An implementation of a parsing algorithm for LL(k) grammars.

Chu, Pei-yue
1983

Find more at https://preserve.lib.lehigh.edu/

This document is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
AN IMPLEMENTATION OF

A PARSING ALGORITHM FOR LL(K) GRAMMARS

by

Pei-yue Chu

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computing Science

Department of Mathematics

Division of Computing and Information Science

Lehigh University

1982
This thesis is accepted and approved in partial fulfilment of the requirements for the degree of Master of Science.

August 27, 1982
Date

Professor in Charge

Head of the Division
Acknowledgment

The author wishes to thank Professor Samuel L. Gulden for his helpful suggestions and guidance in the preparation of this paper.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>3.</td>
<td>Definition of LL(k) Grammars</td>
<td>4</td>
</tr>
<tr>
<td>4.</td>
<td>Analysis of the Parsing Algorithm</td>
<td>13</td>
</tr>
<tr>
<td>5.</td>
<td>Implementation of the Parsing Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>6.</td>
<td>Conclusions</td>
<td>52</td>
</tr>
<tr>
<td>7.</td>
<td>Bibliography</td>
<td>53</td>
</tr>
<tr>
<td>8.</td>
<td>Vita</td>
<td>54</td>
</tr>
</tbody>
</table>
1. Abstract

AN IMPLEMENTATION OF A PARSING ALGORITHM FOR LL(k) GRAMMARS

by Pei-yue Chu

A k-predictive parsing algorithm described by Aho and Ullman is implemented in this paper. The class of context-free grammars which permits deterministic left-to-right top down parsing with a k lookahead symbols is defined and procedures are given for determining if a context free grammar is LL(k) for a given value of k. The pushdown list, the parsing table, the lookahead string and the left parse, which form the essence of the predictive parsing algorithm, are discussed. A program written in PASCAL, using k equal to two, to implement the concept of the parsing algorithm is partially discussed.
2. Introduction

For a program written in a higher level programming language such as PASCAL, FORTRAN or COBOL to be "executed" by the computer, the program (called source program) has to be converted into an object program which consists of binary machine instructions. The source program must first be analyzed to uncover its underlying syntactic structure. This process is called parsing. A compiler is itself a program written partly for this purpose. There are many different parsing techniques. The algorithm to be discussed in this paper is characterized by the facts that the input string is scanned once from left to right and the parsing process is completely deterministic, in other words, one-pass no backtrack parsing. We will concentrate on the algorithm for parsing LL(k) grammars for which the left parser can be made to work deterministically if it is allowed to look at k input symbols to the right of the current input position.

LL(k) grammars were first defined by Lewis and Stearns [4] and were called TD(k) grammars, TD being an acronym for "top-down". The theory of LL(k) grammars was later extensively developed by Rosenkrantz and
Throughout the paper we will attempt to define the concept of LL(k) grammars by introducing formal language theory, the algorithms discussed by Barrett and Couch [2] to obtain the FIRST and the FOLLOW sets and to test for LL(k) conditions. We shall use the "k-predictive parsing algorithm" described by Aho and Ullman [1] to set up the parsing table for LL(k) grammars and to parse the input strings. Lastly, an implementation of the algorithm, using $k = 2$, will be discussed.
3. Definition of LL(k) grammars

In order to explain what LL(k) grammars are, we must first define some concepts in formal language theory. A computer program may be constructed from a sequence of characters drawn from the computer's character set. Such a sequence may be composed into tokens (corresponding to words in English) and the tokens composed into sentences through a grammar.

3.1 Alphabets and tokens

An alphabet is a finite set of tokens, given at the time when the source language is defined. Every source program consists of some sequence of tokens drawn from the alphabet of the source language. The symbol $\Sigma$ is used to designate an alphabet.

3.2 Strings

A string is a sequence of elements drawn from an alphabet. The length of the string is the number of characters in the string. Thus, the length of the string "PASCAL" is 6. The set of all strings of length one or more, consisting of members of $\Sigma$ will be
designated $\Sigma^+$. A string of length zero is called the empty string, written $\lambda$. The set $\Sigma^*$ is the set $\Sigma^+$ union the empty string set, that is, $\Sigma^* = \Sigma^+ \cup \{\lambda\}$. A language on $\Sigma$ is a subset of $\Sigma^*$.

3.3 Terminals and nonterminals

A terminal is a member of $\Sigma$. In order to define structural rules for a grammar we introduce a finite set of objects called nonterminals. The set of nonterminals is disjoint from $\Sigma$. The set of nonterminals will usually be designated $\text{N}$. 

3.4 Production rules and grammars

A production rule, or production for short, has the general form $x \rightarrow y$ where $x$ and $y$ are strings in the terminal and nonterminal sets of a given grammar. The $y$ may be an empty string, but $x$ can never be. Productions are used to generate other strings in the language. For example, if we start with a string of the form $axb$ we can derive a new string of the form $ayb$ by replacing the $x$ with the $y$ of the production.

A grammar is a four tuple $(N, \Sigma, P, S)$ where $\Sigma$ is
the terminal set, $N$ is the nonterminal alphabet, $\Sigma$ and $N$ are disjoint, $P$ is a finite set of productions of the form $x \rightarrow y$, where $x$ and $y$ are in $(N \cup \Sigma)^*$, and $x$ contains at least one element in $N$ and $S$ is a designated start symbol in $N$. The language of a grammar is the set of strings in $\Sigma^*$ which can be derived from $S$ in a finite number of steps using the productions. Strings in $(N \cup \Sigma)^*$ are called sentential forms.

3.5 Context free grammars

Chomsky [3] distinguished four general classes of grammars: unrestricted grammars, context sensitive grammars, context free grammars and right linear grammars. In the context free grammar, or CFG, each production has the form $A \rightarrow w$, where $A$ is a member of $N$ and $w$ is any string in $(N \cup \Sigma)^*$. $A$ can be replaced by $w$ regardless of the position of $A$. The grammar $G$ illustrated below is an example of the context free grammar.

$$G = (N, \Sigma, P, S)$$
where $N = \{E, T, F\}$,
$\Sigma = \{+, *, (, ), a\}$
$S = E$

and $P$ is the following productions:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$

6
A derivation tree displays the derivation of some sentential form in a grammar. Each node of a derivation is associated with a single terminal or nonterminal. If we use the grammar $G$ described in §3.5 as an example, a derivation tree can be shown as follows:

3.7 Ambiguous and unambiguous grammars

If we have a grammar $G = (N, \Sigma, P, S)$ and a sentence $w$ and there exists more than one distinctive derivation trees for $w$ then we say that the grammar is ambiguous. If no sentence has more than one derivation tree, we then say that the grammar is unambiguous. For example, grammar $G$ with productions as follows:
is ambiguous because there are two distinctive derivation trees for the sentence $a+a*a$.

(a) $E \rightarrow E + E$
(b) $E \rightarrow E * E$

3.8 Left-recursive nonterminals

If a nonterminal $A$ has at least one production in the form of $A \rightarrow Aw$, $w$ being a non-empty string, or, a derivation can be induced so that $A \rightarrow^* Au$, $u$ being a non-empty string, we then say that $A$ is a left-recursive nonterminal.

3.9 LL(k) grammars

LL(k) grammar is a context free grammar which permits deterministic left-to-right top-down recognition with a lookahead $k$ symbols, $k$ being greater or equal to one. Suppose we have grammar $G = (N, \Sigma, P, S)$ and a sentence $w = a_1a_2\ldots a_n$ in the language $L(G)$ then there
exists a unique sequence of left sentential form \( \alpha_0, \alpha_1, \ldots, \alpha_m \) such that \( S = \alpha_0, \alpha_i \to \alpha_{i+1} \) for \( 0 \leq i < m \) and \( \alpha_m = w \). The left parse for \( w \) is \( p_0p_1 \cdots p_{m-1} \).

Suppose that we want to find this left parse by scanning \( w \) once from left to right. We might try to do this by constructing \( \alpha_0, \alpha_1, \ldots, \alpha_m \), the sequence of left-sentential forms. If \( \alpha_i = a_1 \cdots a_j A \beta \), then at this point we could have read the first \( j \) input symbols and compared them with the first \( j \) symbols of \( \alpha_i \). It would be desirable if \( \alpha_{i+1} \) could be determined knowing only \( a_1 \cdots a_j \) (the part of the input we have scanned to this point), the next few input symbols \( (a_{j+1}a_{j+2} \cdots a_{j+k} \text{ for some fixed } k) \), and the nonterminal \( A \). If these three quantities uniquely determine which production is to be used to expand \( A \), we can then precisely determine \( \alpha_{i+1} \) from \( \alpha_i \) and the \( k \) input symbols \( a_{j+1}a_{j+2} \cdots a_{j+k} \).

A grammar in which each leftmost derivation has this property is said to be an LL(k) grammar.

3.91 LL(k) productions

A production \( A \to x_1 \) in a CFG is called an LL(k) production if in \( G \),

\[
S \to ^* wAy \to wx_1y \to ^* wz \ldots,
\]
and,

\[
S \rightarrow ^* wA\ y' \rightarrow wx_2y' \rightarrow ^* wz \ldots \\
\]

with \(|z| = k\) and \(z \subseteq \Sigma^*\) implies \(x_1 = x_2\).

3.92 LL(k) nonterminals

A nonterminal in a CFG is called an LL(k) nonterminal if all of its productions are LL(k) productions. For example, in the grammar G below:

\[
\begin{align*}
S & \rightarrow A \ b \ c \\
S & \rightarrow a \ A \ c \ b \\
A & \rightarrow \_ \\
A & \rightarrow b \\
A & \rightarrow c
\end{align*}
\]

\(S\) is an LL(1) nonterminal, while \(A\) is an LL(2) nonterminal. The strings derivable from \(S\) are (first production) bbc, cbc, and bc and (second production) acb, abcb, and accb. Clearly, the second production is uniquely selected on the lookahead symbol \(a\), while the first production is selected on symbols \(b\) or \(c\). For the \(A\) productions we need to consider the sentential forms in which \(A\) is embedded, e.g., \(Abc\) and \(aAcb\). The string preceding \(A\) is different so we only need to consider whether the three \(A\) productions can be distinguished within \(Abc\) and within \(aAcb\). In the former, we have the three lookahead strings bc, bb and cb. These are
distinguishable with k=2 but not with k=1. In the latter production the lookahead strings are cb, cc and bc, again distinguishable with k=2. The A productions are therefore LL(2) and S productions are LL(1). It is clear that an LL(k) grammar is also an LL(k+1) grammar for k greater or equal to one. A grammar such that every production is LL(k) is an LL(k) grammar.

3.93 Properties of LL(k) grammars

Every LL(k) grammar is unambiguous. An ambiguous grammar will lead to more than one derivation tree for a given sentence which is a contradiction to the definition of LL(k) grammars in which every derivation step is uniquely determined by only one production rule.

An LL(k) grammar has no left-recursive nonterminals. Given a simple left-recursive nonterminal of the form \( A \rightarrow A \ x \ | \ A \ y \ | \ldots \ | \ w \ | \ z \ | \ldots \), we can transform it into an LL(1) grammar through the following procedures:

We first stratify this production set by introducing two new nonterminals B and C, as follows:

\[
\begin{align*}
A & \rightarrow A \ B \ | \ C \\
B & \rightarrow x \ | \ y \ | \ldots \\
C & \rightarrow w \ | \ z \ | \ldots
\end{align*}
\]
Note that B collects the strings past the A in the left-recursive A productions and that C collects all other A production right hand parts. Then the productions $A \rightarrow A \ B \mid C$ are rewritten as $A \rightarrow C \ A'$; $A' \rightarrow B \ A' \mid \lambda$, where $A'$ is another new nonterminal.
4. Analysis of the parsing algorithm

We can parse LL(k) grammars by using k-predictive parsing algorithm. However, before we attempt to discuss it we need to develop two useful functions: FIRST\(_k\)(w) and FOLLOW\(_k\)(A).

4.1 FIRST and FOLLOW sets

The domain of FIRST\(_k\) is some string w in \( \{ \text{N U } \Sigma \}^* \) and the domain of FOLLOW\(_k\) is a nonterminal A in N. The functions are thus defined as follow:

\[
\text{FIRST}_k(w) = \{ x \mid w \rightarrow^* xy, x, y \subseteq \Sigma^* \},
\]

and

\[
\text{FOLLOW}_k(A) = \{ x \mid S \rightarrow^* uAy \text{ and } x \subseteq \text{FIRST}_k(y) \}
\]

where the derivations are left most. That is, FIRST\(_k\)(w) for some string w is the set of all leading terminal strings of length k or less in the strings derivable from w. FOLLOW\(_k\)(A) is the set of all derivable terminal strings of length k or less which can follow A in some left most sentential form.

To compute the FIRST and the FOLLOW set we can use the following rules:

(1) FIRST\(_k\)(aw) = a FIRST\(_k\)-1(w) for any string w,

where a is in the terminal set.
(2) \( \text{FIRST}_k(\lambda) = \{ \lambda \} \), \( \lambda \) being the empty string.

(3) \( \text{FIRST}_k(xy) = \text{FIRST}_k(\text{FIRST}_k(x) \text{ FIRST}_k(y)) = \text{FIRST}_k(x \text{ FIRST}_k(y)) = \text{FIRST}_k(\text{FIRST}_k(x) y) \).

(4) Given a production \( A \rightarrow w \) in \( G \), \( \text{FIRST}_k(A) \) contains \( \text{FIRST}_k(w) \).

(5) Given a production \( A \rightarrow uXy \) in \( G \), \( \text{FOLLOW}_k(X) \) contains \( \text{FIRST}_k(y \text{ FOLLOW}_k(A)) \).

(6) Given a production \( A \rightarrow uX \), if \( X \) is the last symbol of the production, then \( \text{FOLLOW}_k(X) \) contains \( \text{FOLLOW}_k(A) \). And if we designate \( \$ \) the endmarker then \( \text{FOLLOW}_k(X) \) also contains \( \$ \).

(7) \( \text{FOLLOW}_k(S) \) contains \( \$ \), where \( S \) is the start symbol of \( G \).

Let us consider the following LL(1) grammar:

1. \( E \rightarrow T \ E' \)
2. \( E' \rightarrow + T \ E' \)
3. \( E' \rightarrow \lambda \)
4. \( T \rightarrow F \ T' \)
5. \( T' \rightarrow * F \ T' \)
6. \( T' \rightarrow \lambda \)
7. \( F \rightarrow ( \ E \ ) \)
8. \( F \rightarrow a \)

then \( \text{FIRST}_1(A) \) and \( \text{FOLLOW}_1(A) \) are as follows:

\[
\text{FIRST}_1(F) = \text{FIRST}_1(T) = \text{FIRST}_1(F) = \{ \{ , n \} \}
\]
FIRST₁(E') = { +, , }  
FIRST₁(T') = { *, , }  
FOLLOW₁(E) = FOLLOW₁(E') = { ) , $ }  
FOLLOW₁(T) = FOLLOW₁(T') = { +, ), $ }  
FOLLOW₁(F) = { +, * , ), $ }  

4.2 Checking LL(k) grammars

Having generated FIRST and FOLLOW sets we will be able to check if a given grammar is an LL(k) grammar. Let G be a CFG, then G is LL(k) if and only if for every distinctive pair of productions A → u and A → v, FIRSTₖ(u) ∩ FIRSTₖ(v) is empty set and if empty string is derivable from the productions then FIRSTₖ(u) ∩ FIRSTₖ(v) ∩ FOLLOWₖ(A) must be empty.

4.3 Generating LL(k) parsing table

After obtaining the FIRST set for each string of the right hand side of the production and the FOLLOW set for each of the nonterminals of the grammar, we can use the following algorithm to generate the LL(k) parsing table.

Let M be the parsing table for grammar $G = (N, \Sigma, P, S)$. 

15
(1) For each production \( A \rightarrow w, A \) being non-terminal and \( w \) a string, we have \( M( A, x) = (w, i) \); where \( x \) is in FIRST\(_k\)(\( w \)); and if empty string is in FIRST\(_k\)(\( w \)) then \( x \) is contained in FIRST\(_k\)(\( w \)) union FOLLOW\(_k\)(\( A \)); \( i \) is the production number.

(2) \( M(a, au) = \text{"pop" for all } u \text{ in } \Sigma^{*(k-1)} \)

(3) \( M(\$, $) = \text{"accept"} \)

(4) otherwise, \( M(X,u) = \text{"error"} \)

If we use the FIRST and FOLLOW sets illustrated in §4.1 for LL(1) grammar then we can construct the parsing table as follows:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T E', 1</td>
<td>error</td>
<td>error</td>
<td>T E', 1</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>E'</td>
<td>error</td>
<td>+T E', 2</td>
<td>error</td>
<td>error</td>
<td>( \lambda, 3 )</td>
<td>( \lambda, 3 )</td>
</tr>
<tr>
<td>T</td>
<td>F T', 4</td>
<td>error</td>
<td>error</td>
<td>F T', 4</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>T'</td>
<td>error</td>
<td>( \lambda, 6 )</td>
<td>*F T', 5</td>
<td>error</td>
<td>( \lambda, 6 )</td>
<td>( \lambda, 6 )</td>
</tr>
<tr>
<td>F</td>
<td>( a, 8 )</td>
<td>error</td>
<td>error</td>
<td>(E), 7</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>a</td>
<td>pop</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>+</td>
<td>error</td>
<td>pop</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>*</td>
<td>error</td>
<td>error</td>
<td>pop</td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>(</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>pop</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>)</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>pop</td>
<td>error</td>
</tr>
<tr>
<td>$</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>
4.4 Parsing the LL(k) grammars

We shall use "k-predictive parsing algorithm" to parse the LL(k) grammars. For a CFG, \( G = (N, \Sigma, P, S) \) the algorithm uses an input tape, a pushdown list and an output tape as shown in the following figure:

The input tape contains the input string to be parsed; \( $ \) is added to the end of the input string as the sentinel. The input tape is read by an input head capable of reading the next k input symbols. The string scanned by the input head is called the lookahead string.
The pushdown list contains a string $x\alpha\$, where $\alpha$ is a string of pushdown symbols and $\$ is the endmarker for the pushdown list. The symbol $X$ is on top of the pushdown list. We shall use $r$ to represent the alphabet of the pushdown list symbols (excluding $\$).

The output tape contains a string $\pi$ of production indices.

We shall represent the configuration of a predictive parsing algorithm by a triple $(x, X\alpha, \pi)$, where

(1) $x$ represents the unused portion of the original input string.

(2) $X$ represents the string on the pushdown list (with $X$ on top).

(3) $\pi$ is the string on the output tape.

The action of a $k$-predictive parsing algorithm is dictated by a parsing table $M$, which is a mapping from $(r \cup \{|\$\}| \times \Sigma^*)$ to a set containing the following elements:

(1) $(\beta, i)$, where $\beta$ is in $r^*$ and $i$ is a production number. Presumably, $\beta$ will be either the
right side of production i or a representation of it.

(2) pop.

(3) accept.

(4) error.

The parsing algorithm parses an input by making a sequence of moves. In a move the lookahead string $u$ and the symbol $X$ on top of the pushdown list are determined. Then the entry $M(X, u)$ in the parsing table is consulted to determine the actual move to be made. We shall describe the moves of the parsing algorithm in terms of a relation $\vdash$ on the set of configurations. Let $u$ be $\text{FIRST}_k(x)$. We write

\[
1. \quad (x, Xa, \pi) \vdash (x, \beta a, \pi_i) \text{ if } M(X, u) = (\beta, i).
\]

Here the top symbol $X$ on the pushdown list is replaced by the string $\beta$ and the production number $i$ is appended to the output. The input head is not moved.

\[
2. \quad (x, a\alpha, \pi) \vdash (x', \alpha, \pi) \text{ if } M(a, u) = \text{pop and } x = ax'. \quad \text{When the symbol on top of the pushdown list matches the first symbol of the lookahead string, the pushdown list is popped and the input head is moved one symbol to the right.}
\]
(3) If the parsing algorithm reaches configuration \( (\$, \$, \pi) \), then parsing ceases, and the output string \( \pi \) is the parse of the original input string.

(4) If the parsing algorithm reaches configuration \( (x, X\alpha, \pi) \) and \( M(X, u) = \text{error} \), then parsing ceases, and an error message is reported. The configuration \( (x, X\alpha, \pi) \) is called an error configuration.

If we take the following LL(1) grammar as an example, we can construct a 1-predictive parsing algorithm. The productions are numbered as follows:

\[
\begin{align*}
\text{(1)} & \quad S \rightarrow aAS \\
\text{(2)} & \quad S \rightarrow b \\
\text{(3)} & \quad A \rightarrow a \\
\text{(4)} & \quad A \rightarrow bSA
\end{align*}
\]

A parsing table is shown in the following figure:

<table>
<thead>
<tr>
<th>Symbol on top of push-down list</th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>Lookahead string</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aAS,1</td>
<td>b,2</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>a,3</td>
<td>bSA,4</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>pop</td>
<td>error</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>error</td>
<td>pop</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>error</td>
<td>error</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

Using this table, we can parse the input string
abbab as follows:

\[
\begin{align*}
(a & b b a b \ S, S, ) \rightarrow (a & b b a b \ S, a A S \ S, 1) \\
& \rightarrow (b b a b \ S, A S \ S, 1) \\
& \rightarrow (b b a b \ S, b S A S \ S, 14) \\
& \rightarrow (b a b \ S, S A S \ S, 14) \\
& \rightarrow (b a b \ S, b A S \ S, 142) \\
& \rightarrow (a b \ S, A S S, 142) \\
& \rightarrow (a b \ S, a S S, 1423) \\
& \rightarrow (b \ S, S S, 1423) \\
& \rightarrow (b \ S, b \ S, 14232) \\
& \rightarrow (S, S, 14232)
\end{align*}
\]

So, the left parse of abbab is 14232.
5. Implementation of the parsing algorithm

The implementation of the algorithm is written in PASCAL, using \( k = 2 \). Throughout the discussion of the program, we shall use the following LL(2) grammar, \( G_2 \) as an example:

\[
\begin{align*}
(1) & \quad S \rightarrow A \ b \ c \\
(2) & \quad A \rightarrow ^* \\
(3) & \quad A \rightarrow b \\
(4) & \quad S \rightarrow a \ A \ c \ b \\
(5) & \quad A \rightarrow c
\end{align*}
\]

where symbol \(^*\) is the empty string.

5.1 Production tree

Procedure \texttt{getproduction} is to read the productions from the input files. Meanwhile, it will set up the productions as the linked tree for the grammar \( G = ( \mathcal{N}, \Sigma, P, S ) \), it shall designate the first symbol read as the start symbol \( S \). The nonterminals will be those symbols found on the left hand side of the arrow sign \( \rightarrow \).

We shall introduce a number set called "alset". Each symbol in \( \mathcal{N} \) is assigned a number according to the order in which it is read. Those symbols not found on the left hand side of the productions are considered terminals and are also assigned a number value.
The partial data structure of the productions is as follows:

vertex = (leftproduction, rightproduction);
ptr = *production;
production = RECORD
next, link: ptr;
entry: ALFA;
count: INTEGER;
.........
CASE vertex OF
leftproduction:
(first, second, follow:
alset; ...);
rightproduction:
(num: INTEGER;
firstset, llset:
alset; ...);
END;

where,

"next" is to link to the succeeding symbols.
"link" is to link to the next production.
"entry" is for the symbol.
"count" is the number value of the symbol.
"first" is for FIRST₁(A).
"second" is for FIRST₂(A).
"num" is for the production number.
"firstset" is for the FIRST₂(w), w being a string.
"llset" is FIRST₂(w) if empty string is not in FIRST₂(w), or, FIRST₂(w) U FOLLOW₂(A) if empty string is in FIRST₂(w).

Each symbol is obtained through the procedure getword, using blank as the sentinel and the end of the line is used to signal the end of the production. So, after the execution of the procedure getproduction, we have a production tree. If we use G₂ described on page 22 as an example then the production tree looks like
After the production tree is set up, we will attempt to find FIRST₂(w) and FOLLOW₂(A), w being the right hand side of the production and A a nonterminal.

5.2 FIRST set

To find FIRST₂(w) we need procedure buildsecond and procedure transsecond. Procedure buildsecond searches the production tree to find a string which is either of length 1 and a terminal symbol or with the first two symbols being terminals. The symbols are represented by number value and are stored in the fields "firstset" for FIRST₂(w) and "second" for FIRST₂(A). Those strings with nonterminals will wait for the procedure transsecond to transform them into terminals. Procedure buildsecond is as follows:
PROCEDURE buildsecond(VAR rrt:ptr);

VAR
  cursor, cur1:ptr;
  temp: ALFA;

BEGIN (*buildsecond*)
  cursor := rrt;
  WHILE cursor <> NIL DO
    BEGIN
      cur1 := cursor^.next;
      WHILE cur1 <> NIL DO
        BEGIN
          IF cur1^.count IN termset THEN
            BEGIN
              IF cur1^.next = NIL THEN
                BEGIN
                  cursor^.second := cursor^.second + [cur1^.count];
                  cur1^.firstset := cur1^.firstset + [cur1^.count]
                END;
            ELSE
              BEGIN
                IF (cur1^.next^.count IN termset) THEN
                  BEGIN
                    temp := blk;
                    temp[1] := cur1^.entry[1];
                    temp[2] := cur1^.next^.entry[1];
                    fadds(cur1^.firstset, cursor^.second, kk, temp)
                  END;
                ELSE
                  BEGIN
                    cur1^.secdset := FALSE;
                    cursor^.allterm := FALSE;
                  END;
              END;
            ELSE
              BEGIN
                cur1^.secdset := FALSE;
                cursor^.allterm := FALSE;
              END;
          END;
        END;
      END;
    END;
  END (*buildsecond*);
Where "termset" is a set of terminals and procedure \texttt{fadds} is to add string of length 2 to the fields "firstset" and "second" and to append the tree "alpha" for alphabet. Procedure \texttt{fadds} is described below:

\begin{verbatim}
PROCEDURE fadds(VAR f, s: alset; VAR jj: INTEGER; wd: ALFA);
VAR
cursor: ptr;
BEGIN
cursor := alpha;
search(cursor, wd);
IF cursor = NIL THEN
BEGIN
jj := jj + 1;
adds(alpha, wd, 2, jj);
f := f + [jj];
s := s + [jj]
END
ELSE
BEGIN
f := f + [cursor^. count];
s := s + [cursor^. count]
END
END (*fadds*);
\end{verbatim}

For the strings having a nonterminal as the first symbol we shall use procedure \texttt{fnonterm} to derive the FIRST\textsubscript{2}(w) and FIRST\textsubscript{2}(A). Whereas for those strings having a terminal as the first symbol and a nonterminal as the second symbol we shall use procedure \texttt{fterm}.

Before discussing procedure \texttt{fterm}, we shall first try to obtain FIRST\textsubscript{1}(A) by executing procedure 26
buildfirst and procedure transfirst.

Procedure buildfirst looks up the production tree. For each production \( A \rightarrow w \), we define \( \text{FIRST}_1(A) = \text{FIRST}_1(w) \). Here we introduce a "nullset" which contains all of the nullable nonterminals. Nullset can be obtained by using procedure findnull.

Let \( w = x_1x_2...x_n \); we say that \( \text{FIRST}_1(w) = x_i \) if \( x_i \) is not in nullset; \( x_i \) can either be a terminal or a nonterminal. However, if \( x_i \) is a nullable nonterminal then \( \text{FIRST}_1(w) = \text{FIRST}_1(x_i) \cup \text{FIRST}_1(x_2) \cup .... \). The procedure buildfirst is as follows:

PROCEDURE buildfirst(VAR rrt: ptr);

VAR
    cursor, cur, cur2: ptr;
    stop: BOOLEAN;
BEGIN
    cursor := rrt;
    WHILE cursor <> NIL DO
    BEGIN
        cur := cursor^.next;
        WHILE cur <> NIL DO
        BEGIN
            cursor^.first := cursor^.first + [cur^.count];
            IF (cur^.count IN nullset) THEN
                IF cur^.next <> NIL THEN
                BEGIN
                    cur2 := cur^.next;
                    stop := FALSE;
                    REPEAT
                        cursor^.first :=
                        cursor^.first +
                    END;
                END;
            END;
        END;
    END;
END;
[cur2^ .count];
IF NOT (cur2^ .
count IN nullset)
THEN
  stop := TRUE
ELSE
  cur2 :=
  cur2^ .next
UNTIL (stop=TRUE)
OR (cur2=NIL);
END;
cur := cur^ . link
END;
cursor := cursor^ . link
END;
END (*buildfirst*);

After executing the procedure buildfirst we may
find some nonterminals in the FIRST set. We shall use
procedure transfirst to search the production tree and
to replace those nonterminals with their FIRST set,
FIRST_1(A). Procedure transfirst is as follows:

PROCEDURE transfirst(VAR rrt: ptr);
VAR
  kursor, cur1: ptr;
  save, hold: alset;
  stop: BOOLEAN;

PROCEDURE fclosure (VAR ssave, hhold: alset;
  VAR kur1: ptr);
VAR
  temp: alset;
BEGIN
  REPEAT
    IF kur1^. count IN hhold
    THEN
      BEGIN
28
IF kur1^.count IN ssave
THEN
  hhold := hhold - [kur1^.count]
ELSE
BEGIN
  hhold := hhold + kur1^.first;
  ssave := ssave + [kur1^.count];
  temp := hhold * termset;
  ssave := ssave + temp;
  hhold := hhold - ssave;
END;
IF hhold = []
THEN
  stop := TRUE
ELSE
  kur1 := kur1^.link
UNTIL (kur1 = NIL) OR (stop = TRUE);
END (*fclosure*);
Again, if we use $G_2$ (on page 22) as an example we shall obtain $\text{FIRST}_1(A)$ through procedure _buildfirst_ as follows:

<table>
<thead>
<tr>
<th>nonterminals</th>
<th>FIRST set</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>{ A, b, a}</td>
</tr>
<tr>
<td>A</td>
<td>{ ^, b, c }</td>
</tr>
</tbody>
</table>

Finally, using procedure _transfirst_ we shall replace those nonterminals found in the FIRST set as follows:

<table>
<thead>
<tr>
<th>nonterminals</th>
<th>FIRST set</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>{ ^, b, c, a }</td>
</tr>
<tr>
<td>A</td>
<td>{ ^, b, c }</td>
</tr>
</tbody>
</table>

Now let us resume the discussion of procedure _fterm_. As mentioned earlier, procedure _fterm_ is to _transclose_ those strings with the first symbol being terminal and the second symbol nonterminal. We shall search the production tree to obtain the $\text{FIRST}_1(A)$, A being the second symbol of the string. If we designate u the $\text{FIRST}_1(A)$ then, for each terminal in u (excluding the empty string) we shall form a new string by
concatenating the first symbol of the string with the terminal found in the \textit{FIRST}_1(A). This new string is then added to the \textit{FIRST}_2(A). We can have it done by using procedure \texttt{ffadds} which is described as follows:

\begin{verbatim}
PROCEDURE ffadds(save:alset; VAR f, sec: alset; wd: ALFA);

VAR
cursor: ptr;
temp: ALFA;

BEGIN
cursor := alpha;
WHILE cursor <> NIL DO
    BEGIN
        IF (cursor^ count IN save) AND (cursor^ count <> nullcount)
        THEN
            BEGIN
                temp := blk;
                temp[1] := wd[1];
                temp[2] := cursor^ entry[1];
                fadds(f, sec, kk, temp);
            END;
            cursor := cursor^ . link
        END;
    END;
END (*ffadds*);
\end{verbatim}

If the empty string is in \textit{FIRST} set \textit{u} and the production string \textit{w} is of length 2 then the first terminal symbol of \textit{w} is added to the set of \textit{FIRST}_2(A); otherwise a recursive call for procedure \texttt{fterm} is made. Procedure \texttt{fterm} is as follows:

\begin{verbatim}
PROCEDURE fterm(cursor, crt: ptr; wd: ALFA;
    VAR save: alset);

VAR
\end{verbatim}
Next, if we have a string w with the first symbol being nonterminal then we shall have procedure fnonterm search the production tree for that nonterminal. If
FIRST_2(A) is already obtained and if string w is of length 1 (that is, w = A) then FIRST_2(w) = FIRST_2(A), otherwise, there are three cases to consider: (1) if v is a member of FIRST_2(A) and length of v is 2 then v is added to FIRST_2(w); (2) if length of v is 1 and v is not empty string then the procedure fterm is called; and (3) if v is the empty string then the second symbol of w is considered. If the second symbol is a terminal then procedure fterm is called, otherwise a recursive call for procedure fnonterm is activated. Procedure fnonterm is described as follows:

PROCEDURE fnonterm(cursor, crt: ptr;
                  VAR save: alset);

VAR
  kursor, kur: ptr;
  kur2: ptr;
BEGIN
  kursor := rrt;
  search(kursor, crt^. entry);
  IF kursor^. allterm = FALSE THEN
    BEGIN
      cursor^. allterm := FALSE;
      done := FALSE
    END
  ELSE
    IF crt^. next = NIL THEN
      BEGIN
        save := save +
               kursor^. second;
        cursor^. second := cursor^. second + kursor^. second
      END
    ELSE
      BEGIN
          33
      END
kur := alpha;
WHILE kur <> NIL DO
BEGIN
  IF kur^.count IN
    kursor^.second
  THEN
    BEGIN
      IF NOT (kur^.count
        count IN termset)
      THEN
        BEGIN
          save:=save +
            [kur^.count];
          cursor^.second:=
            cursor^.second +
            [kur^.count]
        END
      ELSE
        END;
      IF kur^.count<>nullcount THEN
        fterm(cursor,
          crt^.next,
          kur^.entry,
          save)
    END;
    kur := kur^. link
  END;
  IF (nullcount IN kursor^. second)
  THEN
    BEGIN
      kur2 := crt^. next;
      IF kur2^. count IN termset
      THEN
        BEGIN
          IF kur2^. next = NIL
          THEN
            BEGIN
              save := save +
                [kur2^.count];
              cursor^.second:=
                cursor^.second +
                [kur2^.count]
            END
          ELSE
            fterm(cursor,
              kur2^.next,
              kur2^.entry,
              save)
        END
      ELSE
        fnonterm(cursor, kur2, save)
    END
  END
Having discussed procedure \texttt{fterm} and procedure \texttt{fnonterm}, we shall proceed to describe procedure \texttt{transsecond} as follows:

PROCEDURE transsecond(rrt:ptr);

VAR
  cursor: ptr;
  done: BOOLEAN;
  cur: ptr;
  save: alset;

BEGIN (*transsecond*)
  REPEAT
    done := TRUE;
    cursor := rrt;
    WHILE cursor <> NIL DO
      BEGIN
        IF cursor^.allterm = FALSE
         THEN
         BEGIN
           cursor^.allterm := TRUE;
           cur := cursor^.next;
           WHILE cur <> NIL DO
             BEGIN
               save := cur^.firstset;
               IF cur^.secdset = FALSE
               THEN
               BEGIN
               END
             END
           END
         END
        END
      END
  END
END
ELSE
BEGIN
  fnonterm(cursor, cur, save);
  IF cursor\^.
    allterm=TRUE
  THEN
    cur\^.
    secdset := TRUE;
END;
END;
cur\^.
  firstset := save;
cur := cur\^.
  link
END;
cursor := cursor\^.
  link
END
UNTIL done = TRUE
END(*transsecond*);

Where fields "allterm" and "secdset" are used to avoid repeatedly working on those strings whose set "second" have been obtained. And so we can obtain FIRST\_2(w) and FIRST\_2(A) by calling procedure firstnonterm which contains four procedure calls as follows:

Procedure firstnonterm (VAR rt:ptr);
Begin
  buildfirst(rt);
  transfirst(rt);
  buildsecond(rt);
  transsecond(rt);
END;

Using procedure firstnonterm, we can obtain FIRST\_2(w) and FIRST\_2(A) for the example G\_2 (on page 22)
as below:

<table>
<thead>
<tr>
<th>nonterminals</th>
<th>FIRST$_2$(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>{ bb, cb, bc, ab, ac }</td>
</tr>
<tr>
<td>A</td>
<td>{ ^, b, c }</td>
</tr>
</tbody>
</table>

- FIRST$_2$( Abc ) = { bb, cb, bc }
- FIRST$_2$( aAc ) = { ab, ac }
- FIRST$_2$( ^ ) = { ^ }
- FIRST$_2$( b ) = { b }
- FIRST$_2$( c ) = { c }

5.3 FOLLOW set

We shall use procedure follownonterm to find FOLLOW$_2$(A). The procedure follownonterm is as follows:

Procedure follownonterm (VAR rt:ptr);

BEGIN
    buildfollow(rt);
    transfollow(rt);
END;

5.31 Procedure buildfollow

Besides adding "dolarcount", which is the value of $, to the FOLLOW$_2$(S), S being the start symbol of the production, procedure buildfollow scans the production A --> w of the production tree. Once it finds a nonterminal in w, it calls procedure buildfollow (introduced on page 42) to build up the FOLLOW set.
Procedure **buildfollow** is described below:

**PROCEDURE buildfollow**(VAR rrt:ptr);

**VAR**
cursor, cur1, cur2: ptr;
stop: BOOLEAN;

**BEGIN (*buildfollow*)**
rrt\^.
follow := rrt\^.
follow + [dolarcount];
cursor := rrt;
WHILE cursor <> NIL DO
  **BEGIN**
    cur1 := cursor\^.
next;
    **WHILE** cur1 <> NIL DO
      **BEGIN**
        cur2 := cur1;
        **REPEAT**
        stop := FALSE;
        **REPEAT**
          IF (cur2\^.
count IN termset) THEN
            cur2 := cur2\^.
next
          ELSE
            stop := TRUE
          **UNTIL** (stop = TRUE) OR (cur2 = NIL);
        **IF** stop = TRUE **THEN**
          **BEGIN**
            bbbuildfollow(cursor, cur2);
            cur2 := cur2\^.
next;
          **END**
        **UNTIL** (cur2 = NIL);
        cur1 := cur1\^.
link;
      **END**;
    cursor := cursor\^.
link;
  **END**;
**END(*buildfollow*);**

Let us suppose that \(w = vXu\), then \(\text{FOLLOW}_2(X) = \text{FIRST}_2(u \text{FOLLOW}_2(A))\). Procedure **bbbuildfollow** checks for the following three cases: (1) if \(u\) is empty string, that is, \(X\) is the last symbol of the string \(w\) then
"dolarcount" is added to the FOLLOW\textsubscript{2}(X) and if A →→ w is the production where X is found in w then FOLLOW\textsubscript{2}(A) is contained in FOLLOW\textsubscript{2}(X); (2) if a terminal follows X then procedure \texttt{b1buildfollow} is called; and (3) if a nonterminal succeeds X then procedure \texttt{b2buildfollow} is called.

5.311 Procedure \texttt{b1buildfollow}

Let us suppose A →→ w is the production and w = vxau', a being a terminal and u' a string. If u' is the empty string then procedure \texttt{b1buildfollow} sets up a linked list called "flowset", through procedure \texttt{adds}, for X. In other words, the "flowset" for X will be represented as aA →NIL (→ is used to represent the linked tree). The "flowset" will be \texttt{transclosed} later into strings of terminal symbols through the procedure \texttt{transfollow}. Suppose w = uXabv, a and b being terminals and v a string, then the new string ab is added to the FOLLOW\textsubscript{2}(X). However if w = uXav', a being a terminal, Y a nonterminal and v' a string then procedure \texttt{ffadds} is called to add new member to the FOLLOW\textsubscript{2}(X). Procedure \texttt{b1buildfollow} is described below:

PROCEDURE \texttt{b1buildfollow} (cursor,kursor,
kurr:ptr; wd: ALFA);

VAR
  temp: ALFA;
  cur1: ptr;
BEGIN
  temp := blk;
  IF kurr = NIL
  THEN
    BEGIN
      temp[1] := wd[1];
      temp[2] := cursor^. entry[1];
      adds(kursor^. flowset, temp, 2, 0)
    END
  ELSE
    IF (kurr^. count IN termset)
    THEN
      BEGIN
        temp[1] := wd[1];
        temp[2] := kurr^. entry[1];
        fadds(cursor^. next^. llset,
               kursor^. follow, kk, temp)
      END
    ELSE
      BEGIN
        cur1 := rrt;
        search(cur1, kurr^. entry);
        ffadds(cur1^. second,
               cursor^. next^. llset,
               kursor^. follow, wd);
        IF (nullcount IN cur1^. second)
        THEN
          b1buildfollow(cursor,
                        kursor, kurr^. next, wd);
      END;
  END (*b1buildfollow*);

where "llset" used in this procedure is a dummy field.

5.312 Procedure b2buildfollow

Suppose $A \rightarrow w$ is the production and $w = \alpha XYu$, $Y$
being a nonterminal and $u$ a string. Procedure
b2buildfollow searches the production tree to find \( \text{FIRST}_2(Y) \). Suppose \( v \in \text{FIRST}_2(Y) \). If \( \text{length}(v) = 2 \) then the value of \( v \) is added to the \( \text{FOLLOW}_2(X) \) and if \( \text{length}(v) = 1 \) and \( v \) is not the empty string then procedure b1buildfollow is called. However, if \( v \) is the empty string then \( \text{FOLLOW}_2(X) = \text{FIRST}_2(u \ \text{FOLLOW } (A)) \).

Procedure b2buildfollow is as follows:

```plaintext
PROCEDURE b2buildfollow(cursor, kursor, kurr:ptr);

VAR
    cur1, cur2: ptr;
    temp: ALFA;

BEGIN
    temp := blk;
    cur1 := rrt;
    search(cur1, kurr\. entry);
    cur2 := alpha;
    WHILE cur2 <> NIL DO
        BEGIN
            IF (cur2\. count IN cur1\. second) THEN
                BEGIN
                    IF NOT (cur2\. count IN termset) THEN
                        kursor\. follow:=kursor\. follow + [cur2\. count]
                    ELSE
                        IF (cur2\. count <> nullcount) THEN
                            b1buildfollow(cursor, kursor, kurr\. next, cur2\. entry)
                END;
                cur2 := cur2\. link
            END;
    IF (nullcount IN cur1\. second) THEN
        BEGIN
            IF kurr\. next = NIL THEN
            END;

41
BEGIN
Kursor^\'.follow:=Kursor^\'.follow
+ [dolarcount];
adds(Kursor^\'.flowset,
    cursor^\'.entry, 1,0)
END
ELSE
BEGIN
kurr := kurr^\'.next;
IF (kurr^\'.count IN termset)
THEN
    b1buildfollow(cursor,
        kursor,kurr^\'.next,
        kurr^\'.entry)
ELSE
    b2buildfollow(cursor,
        kursor,kurr)
END;
END (*b2buildfollow*);

So, procedure bbuildfollow looks like this:

PROCEDURE bbuildfollow(cursor,crt:ptr);
VAR
    kursor, kur: ptr;
BEGIN (*bbuildfollow*)
kursor := rrt;
search(kursor, crt^\'.entry);
kur := crt^\'.next;
IF kur = NIL
THEN
    BEGIN
        adds(kursor^\'.flowset,cursor^\'.entry,
            1,0);
        kursor^\'.follow:=kursor^\'.follow
            + [dolarcount]
    END
ELSE
    IF (kur^\'.count IN termset)
    THEN
        b1buildfollow(cursor,kursor,
            kur^\'.next,kur^\'.entry)
    ELSE
        b2buildfollow(cursor, kursor, kur)
    END (*bbuildfollow*);
5.32 Procedure transfollow

After the linked list "flowset" has been set up for the nonterminal X, we shall use procedure transfollow to add terminal strings to the FOLLOW\(_2\)(X). There are two cases to consider: (1) if the symbol of the "flowset" is a nonterminal (of length 1) then procedure t1transfollow is to be called; otherwise, (2) procedure t2transfollow is to be executed. Procedure transfollow is described as follows:

```plaintext
PROCEDURE transfollow(VAR rrt: ptr);
VAR
cursor, kursor: ptr;
done, stop: BOOLEAN;

PROCEDURE t1transfollow(crt:ptr;VAR stop:BOOLEAN);
VAR
kursor: ptr;
BEGIN
WITH crt^ DO
BEGIN
IF entry <> flowset^. entry THEN
BEGIN
kursor:=rrt;
search(kursor,flowset^.entry);
IF (kursor^.flowset = NIL) THEN
BEGIN
follow:=follow +
```
PROCEDURE t2transfollow(crt:ptr;  
VAR stop: BOOLEAN);  
VAR  
kursor: ptr;  
temp: ALFA;  
BEGIN  
WITH crt^ DO  
BEGIN  
temp := blk;  
temp[1] := flowset^. entry[2];  
kursor := rrt;  
search(kursor, temp);  
IF kursor^. flowset = NIL  
THEN  
BEGIN  
ffadds(kursor^. follow, crt^. next^. llset, crt^. follow, temp);  
flowset := flowset^. link  
END  
ELSE  
BEGIN  
done := FALSE;  
stop := TRUE;  
END;  
END;  
END (*t2transfollow*);  
BEGIN (*transfollow*)  
REPEAT  
done := TRUE;
cursor := rrt;
WHILE cursor <> NIL DO
  WITH cursor DO
    BEGIN
      IF flowset <> NIL THEN
        BEGIN
          stop := FALSE;
          REPEAT
            IF flowset length = 1 THEN
              t1transfollow(cursor, stop)
            ELSE
              t2transfollow(cursor, stop)
            UNTIL (flowset=NIL) OR (stop = TRUE)
          END;
          cursor := cursor'. link
        END
      END
    END
UNTIL (done = TRUE)
END(*transfollow*);

If we apply procedure follownonterm to obtain the FOLLOW set, using $G_2$ (on page 22) as an example then the FOLLOW set for each of the nonterminals is listed in the following table:

<table>
<thead>
<tr>
<th>nonterminals</th>
<th>FOLLOW$_2$(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>${$}$</td>
</tr>
<tr>
<td>$A$</td>
<td>${bc, cb}$</td>
</tr>
</tbody>
</table>

After we have obtained the FIRST and the FOLLOW sets, we shall be able to check LL(k) conditions and to generate the LL(k) parsing table.

5.4 Testing LL(k) conditions
If generalized productions are of the form $A \rightarrow w_1 \mid w_2 \mid w_3 \mid \ldots \mid w_n$ then the grammar is LL(k) if and only if strings $w_i$ are pair wise disjoint. To implement this concept with $k = 2$, we introduce "llset", which is $\text{FIRST}_2(w)$ if the empty string is not in the $\text{FIRST}_2(w)$; otherwise "llset" is the union of two sets: $\text{FIRST}_2(w)$ and $\text{FOLLOW}_2(A)$. The procedure `checkll` is described below:

PROCEDURE checkll(rt: ptr);
VAR
cursor, cur: ptr;
hold, temp: alset;
BEGIN
cursor := rt;
WHILE cursor <> NIL DO
BEGIN
cur := cursor\^'. next;
hold := cur\^'. llset;
WHILE cur <> NIL DO
BEGIN
IF cur\^'. link <> NIL THEN
BEGIN
    temp:=cur\^'. link\^'. llset;
    IF hold * temp = []
    THEN
        hold := hold + temp
    ELSE
        error(2)
    END;
cur := cur\^'. link
END;
cursor := cursor\^'. link
END;
WRITELN(OUTPUT, '** This is an LL(k)
grammar**');
END (*checkll*);
Here error(2) lists the error message if the strings are not pairwise disjoint.

5.5 Generating the LL(k) parsing table

Procedure llktable contains mainly two procedures: procedure ltabl and procedure ltab2. Having obtained the "llset" for each of the production string w (suppose production is A → w), we will use procedure ltabl to set up the matrix tree, M. For each string u contained in "llset" of w, M(A, u) = i, A being the nonterminal and i the production number. Because of the restriction of the space, we shall not list production string w in the matrix M. Later when parsing the input string we shall use procedure ssearch to look up the production tree for w. For those strings v not contained in "llset" of w, we set M(A, v) = 0, 0 being designated "error". Procedure ltab2 expands the matrix tree M; and M(a, au) = -1; a is in Σ; -1 is designated "pop"; finally M($, $) = -2; -2 is designated "accept"; otherwise M(a, v) = 0 for "error". If we again use the LL(2) grammar, G2 described on page 22 as an example
then the parsing table is illustrated below:

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>bb</th>
<th>cb</th>
<th>bc</th>
<th>ab</th>
<th>ac</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>err</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>5</td>
<td>err</td>
<td>err</td>
<td>2</td>
<td>2</td>
<td>err</td>
<td>err</td>
<td>err</td>
</tr>
<tr>
<td>b</td>
<td>pop</td>
<td>err</td>
<td>pop</td>
<td>err</td>
<td>pop</td>
<td>err</td>
<td>err</td>
<td>pop</td>
<td>err</td>
</tr>
<tr>
<td>c</td>
<td>err</td>
<td>pop</td>
<td>err</td>
<td>pop</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>pop</td>
<td>err</td>
</tr>
<tr>
<td>a</td>
<td>err</td>
<td>err</td>
<td>pop</td>
<td>err</td>
<td>err</td>
<td>pop</td>
<td>pop</td>
<td>err</td>
<td>pop</td>
</tr>
<tr>
<td>$</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>err</td>
<td>accept</td>
<td>err</td>
</tr>
</tbody>
</table>

where "err" stands for "error".

5.6 Parsing the LL(k) grammars

First, we use procedure `getstring` to set up input string called "istring", with $ attached to the end of the "istring", in the meantime, checking to see if the symbols read from the input file belong to the alphabet of the grammar. An error message is repored if a symbol not belonging to the grammar is detected. Next, we build the initial pushdown list ("stackstring"), which is S$, S being the start symbol of the grammar. To parse the "istring", procedure `parse` has to consider the following cases: (1) if the top of the "stackstring" is the same as that of the "istring" and the symbol is not
$, then, the top symbols of both strings are popped out, or, (2) if the top of the "stackstring" is the empty string then the empty string is removed, otherwise, (3) it searches the parsing table for \( M(A, u) \), \( A \) being the top of the pushdown list, \( u \) the first 2 symbols of the "istring". If field "num" = 0 then the parsing ceases and an error message is reported and if "num" = -2 then the message "accept" is reported and the parsing is completed; otherwise, the procedure `pparse` is called to search the production tree for the production number which is equal to the value of "num" so that the top of the pushdown list can be replaced by the production and the production number is added to the end of the production number string called "numstring". Procedure `pparse` is listed below:

```plaintext
PROCEDURE pparse(VAR sstack: lptr; VAR nnumstring: nptr; rrt: ptr; i: INTEGER);
VAR
    cursor, cur1: ptr;
    temp, kursor: lptr;
BEGIN
    cursor := rrt;
    search(cursor, sstack^.entry);
    cur1 := cursor^.next;
    WHILE cur1^.num <> i DO
        cur1 := cur1^.link;
        temp := NIL;
    WHILE cur1 <> NIL DO
        BEGIN
            adds2(temp, cur1^.entry,
                  cur1^.length);
    ```
cur1 := cur1^. next
END;
kursor := temp;
WHILE kursor^. link <> NIL DO
  kursor := kursor^. link;
  kursor^. link := sstack^. link;
  sstack := temp;
  adds3(nnumstring, i);
END (*pparse*);

Where procedure **adds2** adds symbol to the strings
and procedure **adds3** performs the task of expanding the
"numstring".

Procedure **parse** is as follows:

PROCEDURE parse (VAR lltab,iistring,sstack: lptr;VAR nnumstring:nptr;rt:ptr);
VAR
  stop: BOOLEAN;
  temp: ALFA;
  i: INTEGER;
BEGIN(*parse*)
  adds2(sstack, lltab^. entry, lltab^. length);
  adds2(sstack, dollarsym, 1);
  stop := FALSE;
  REPEAT
    printstring(iistring, sstack, nnumstring);
    IF sstack^. entry = null
    THEN
      sstack := sstack^. link
    ELSE
      IF (sstack^. entry= iistring^. entry)
         AND (sstack^. entry<> dollarsym)
    THEN
      BEGIN
        iistring := iistring^. link;
        sstack := sstack^. link
      END
    ELSE
      BEGIN
        50
temp := blk;
temp[1] := iistring^.entry[1];
IF iistring^.link <> NIL THEN
  IF iistring^.link^.entry <>
    dollarsym THEN
    temp[2] := iistring^.link^.entry[1];
ssearch(lltab, sstack^.entry, temp, i);
  IF (i = -2) OR (i = 0) THEN
    BEGIN
      stop := TRUE;
      IF i = -2 THEN
        WRITELN(OUTPUT, 'Accept ')
      ELSE
        WRITELN(OUTPUT, 'Error ');
      END
    ELSE
      ELSE
      pparse(sstack, nnumstring, rt, i)
    END
  END
UNTIL (stop = TRUE)
END(*parse*);

where, procedure printstring prints out the current
"istring" followed by "stackstring" followed by
"numstring".

If we use the LL(2) grammar G2 (on page 22) to parse
an input string acb then the parsing sequences are
illustrated as follows:

```
a c b $  S $  
a c b $  a A c b $  4
b $  A c b $  4
c b $  A c b $  4 2
b $  c b $  4 2
b $  4 2
$c  4 2
```
accept
6. Conclusions

The $k$ predictive parsing algorithm we have analyzed is effective in constructing a deterministic left parser for a restrictive class of context free grammars. It is apparent from the algorithm that if a grammar fails to generate the parsing table then it is not LL($k$). If the table generation succeeds, then the grammar is LL($k$) and we will not only have a definition of a deterministic parser, but will know that the grammar is unambiguous. However, the implementation of the algorithm becomes laborious when $k$ becomes greater than two and the production rules are numerous.

The complete computer program developed to implement the parsing algorithm was written in the language PASCAL. It is filed with Professor Samuel L. Gulden at the Division of Computing and Information Science, Lehigh University, Bethlehem, Pennsylvania.
7. Bibliography


8. Vita

The author was born to Mr. and Mrs. I. S. Liu on November 20, 1942 in Tainan, Taiwan, Republic of China. She was graduated from National Taiwan University in 1964 with a B. S. degree in Psychology. She had worked for four years as a librarian at Waynesburg College, Waynesburg, Pennsylvania after receiving a master degree in Library and Information Science from the University of Pittsburgh, Pittsburgh, Pennsylvania in 1968. In 1979 she began the graduate program in Computing Science at Lehigh University. She was a teaching assistant in mathematics for the Learning Center at Lehigh University.