXP1> <INT CONSVAR>

6 <EXP2> ::= <EXP1> <+,-,x> <EXP2 FACT>

7 <EXP2> ::= <EXP2> <+,-,x> <EXP2 FACT>

8 <EXP2> ::= <EXP2> <+,-,x> <EXP2 FACT>

9 <EXP2> ::= 00 <EXP2 FACT>

10 <EXP2> ::= <EXP2 FACT>

11 <EXP2> ::= <EXP2 FACT>

12 <EXP2> ::= <EXP2 FACT>

13 <EXP2> ::= LOOP VAR

NON TERMINAL SYMBOLS

1 <FOR LOOP> 2 <EXP1> 3 <EXP1 FACT> 4 <EXP2>

5 <EXP2 FACT>

TERMINAL SYMBOLS

6 FOR 7 LOOP VAR 8 * 9 STEP

10 UNTIL 11 DO 12 <+,-,x> 13 ( 

14 ) 15 INT CONSVAR

MATRIX IS 18 x 10

Figure 11
MATRIX SUBROUTINES, THOSE SUBS WHICH ARE ACTUALLY MATRIX ENTRIES HAVE NUMBERS LESS THAN 400

1. \( \text{ST1P+ST1P} \div \text{P+P+1} \div \text{ST1P+ROW(T1)} \div \text{NEW T1} \)
2. \( \text{ST1P+ST1P} \div \text{ST2P+EXP1 FACT} \div \text{FOR PRODUCTION<EXP1 FACT>11= INT CONS+VAR} \div \text{NEW T1} \)
3. \( \text{ST1P+ST1P} \div \text{ST2P+EXP2 FACT} \div \text{FOR PRODUCTION<EXP2 FACT>11= LOOP VAR} \div \text{NEW T1} \)
4. \( \text{ST1P+ROW} \div \text{(FOR LOOP VAR)} \div \text{NEW T1} \)
5. \( \text{ST1P+ROW} \div \text{(FOR LOOP VAR + ) \div NEW T1} \)
6. \( \text{USE SUB 402 IF ST2P IS<EXP2 FACT>} \)
   EITHER
   \( \text{USE SUB 401 IF STEP IS<EXP1 FACT>} \)
   \( \text{USE SUB 401 IF ST2P IS<EXP2 FACT>} \)
7. \( \text{USE SUB 431 IF ST2P IS<EXP1 FACT>} \)
8. \( \text{USE SUB 400 IF ST2P IS<EXP1 FACT>} \)
9. \( \text{USE SUB 434 IF ST2P IS<EXP2 FACT>} \)
10. \( \text{USE SUB 436 IF ST2P IS<EXP2 FACT>} \)
11. \( \text{USE SUB 435 IF ST2P IS<EXP1 FACT>} \)
12. \( \text{USE SUB 435 IF ST2P IS<EXP1 FACT>} \)
13. \( \text{USE SUB 435 IF ST2P IS<EXP1 FACT>} \)
   \( \text{USE SUB 436 IF ST2P IS<EXP2 FACT>} \)
14. \( \text{USE SUB 408 IF ST2P IS<EXP2 FACT>} \)
15. \( \text{USE SUB 407 IF ST2P IS<EXP1 FACT>} \)
16. \( \text{USE SUB 407 IF ST2P IS<EXP1 FACT>} \)
   \( \text{USE SUB 408 IF ST2P IS<EXP2 FACT>} \)
17. \( \text{USE SUB 406 IF ST2P IS<EXP1 FACT>} \)
   \( \text{USE SUB 410 IF ST2P IS<EXP2 FACT>} \)
18. \( \text{USE SUB 411 IF ST2P IS<EXP1 FACT>} \)
   \( \text{USE SUB 412 IF ST2P IS<EXP2 FACT>} \)

400. \( \text{P+P+1} \div \text{ST1P+ROW<EXP1><+++><>} \div \text{NEW T1} \)
401. \( \text{P+P+1} \div \text{ST1P+ROW<EXP2><+++><>} \div \text{NEW T1} \)
402. \( \text{ST1P+ROW} \div \text{(FOR LOOP VAR +<EXP2>STEP)} \div \text{NEW T1} \)

Figure 12 (continued)
Figure 12 (continued)
The loop checker disconnects itself as soon as it determines that the loop is not of the right form, or when it is finished. Similarly, the optimizable subscript checker is connected when the $[\ldots]$ of the subscript variable $A[E_1, \ldots, E_n]$ is first scanned. The optimizable variable checker disconnects itself when finished or when it is determined that this subscripted variable does not satisfy one of the restrictions.

The loop checker and optimizable subscript checker are not concerned with errors and error recovery. If any error occurs, they simply disconnect themselves, or will be disconnected by the syntax checker itself.

10. Summary.

The ideas in this paper have been used intuitively in the ALCOR-ILLINOIS 7090 Compiler ([5],[1]). The second pass actually contains the three matrices illustrated in Section 9. The matrix technique has its most important use, in my opinion, in a student system, where a very fast compiler resides in core and must also produce excellent error messages. Because the syntax checker matrix for ALGOL is so large (on
the 7090 (100 x 45) and because over sixty percent of the array elements represent illegal symbol pairs, a much wider variety of error messages is efficiently possible. An algorithm is being developed for producing, from the grammar, an error recovery subroutine for each "error" element of the matrix. Another advantage of the matrix technique is the simplicity of the overall design.

The only disadvantage is the space used. A partial solution to this problem might be to parse those constructions of the grammar which are most used (for instance, expressions) using the matrix technique and to use some other slower but less space-consuming technique for the rest of the grammar. Note also that the size of the matrix may be cut in half by allowing only one row for each stack nonterminal symbol. Each subroutine must then check whether a nonterminal exists in \( ST_2 \) or not ( \( ST_2 \) nonzero or not).
REFERENCES


Appendix A.

Below is the grammar of an ALGOL-like language defined by Floyd ([4]) in his article on operator precedence. Note that in the subroutines the phrase "ambiguity in sub" appears 6 times. This means that in this subroutine the reduction to make is not uniquely determined from $U^*_1$, $U_1$ and $T_1$, and not that the grammar is ambiguous. The difficulties can be circumvented by either changing the grammar or using semantic information to determine which reduction to be made.

The matrix and subroutines were constructed using the grammar as input.
<table>
<thead>
<tr>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATEMENT</strong></td>
<td><strong>ST LIST</strong></td>
<td><strong>PROGRAM</strong></td>
</tr>
<tr>
<td>TERMINAL SYMBOLS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>LETTER</td>
<td>41</td>
</tr>
<tr>
<td>44</td>
<td>[</td>
<td>45</td>
</tr>
<tr>
<td>48</td>
<td>**</td>
<td>49</td>
</tr>
<tr>
<td>52</td>
<td>IF</td>
<td>53</td>
</tr>
<tr>
<td>56</td>
<td><strong>&lt; T O R F &gt;</strong></td>
<td>57</td>
</tr>
<tr>
<td>60</td>
<td>IMPLIES</td>
<td>61</td>
</tr>
<tr>
<td>64</td>
<td>VALUE</td>
<td>65</td>
</tr>
<tr>
<td>68</td>
<td>**</td>
<td>69</td>
</tr>
<tr>
<td>71</td>
<td>FUNCTION</td>
<td>72</td>
</tr>
<tr>
<td>80</td>
<td>GO TO</td>
<td>81</td>
</tr>
</tbody>
</table>

**MATRIX IS 116 X 42**
<table>
<thead>
<tr>
<th>24</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /> <img src="image2.png" alt="Image" /> <img src="image3.png" alt="Image" /> <img src="image4.png" alt="Image" /> <img src="image5.png" alt="Image" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**<DISJ>**

**OR**

**SECONDARY ROW**

**<IMPL>**

**IMPLIES**

**SECONDARY ROW**

**<BOOL EXP>**

**EQUIV**

**SECONDARY ROW**

**<EXP LIST>**

**SECONDARY ROW**

**<ARITH EXP>**

**I**

**SECONDARY ROW**

**<LISP LIST>**

**SECONDARY ROW**

**<TYPE>**

**VALUE**

**SECONDARY ROW**

**<TYPE>**

**ARRAY**

**SECONDARY ROW**

**<CONST LIST>**

**SECONDARY ROW**

**<NAME PART>**

**J**

**SECONDARY ROW**

**<PROCEDURE>**

**<NAME PART>**

**K**

**SECONDARY ROW**

**<PROCEDURE>**

**<NAME PART>**

**L**

**SECONDARY ROW**

**<FUNCTION>**

**SECONDARY ROW**

**<FUNCTION>**

**<NAME PART>**

**M**

**SECONDARY ROW**

**<DEC LIST>**

**SECONDARY ROW**

**<ARITH EXP>**

**STEP**

**SECONDARY ROW**

**<ARITH EXP>**

**STEP**

**<ARITH EXP>**

**UNTIL**
MATRIX SUBROUTINES, THOSE SUBS WHICH ARE ACTUALLY MATRIX ENTRIES HAVE NUMBERS LESS THAN 400

1  \text{ST1P+ST1P+1 / P+P+1 / ST1P+ROW(T1) / NEW T1}

2  \text{ST1P+ST1P+1 / ST2P+ID \quad \text{FOR PRODUCTION 10 \quad \text{LETTER / NEW T1}}}

3  \text{ST1P+ST1P+1 / ST2P+BOOL PRIM \quad \text{FOR PRODUCTION BOOL PRIM 11 \quad \text{<T OR F> / NEW T1}}}

4  \text{ST1P+ST1P+1 / ST2P+CL STATE \quad \text{FOR PRODUCTION CL STATE 11 \quad \text{COMMENT / NEW T1}}}

5  \text{USE SUB 424 IF ST2P IS ID}

6  \text{P+P+1 \quad ST2P+<LITCON> \quad \text{FOR PRODUCTION <LITCON> 11 \quad \text{DIGITSTR / NEW T1}}}

7  \text{ST1P+ROW \quad \text{DIGITSTR / NEW T1}}

8  \text{P+P+1 \quad ST2P+<LITCON> \quad \text{FOR PRODUCTION <LITCON> 11 \quad \text{DIGITSTR / NEW T1}}}

9  \text{P+P+1 \quad ST2P+<LITCON> \quad \text{FOR PRODUCTION <LITCON> 11 \quad \text{DIGITSTR}}}

10 \text{P+P+1 \quad ST2P+<LITCON> \quad \text{FOR PRODUCTION <LITCON> 11 \quad \text{DIGITSTR / NEW T1}}}

11 \text{USE SUB 426 IF ST2P IS <AEXP LIST> \quad \text{<ARITH EXP> <SIMP AEXP> <TERM> <FACTOR> <PRIMAR Y> <FUNC DES>}}

12 \text{USE SUB 421 IF ST2P IS ID}

13 \text{USE SUB 425 IF ST2P IS <AEXP LIST> \quad \text{<ARITH EXP> <SIMP AEXP> <TERM> <FACTOR> <PRIMAR Y> <FUNC DES>}}

14 \text{USE SUB 421 IF ST2P IS ID}

15 \text{USE SUB 433 IF ST2P IS <PRIMAR Y> \quad \text{<FUNC DES> <VARIABLE> ID <SUB VAR> <LITCON>}}

16 \text{USE SUB 435 IF ST2P IS <SIMP AEXP> \quad \text{<TERM> <SUB VAR> <LITCON> <FACTOR> <PRIMAR Y> <FUNC DES> <VARIABLE> ID}}

17 \text{USE SUB 424 IF ST2P IS <TERM> <FACTOR> <PRIMAR Y> <FUNC DES> <VARIABLE> ID <SUB VAR>}

18 \text{EITHER}

19 \text{EITHER}

AMBIGUITY IN SUB 18

19 \text{EITHER}

USE SUB 427 IF ST2P IS <IDLIST>
USE SUB 426 IF ST2P IS <EXP LIST>
USE SUB 428 IF ST2P IS <EXP LIST>

USE SUB 427 IF ST2P IS <IDLIST>
USE SUB 426 IF ST2P IS <EXP LIST>
USE SUB 428 IF ST2P IS <EXP LIST>
AMBIGUITY IN SUB 19

20 EITHER
USE SUB 437 IF ST2P IS <ARITH EXP> <SIMP A EXP> <TERM> <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON> <BOOL EXP> <IMPL>
USE SUB 406 IF ST2P IS <RELATION> IO <SUB VAR> <LITCON> <BOOL SEC> <BOOL PRIM> <RELATION>

21 USE SUB 409 IF ST2P IS <CONJ> <RELATION> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <FUNC DES>

22 USE SUB 410 IF ST2P IS <DISJ> <RELATION> <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <FUNC DES>

23 USE SUB 411 IF ST2P IS <IMPL> <SUB VAR> <DISJ> <RELATION> <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <FUNC DES>

24 USE SUB 412 IF ST2P IS <BOOL EXP> IO <IMPL> <DISJ> <RELATION> <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <FUNC DES>

AMBIGUITY IN SUB 25

25 EITHER
USE SUB "29 IF ST2P IS <ARITH EXP> <SIMP A EXP> <TERM> <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON> <BOOL EXP> <IMPL>
USE SUB 430 IF ST2P IS <BOOL EXP> IO <IMPL> <DISJ> <RELATION> <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <FUNC DES>

26 USE SUB 431 IF ST2P IS <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON>

27 USE SUB 432 IF ST2P IS <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON>

28 USE SUB 433 IF ST2P IS <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON>

29 USE SUB 434 IF ST2P IS <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON>

30 USE SUB 435 IF ST2P IS <TERM> <LITCON> <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON>

31 USE SUB 436 IF ST2P IS <BOOL EXP> IO <IMPL> <DISJ> <RELATION> <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <FUNC DES>

32 EITHER
USE SUB 437 IF ST2P IS <ARITH EXP> <SIMP A EXP> <TERM> <FACTOR> <PRIMARY> <FUNC DES> <VARIABLE> IO <SUB VAR> <LITCON> <BOOL EXP> <IMPL>
USE SUB 438 IF ST2P IS <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMP ST>

33 USE SUB 422 IF ST2P IS IO

34 USE SUB 439 IF ST2P IS <STATEMENT> <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMP ST> <OP STATE>
USE SUB 421 IF ST2P IS <VARIABLE> IO <SUB VAR> <FUNCTION DES> <VARIABLE>

USE SUB 441 IF ST2P IS <ARITH EXP> <SIMP AEXP> <TERM> IO <SUB VAR> <LITCON> <FUNCTION DES> <VARIABLE>

USE SUB 442 IF STEP IS <ARITH EXP> IO <SUB VAR> <LITCON> <FUNCTION DES> <VARIABLE>

USE SUB 443 IF ST2P IS <BOOL PRIM> <VARIABLE> IO <SUB VAR> <LITCON> <FUNCTION DES> <RELATION>

USE SUB 444 IF ST2P IS <BOOL SEC> <VARIABLE> IO <SUB VAR> <LITCON> <FUNCTION DES> <RELATION>

USE SUB 445 IF ST2P IS <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <LITCON> <FUNCTION DES> <RELATION>

USE SUB 446 IF ST2P IS <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <LITCON> <FUNCTION DES> 

USE SUB 447 IF ST2P IS <DISJ> <BOOL SEC> <BOOL PRIM> <VARIABLE> IO <SUB VAR> <LITCON> <FUNCTION DES> 

USE SUB 448 IF ST2P IS <IMPL> <SIMP AEXP> <TERM> IO <SUB VAR> <LITCON> <FUNCTION DES> <VARIABLE, IO>

USE SUB 449 IF ST2P IS <EXPR> <VAR EXP> IO <SUB VAR> <LITCON> <FUNCTION DES> <VARIABLE, IO>

USE SUB 450 IF ST2P IS <ARITH EXP> <SIMP AEXP> <TERM> IO <SUB VAR> <LITCON> <FUNCTION DES> <VARIABLE>

USE SUB 451 IF ST2P IS <LIMP >

USE SUB 452 IF ST2P IS <IDLIST>

ST1P*ROW( <TYPE> VALUE / NEW T1

ST1P*ROW( <TYPE> ARRAY / NEW T1

USE SUB 453 IF ST2P IS <IO LIST>

USE SUB 454 IF ST2P IS <IDLIST>*

USE SUB 455 IF ST2P IS <IO LIST, IO>

USE SUB 456 IF STOP IS <IDLIST>

EITHER USE SUB 452 IF ST2P IS <IDLIST>* USE SUB 453 IF ST2P IS <IO LIST>

AMBIGUITY IN SUB 52

USE SUB 454 IF ST2P IS <IDLIST, IO>

USE SUB 456 IF STOP IS <IDLIST>
55 USE SUB 455 IF ST2P IS <DLIST> ID
56 USE SUB 458 IF ST2P IS <DLIST> ID
57 USE SUB 457 IF ST2P IS <DLIST> ID
58 USE SUB 459 IF ST2P IS <SPECIFIER> ID
59 USE SUB 460 IF ST2P IS ID
60 USE SUB 461 IF ST2P IS <EXPR> <ARITH EXP> <TERM> <FACT OR> <PRIM> <FUNC DES>
   <VARIABLE> ID <SUB VAR> <LITCON> <BOOL EXP> <IMPL> <DISJ>
61 USE SUB 465 IF ST2P IS <LIST LIST> <LIMP LIST> <LIMP ARI>
62 USE SUB 462 IF ST2P IS <LIMP LIST> <LIMP ARI>
63 USE SUB 463 IF ST2P IS <LIMP LIST> <LIMP ARI>
64 USE SUB 454 IF ST2P IS <NAME PART>
65 USE SUB 465 IF ST2P IS <NAME PART>
66 EITHER
   USE SUB 416 IF ST2P IS <SPEC LIST> <SPECIFIER>
   USE SUB 466 IF ST2P IS <SPEC LIST> <SPECIFIER>
   AMBIGUITY IN SUB 66
67 USE SUB 423 IF ST2P IS <ST LIST> <OP STATE>
   <STATEMENT> <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMPST>
68 USE SUB 467 IF ST2P IS <ST LST, <OP STATE>
   <STATEMENT> <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMPST>
69 USE SUB 458 IF ST2P IS <NAME PART>,
70 USE SUB 469 IF ST2P IS <STATEMENT> <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMPST> <OP STATE>
71 USE SUB 470 IF ST2P IS <DECL>
72 USE SUB 471 IF ST2P IS <ARITH EXP> <SIMP AEXP> <TERM> <FACTОР> <PRIM> <FUNC DES> <VARIABLE>
73 USE SUB 472 IF ST2P IS <ARITH EXP> <SIMP AEXP> <TERM> <FACTОР> <PRIM> <FUNC DES> <VARIABLE>
74 USE SUB 473 IF ST2P IS <BOOL EXP> <IMPL> <DISJ> <CONJ> <BOOL SEC> <BOOL PRIM> <VARIABLE>
75 USE SUB 474 IF ST2P IS <FOR LEL> <ARITH EXP> <SIMP AEXP> <TERM> <FACTОР> <PRIM> <FUNC DES>
USE SUB 418 IF ST2P IS ARITH_EXP ID
USE SUB 419 IF ST2P IS ARITH_EXP SIMP_EXP TERM
USE SUB 476 IF ST2P IS ID
USE SUB 475 IF ST2P IS ID
USE SUB 477 IF ST2P IS ARITH_EXP SIMP_EXP TERM
USE SUB 478 IF ST2P IS EXPR ARITH_EXP SIMP_EXP TERM
USE SUB 479 IF ST2P IS ARITH_EXP SIMP_EXP TERM
USE SUB 480 IF ST2P IS STATEMENT CL_STATE
USE SUB 481 IF ST2P IS STATEMENT CL_STATE
USE SUB 482 IF ST2P IS STATEMENT CL_STATE
USE SUB 483 IF ST2P IS CL_STATE
USE SUB 484 IF ST2P IS CL_STATE
USE SUB 485 IF ST2P IS ID
USE SUB 420 IF ST2P IS FOR_LIST FUNC DES
USE SUB 486 IF ST2P IS FOR_LIST FUNC DES
USE SUB 487 IF ST2P IS CL_STATE
USE SUB 488 IF ST2P IS CL_STATE
USE SUB 489 IF ST2P IS CL_STATE
94 EITHER
  USE SUB 499 IF ST2P IS <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMPST>
  USE SUB 490 IF ST2P IS <OP STATE>
95 USE SUB 491 IF ST2P IS <STATEMENT> <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMPST> <OP STATE>
96 USE SUB 492 IF ST2P IS <STATEMENT> <CL STATE> <GO STATE> <ASS STATE> <PROC CALL> <COMPST> <OP STATE>

401 P*P+1 STIP*ROW( <ID LIST> ) / NEW T1
401 P*P+1 STIP*ROW( 10 [ ] ) / NEW T1
402 P*P+1 STIP*ROW( ID ) / NEW T1
403 P*P+1 STIP*ROW( <PRIMARY> ** ) / NEW T1
404 P*P+1 STIP*ROW( <TERM> <OP/> ) / NEW T1
405 P*P+1 STIP*ROW( <SIMPLEX> <SIMPL> ) / NEW T1
406 P*P+1 STIP*ROW( <AE To LIST> ) / NEW T1
407 P*P+1 STIP*ROW( <ARITH EXP> <REL OP> ) / NEW T1
408 P*P+1 STIP*ROW( <RELATION> <REL OP> ) / NEW T1
409 P*P+1 STIP*ROW( <CONJ> AND ) / NEW T1
410 P*P+1 STIP*ROW( <DISJ> OR ) / NEW T1
411 P*P+1 STIP*ROW( <IMPL> IMPLIES ) / NEW T1
412 P*P+1 STIP*ROW( <BOOL EXP> EQUIV ) / NEW T1
413 P*P+1 STIP*ROW( <EXP LIST> ) / NEW T1
414 P*P+1 STIP*ROW( <ARITH EXP> ) / NEW T1
415 P*P+1 STIP*ROW( <LISP LIST> ) / NEW T1
416 P*P+1 STIP*ROW( <SPEC LIST> ) / NEW T1
417 P*P+1 STIP*ROW( <DEC LIST> ) / NEW T1
418 P*P+1 STIP*ROW( <ARITH EXP> STEP ) / NEW T1
419 P*P+1 STIP*ROW( <ARITH EXP> WHILE ) / NEW T1
420 P*P+1 STIP*ROW( <FOR LIST> ) / NEW T1
421 P*P+1 STIP*ROW( <VARIBL E> ) / NEW T1
446 P*P-1 / ST2P<DISJ> FOR PRODUCTION<DISJ> 11= <DISJ> OR <CONJ>
447 P*P-1 / ST2P<IMPL> FOR PRODUCTION<IMPL> 11= <IMPL> IMPLIES <DISJ>
448 P*P-1 / ST2P<BOOL EXP> FOR PRODUCTION<BOOL EXP> 11= <BOOL EXP> EQUIV <IMPL>
449 P*P-1 / ST2P<EXP LIST> FOR PRODUCTION<EXP LIST> 11= <EXP LIST> , <EXPR>
450 P*P-1 / ST2P<LIMPAIR> FOR PRODUCTION<LIMPAIR> 11= <ARITH EXP> 1 <ARITH EXP>
451 P*P-1 / ST2P<LIM LIST> FOR PRODUCTION<LIM p LIST> 11= <LIM p LIST> <LIM PAIR>
452 P*P-1 / ST2P<SPECIFIER> FOR PRODUCTION<SPECIFIER> 11= <TYPE> <ID LIST>
453 P*P-1 / ST2P<DECL> FOR PRODUCTION<DECL> 11= <TYPE> <ID LIST>
454 P*P-1 / ST2P<SPECIFIER> FOR PRODUCTION<SPECIFIER> 11= <TYPE> VALUE <IDLIST>
455 P*P-1 / ST2P<DECL> FOR PRODUCTION<DECL> 11= <TYPE> ARRAY <ID LIST>
456 ST1PROW(TYPE) ARRAY <ID LIST> [ ) } / NEW T1
457 P*P-1 / ST2P<SPECIFIER> FOR PRODUCTION<SPECIFIER> 11= ARRAY <ID LIST>
458 ST1PROW ARRAY <ID LIST> [ ) ) / NEW T1
459 P*P-1 / ST2P<LIM LIST> FOR PRODUCTION<SPEC LIST> 11= <SPEC LIST> ; <SPECIFIER>
460 ST1PROW(CONSTANT IO 1= ) ) / NEW T1
461 P*P-1 ST2P<DECL> FOR PRODUCTION<DECL> 11= Const Type 10 1= <EXPR>
462 P*P-1 / ST2P<DECL> <LIM p LIST> ) ] / NEW T1
463 P*P-1 ST2P<DECL> FOR PRODUCTION<DECL> 11= ARRAY <ID LIST> [ <LIM p LIST>
464 P*P-1 ST2P<DECL> FOR PRODUCTION<DECL> 11= SWITCH <NAME PART>
465 ST1PROW PROCEDURE <NAME PART> ] ) ) / NEW T1
466 ST1PROW PROCEDURE <NAME PART> <SPEC LIST> ) ) ) / NEW T1
467 P*P-1 ST2P<DECL> <ST LIST> END FOR PRODUCTION<DECL> 11= PROCEDURE <NAME PART> <SPEC LIST>
468 ST1PROW FUNCTION <NAME PART> ] ) ) / NEW T1
469 P*P-1 / ST2P<DECL> FOR PRODUCTION<DECL> 11= FUNCTION <NAME PART> <STATEMENT>
470 P*P-1 ST2P<DECL LIST> FOR PRODUCTION<DECL LIST> 11= <DECL LIST> <DECL>
STP+ROWC
CARITH EXP> UNTIL
/ NEW T1

P+P-1 / ST2P<FOR L EL> FOR PRODUCTION <FOR L EL> 1 = <ARITH EXP> STEP <ARITH EXP> UNTIL

P+P-1 / ST2P<FOR L EL> FOR PRODUCTION <FOR L EL> 1 = <ARITH EXP> WHILE <BOOL EXP>

P+P-1 / ST2P<FOR LIST> FOR PRODUCTION <FOR LIST> 1 = <FOR LIST>

P+P-1 / ST2P<GO STATE, FOR PRODUCTION <GO STATE> 1 = GO TO 10

STP+ROWC ( GO TO 10 ) / NEW T1

P+P-1 / ST2P<GO STATE> FOR PRODUCTION <GO STATE> 1 = GO TO 10 

STP+ROW ( ST2P<ASS STATE, FOR PRODUCTION <ASS STATE> *VARIABLE> <EXPR> 

P+P-1 / ST2P<ASS STATE> FOR PRODUCTION <ASS STATE> *VARIABLE> <ASS STATE>,

P+P-1 / ST2P<COMP ST> FOR PRODUCTION <COMP ST> <COMP LIST>

STP+ROW ( BEGIN <DECL LIST> ; ) / NEW T1

P+P-1 / ST2P<COMP ST> FOR PRODUCTION <COMP ST> 1 = BEGIN <DECL LIST> ; <ST LIST>

P+P-1 / ST2P<COMP ST> FOR PRODUCTION <COMP ST> 1 = <CL STATE>

P+P-1 / ST2P<CL STATE> FOR PRODUCTION <CL STATE> 1 = 10 

STP+ROW ( FOR 10 1 = ) / NEW T1

STP+ROW ( FOR 10 1 = <FOR LIST> cc ) / NEW T1

P+P-1 / ST2P<CL STATE> FOR PRODUCTION <CL STATE> 1 = FOR 10 

P+P-1 / ST2P<CL STATE> FOR PRODUCTION <CL STATE> 1 = FOR 10

P+P-1 / ST2P<CL STATE> FOR PRODUCTION <CL STATE> 1 = IF <BOOL EXP> THEN <CL STATE>

P+P-1 / ST2P<CL STATE> FOR PRODUCTION <CL STATE> 1 = IF <BOOL EXP> THEN <CL STATE>

P+P-1 / ST2P<ST LIST> FOR PRODUCTION <ST LIST> 1 = <ST LIST> ; <STATEMENT>

P+P-1 / ST2P<PROGRAM> FOR PRODUCTION <PROGRAM> 1 = PHI <STATEMENT> PHI / NEW T1