df-06199

UNIVERSITY OF MARYLAND

COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

08

Technical Report 68-76 July, 1968

On the Parsing of Context-Free Languages by Pushdown Automata

by

Victor B. Schneider
Computer Science Center
University of Maryland
College Park, Maryland 20742

The research for this paper was supported in part by N.A.S.A. grant (NsG-398) to the Computer Science Center of the University of Maryland.
NGL-21-002-008

Table of Contents

Illustrations and Tables

Notation and Basic Definitions

Let V be a finite, nonempty set of symbols, which we will call the vocabulary. Elements of V are denoted by letters, such as d,e,f,G,H,I, etc. Finite sequences of symbols, including the empty sequence {e}, are called strings and are denoted by late small letters, such as x,y,z, etc. The set of all strings over a set such as V is denoted by V*.

A context-free grammar (abbreviated CFG) is an ordered four-tuple

$$G = (V, T, P, S)$$

where

(a) V is a vocabulary of symbols.

(b) T is a proper subset of V called the terminals.

(c) P is a finite, nonempty set of syntactic rules P_i of the form $U \rightarrow x$, where $U \neq x$, U is in V-T, and x is in V*-{e}. For a rule $P_i = U \rightarrow x$, U is called the left part and x the right part of P_i.

(d) S is a special symbol in V-T, the initial symbol.

As is usual, we say that x directly produces y. $(x \Rightarrow y)$, and conversely y directly reduces to x if and only if there exist strings U, such that $x = uZv$ and $y = uwv$ and

$$Z \rightarrow w \text{ is in } P.$$

x produces y $(x \overset{*}{\Rightarrow} y)$, and conversely y reduces to x if and only if either

$$x = y$$

or there exists a sequence of nonempty strings $w_0, w_1, \ldots, w_n$ such that $x = w_0$ and $y = w_n$ and

$$w_i \Rightarrow w_{i+1} \; (i = 0, 1, \ldots n-1 \text{ and } n \geq 1).$$

x is a sentence of G if x is in $T^* - \{e\}$ and S produces x.

A context-free language (abbreviated CFL) is then the set of strings x that can be produced by grammar G from its initial symbol S:

$$L(G) = \{x: (S \overset{*}{\Rightarrow} x) \ \& \ (x \in T^* - \{e\})\}$$

Let S produce x. A <u>parse</u> of the string x into the symbol S is a sequence of rules $P_1, \ldots P_n$ such that P_j directly reduces w_{j-1} into $w_j (j=1,\ldots,n)$ and $x = w_o$, $S = w_n$.

Let $x = a_1, \ldots a_r$ be a string of symbols a_i in T. Then, in some reduction sequence in which $x = w_o$, let x reduce to $w_j = ua_k, \ldots a_r$, with u in V^* and $1 \leq k \leq r$. If P_j directly reduces string w_j into w_{j+1} and P_{j1} directly reduces w_{j+1} into w_{j+2}, then P_j, P_{j1} is called a <u>leftmost reduction sequence</u> if

$$w_{j+1} = u' \ a_{k'} \ldots a_r \quad u' \text{ in } V^*$$

and $$w_{j+2} = u'' \ a_{k''} \ldots a_r \quad u'' \text{ in } V^*$$

and $$k \leq k' \leq k'' \leq r.$$

A parse $P_1, \ldots, P_n$ is called a <u>leftmost parse</u> if and only if the sequences P_i, P_{i+1} are leftmost reduction sequences for $i = 1, \ldots, n-1$.

If $P_1, \ldots, P_n$ is a parse of string x into symbol S, there exists a permutation of $P_1, \ldots, P_n$ that is leftmost. We define an <u>unambiguous</u> grammar G to be one in which every x in L(G) has exactly one leftmost parse.

We next define a normal form for CFG's, in terms of which a leftmost parsing algorithm can be designed. The correspondence between this leftmost parsing algorithm and a pushdown automaton model to be introduced will then become apparent. Subsequently, an algorithm for facilitating single-scan leftmost parsing in a large class of grammars will be developed.

Normal Form Grammars

A grammar G = (V, T, P, S) will be said to be in normal form if all the rules in P are of the forms

$$A_i \to A_{i1}\ A_{i2} \qquad \text{or } A_j \to A_{j1}$$
$$\text{or } A_k \to A_{k1}\ a_{k2} \qquad \text{or } A_m \to a_{m1}$$

with A_{i1}, A_{i2}, A_{j1}, A_{k1} in V-T and a_{k2}, a_{m1} in T. A very simple algorithm exists for converting any grammar H into a grammar H' such that L(H) = L(H'). Because of this algorithm, all derivations of sentences in L(H) are in one-to-one correspondence with derivations of sentences in L(H'). The algorithm works as follows:

All productions in P of H that are already in normal form are taken into P' of H'. The remaining productions in P are of the form

$$X \to X_1 \ldots X_n \qquad , \quad (n > 2)\ \&\ (X_i \in V).$$

Each production of this form is transferred to P' as a sequence of productions.

$$J_v \to J_{v-1}\ X_{v+1} \qquad \text{for } v = 1, \ldots, n-1$$

where J_{n-1} is X. J_0 is X_1 if X_1 is in V-T of H; otherwise an additional rule of the form

$$J_0 \to X_1$$

is included in P'. The J_v are treated as new elements in V'-T' of H', and the J_v are distinct from the elements in V-T of H.

The fact that the J_v of the algorithm are "new and distinct" leads to a simple proof of the one-to-one correspondence between derivations of sentences in L(H) and L(H'): Since each rule of P corresponds to a particular rule or sequence of rules in P', it follows that, for each derivation possible in H, there is a corresponding derivation in H', and conversely. Because of this unique correspondence, it also follows that ambiguity in L(H) is equivalent to ambiguity in L(H').

Leftmost Parses and Normal-Form Grammars

In order to describe the algorithm for producing leftmost parses of the sentences of a grammar G in normal form, we introduce boundary markers # to the vocabulary of G. A new initial symbol S' now takes the place of S in G, and three new rules are added to G:

$$P_1' = S' \rightarrow J_1 \, \#$$

$$P_2' = J_1 \rightarrow J_2 S \qquad P_3' = J_2 \rightarrow \#$$

This has the effect of putting boundary markers at both ends of all strings produced by the grammar.

Let $w_0 = \#a_1 \ldots a_n\#$ be a string in the language of such a grammar. In the initial step of the leftmost parsing algorithm, rule P_3' is applied, yielding string

$$w_1 = J_2 \, a_1 \ldots a_n \, \#$$

After i steps, w_0 has been reduced to

$$w_i = J_2 \, K_1 \ldots K_r \, a_s \ldots a_n \, \# \quad (1 \leq r < s \leq n+1).$$

In this configuration, $K_1, \ldots, K_r$ are all symbols of V-T in the grammar. If w_0 is in L(G), the leftmost sequence of rules $P_3' = P_1, \ldots, P_j$ are precisely the first j reductions of the leftmost parse of w_0 to S'.

For the (j+1)-th reduction, five different cases must be distinguished:

(0) S' does not produce w_i, where $w_i = J_2 \, K_1 \ldots K_r \, a_s \ldots a_n \, \#$.

If S' does produce w_i, we have to distinguish between the following possibilities:

(1) A rule of the form $P_{j+1} = K'_{r+1} \rightarrow a_s$ reduces w_i to w_{i+1}.

(2) A rule of the form $P_{j+1} = K'_r \rightarrow K_r$ reduces w_i to w_{i+1}.

(3) A rule of the form $P_{j+1} = K'_{r-1} \rightarrow K_{r-1}\ K_r$ reduces w_i to w_{i+1}.

(4) A rule of the form $P_{j+1} = K'_r \rightarrow K_r\ a_s$ reduces w_i to w_{i+1}.

That only these cases need be considered is proved in [10].

In general, the decision concerning which of the cases (1) to (4) apply for the (j+1)-th step of a leftmost parse must be made in terms of <u>context</u>. As an example, there may exist rules in the grammar having $K_{r-1}\ K_r$ <u>and</u> $K_r\ a_s$ on the right part. To decide which case applies at a given step of the parse then requires knowledge of what symbols can be adjacent to the symbols being reduced in that step while w_0 is a sentence of G.

<u>Case (1)</u>

For a rule of the form $P_{j+1} = K'_{r+1} \rightarrow a_s$ to apply for the (j+1)-th reduction, there must be one or more symbol Z in V-T such that $Z \rightarrow K_r\ Y$ is in P and $Y \overset{*}{\Rightarrow} K'_{r+1}\ u$, with u in V*.

Since the set $\{K: V \overset{*}{\Rightarrow} K\ u\ \&\ u \in V^*\}$ can be constructed, the context in which Case (1) applies can be found. The pairs (K_r, a_s) are the contexts in which the rule $K'_{r+1} \rightarrow a_s$ applies.

<u>Case (2)</u>

For a rule of the form $P_{j+1} = K'_r \rightarrow K_r$ to apply for the (j+1)-th reduction, there must be one or more symbols Z in V-T such that either

	$Z \rightarrow K_{r-1}\ Y$	is in P
and	$Y \overset{*}{\Rightarrow} X_u$,	with u in V*.
and	$X \rightarrow RT$	where $R \overset{*}{\Rightarrow} K'_r$
and	$T \overset{*}{\Rightarrow} a_s y$,	with y in V*.
or	$Z \rightarrow Ya_s$	is in P
and	$Y \overset{*}{\Rightarrow} uX$	with u in V*
and	$X \rightarrow K_{r-1}R$	
where	$R \overset{*}{\Rightarrow} K'_r$.	

The pairs (K_{r-1}, a_s) are the contexts in which the rule $K'_r \rightarrow K_r$ applies.

Case (3)

For a rule of the form $P_{j+1} = K'_{r-1} \rightarrow K_{r-1}\ K_r$ to apply for the (j+1)-th reduction, there must be one or more symbols Z in V-T such that

$$Z \rightarrow YX \text{ is in } P$$

and $\quad X \overset{*}{\Rightarrow} a_s u$, with u in V*.

and $\quad Y \overset{*}{\Rightarrow} w\ K'_{r-1}$ with w in V*

The pairs (K_{r-1}, a_s) are the contexts in which the rule $K'_{r-1} \rightarrow K_{r-1}\ K_r$ applies.

Case (4)

For a rule of the form $P_{j+1} = K'_r \rightarrow K_r\ a_s$ to apply for the (j+1)-th reduction, there must be one or more symbols Z in V-T such that

$$Z \rightarrow K_{r-1}\ Y \qquad \text{is in } P$$

and $\quad Y \overset{*}{\Rightarrow} K'_r\ u \qquad$ with u in V*.

The pairs (K_{r-1}, a_s) are the contexts in which rule $K'_r \rightarrow K_r\ a_s$ applies.

After the contexts for which cases (1) - (4) apply have been determined, there may in general still exist rules having the same contexts. The existence of such rules in a grammar may imply the necessity of backtracking methods for use in parsing a given string of that grammar. Or, such a grammar may be ambiguous. In the following section, we sketch a formal model for this normal-form leftmost parsing algorithm. This model is a <u>pushdown automaton</u> (abbreviated PDA) having a single pushdown store, or <u>stack</u>. In terms of this model, we can present an algorithm for eliminating the necessity of backtracking in a large class of unambiguous CFL's.

Pushdown Automaton Parsing Model

A pushdown automaton acceptor A is defined to be an eight-tuple

$$A = (Q, T, N, D, M, \#, S_0, F)$$

such that

(a) Q is a finite set, called the states of the machine.

(b) T is a finite set of symbols, called the input-tape vocabulary.

(c) N is a finite set of symbols, called the pushdown-store vocabulary.

(d) $D = \{1, 2, 3\}$ is called the instruction set.

(e) M is a mapping of $Q \times (T \cup \{e\}) \times (N \cup \{e\})$ into the finite subsets of $Q \times (N \cup \{e\}) \times D$.

(f) # is a special symbol such that

$$\# = T \cap N.$$

(g) S_0 is the initial state of a computation and F is called the final state.

By analogy to our notation regarding CFL's, we define an initial configuration of a computation to be $C_0 = (\# S_0 x \#)$, where x is the input string to be accepted. The final configuration is $(\# F \#)$.

The computation of the machine is essentially a reduction sequence that reduces C_0 to the final configuration. Let C_j and C_{j+1} be two configurations of a computation. Then, C_j directly reduces to C_{j+1} $(C_j \vdash C_{j+1})$ if $C_j = (\# t Z S_1 a w \#)$ and $C_{j+1} = (\# t y S_2 b w \#)$ and (S_2, Y, d) is in $M(S_1, a, Z)$.

(a) If $d = 1$, $(b = a)$ & $(y = Y)$. I.e., Z is a symbol erased from the stack and replaced by symbol Y. Here, the strings # t Z and # t y represent the contents of the stack, and Y may be e, the empty symbol.

(b) If $d = 2$, $(b = e)$ & $(y = ZY)$. I.e., symbol Z is not erased, and Y is written to the right of Z. Also, a is erased from the input string (it is replaced by e). Here, the strings a w # and b w # represent the portions of the input string that remain to be reduced by the computation.

(c) If $d = 3$, $(b = a)$ & $(y = ZY)$. I.e., no erasures occur.

Going a step further, we let

$$C_1 = (\# \; u \; S_1 \; a_1 \; \ldots a_k \; v \; \#)$$

and

$$C_2 = (\# \; u' \; S_{j+1} \; v \; \#)$$

be configurations of some computation. Then C_1 <u>reduces</u> to C_2 ($C_1 \overset{*}{\vdash} C_2$) if there exists a sequence of configurations $H_o, H_1, \ldots, H_k$, with $C_1 = H_o$ and $C_2 = H_k$ and

$$H_i \vdash H_{i+1} \qquad (i = 0, 1, \ldots, k-1).$$

Then, the language accepted by an automaton A is the set of input strings x given by

$$L(A) = \{x : [(\# \; S_o \; x \; \#) \overset{*}{\vdash} (\# \; F \; \#)] \; \& \; (x \in T^* - \{e\})\} \quad .$$

<u>Automaton Realization of Leftmost Parses</u>

With the PDA model as defined above, it is possible to introduce a correspondence between rules of a normal-form grammar and the states and symbols of a PDA. For all rules in the grammar of the form

$$A_i \rightarrow A_{i1} \; A_{i2} \quad \text{and} \quad A_k \rightarrow A_{k1} \; a_{k2},$$

the A_{i2}'s and A_{k1}'s become states of the automaton. The A_{i1}'s become members of N, the stack vocabulary, and the a_{k2}'s become members of T, the input-tape vocabulary. Note that, in this correspondence, the initial symbol S of a grammar becomes the final state F of the PDA. What follows is the algorithm for constructing a PDA from the rules of a normal-form grammar:

Rule $A_i \rightarrow A_{i1}\ A_{i2}$ with contexts (A_{i1}, a_s):

If A_i is in N,

$(S_o, A_i, 1)$ is in $M(A_{i2}, a_s, A_{i1})$.

If A_i is in Q,

$(A_i, e, 1)$ is in $M(A_{i2}, a_s, A_{i1})$.

These transitions take care of all possibilities arising from case (3) of the leftmost parsing algorithm. If A_i is in N, that means that a pair $A_i\ A_{j2}$ appears as the right part of some rule of the grammar. Hence, A_i is placed "on top" of the stack and the automaton is placed in the initial position for discovering A_{j2}. If A_i is in Q, then A_i is either the second nonterminal of some rule in the grammar or the first nonterminal in some rule of the form $A_k \rightarrow A_i\ a_{k2}$.

Rule $A_k \rightarrow A_{k1}\ a_{k2}$ with contexts (K_{r-1}, a_{k2})

If A_k is in N,

$(S_o, A_k, 2)$ is in $M(A_{ki}, a_{k2}, K_{r-1})$.

If A_k is in Q,

$(A_k, e, 2)$ is in $M(A_{ki}, a_{k2}, K_{r-1})$.

These transitions take care of all possibilities arising from case (4) of the leftmost parsing algorithm.

Rule $A_j \rightarrow a_{j1}$ with contexts (K_r, a_s)

If A_j is in N,

$(S_o, A_j, 2)$ is in $M(S_o, a_s, K_r)$.

If A_j is in Q,

$(A_j, e, 2)$ is in $M(S_o, a_s, K_r)$.

These transitions take care of all possibilities arising from Case (1) of the leftmost parsing algorithm.

Rule $A_j \to A_{j1}$ with contexts (K_{r-1}, a_s)

For every chain of rules in the grammar of the form $P_1 = A \to A^{(1)}$, $P_2 = A^{(1)} \to A^{(2)}, \ldots, P_n = A^{(n-1)} \to A^{(n)}$ with $n \geq 2$, and such that there is at least one context (K'_{r-1}, a'_s) common to rules $P_1, \ldots, P_n$, we introduce transitions of the form

$$(A^{(n-1)}, e, 3) \text{ is in } M(A^{(n)}, a'_s, K'_{r-1})$$
$$\vdots$$
$$(A^{(1)}, e, 3) \text{ is in } M(A^{(2)}, a'_s, K'_{r-1})$$

These $A^{(1)}, \ldots, A^{(n)}$ are thus treated as states of the PDA.

For all additional contexts (K_{r-1}, a_s) associated with individual rules $A_j \to A_{j1}$, we have the following:

If A_j is in N,

$(S_o, A_j, 3)$ is in $M(A_{j1}, a_s, K_{r-1})$.

If A_j is in Q,

$(A_j, e, 3)$ is in $M(A_{j1}, a_s, K_{r-1})$.

These transitions take care of all possibilities arising from case (2) of the leftmost parsing algorithm.

When all transitions of a machine have been defined as described above, there results a PDA whose language is the language of the grammar from which it is constructed.

A Simple Programming Language Translator

The following is a simplified grammar for a computer programming language having nested block structure, conditional statements, and arithmetic assignment statements. The ALGOL conventions are used for representing symbols of the grammar; i.e., members of V-T are enclosed by "⟨" "⟩" and members of T are not. The symbol "|" is a separator that allows two or more rules having the same left part to be written together:

```
G:   ⟨program⟩ → ⟨body⟩ ⟨stat⟩ end
        ⟨body⟩ → begin | ⟨body⟩ ⟨stat⟩ ;
        ⟨stat⟩ → ⟨program⟩ | ⟨assignment⟩
        ⟨assignment⟩ → ⟨var⟩:= ⟨expr⟩
        ⟨expr⟩ → ⟨simple expr⟩ | ⟨if clause⟩
                  ⟨simple expr⟩ else ⟨expr⟩
        ⟨simple expr⟩ → ⟨term⟩ | ⟨simple expr⟩ + ⟨term⟩
        ⟨term⟩ → ⟨factor⟩ | ⟨term⟩* ⟨factor⟩
        ⟨factor⟩ → ⟨var⟩ | ⟨number⟩ | [ ⟨expr⟩]
        ⟨if clause⟩ → if ⟨relation⟩ then
          ⟨relation⟩ → ⟨simple expr⟩ = ⟨simple expr⟩
        ⟨var⟩ → A|B|C|...|Z
        ⟨number⟩ → ⟨digit⟩ | ⟨number⟩⟨digit⟩
        ⟨digit⟩ → 0|1|...|9
```

The programming language G easily reduces to the following normal-form grammar G', augmented by the addition of endmarkers:

$G'\colon S \rightarrow Y_1 \; \#$

$Y_1 \rightarrow Y_2 \; \langle \text{program} \rangle$

$Y_2 \rightarrow \#$

$\langle \text{program} \rangle \rightarrow X_1 \; \underline{\text{end}}$

$X_1 \rightarrow \langle \text{body} \rangle \; \langle \text{stat} \rangle$

$\langle \text{body} \rangle \rightarrow \underline{\textbf{begin}} \mid X_2;$

$X_2 \rightarrow \langle \text{body} \rangle \langle \text{stat} \rangle$

$\langle \text{stat} \rangle \rightarrow \langle \text{program} \rangle \mid \langle \text{assignment} \rangle$

$\langle \text{assignment} \rangle \rightarrow X_3 \; \langle \text{expr} \rangle$

$X_3 \rightarrow \langle \text{var} \rangle :=$

$\langle \text{expr} \rangle \rightarrow \langle \text{simple expr} \rangle \mid X_4 \; \langle \text{expr} \rangle$

$X_4 \rightarrow X_5 \; \underline{\text{else}}$

$X_5 \rightarrow \langle \text{if clause} \rangle \; \langle \text{simple expr} \rangle$

$\langle \text{simple expr} \rangle \rightarrow \langle \text{term} \rangle \mid X_6 \; \langle \text{term} \rangle$

$X_6 \rightarrow \langle \text{simple expr} \rangle \; +$

$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \mid X_7 \; \langle \text{factor} \rangle$

$X_7 \rightarrow \langle \text{term} \rangle *$

$\langle \text{factor} \rangle \rightarrow \langle \text{var} \rangle \mid \langle \text{number} \rangle \mid X_8]$

$X_8 \rightarrow X_9 \; \langle \text{expr} \rangle$

$X_9 \rightarrow [$

$\langle \text{if clause} \rangle \rightarrow X_{10} \; \underline{\text{then}}$

$X_{10} \rightarrow X_{11} \; \langle \text{relation} \rangle$

$X_{11} \rightarrow \underline{\text{if}}$

$\langle \text{relation} \rangle \rightarrow X_{12} \; \langle \text{simple expr} \rangle$

$X_{12} \rightarrow \langle \text{simple expr} \rangle =$

⟨var⟩ → A|B|C|...|Z

⟨number⟩ → ⟨digit⟩ | ⟨number⟩⟨digit⟩

⟨digit⟩ → 0|1|...|9

What follows is a table of contexts in which the rules of G' can be applied during a leftmost parse of some string in L(G'):

Rule	Contexts
S → Y_1 #	(e, #)
Y_1 → Y_2 program	(Y_2, #)
Y_2 → #	(e, #)
⟨program⟩ → X_1 end	(Y_2, end), (⟨body⟩, end)
X_1 → ⟨body⟩ ⟨stat⟩	(⟨body⟩, end)
⟨body⟩ → begin	(Y_2, begin), (⟨body⟩, begin)
⟨body⟩ → X_2;	(Y_2, ;), (⟨body⟩, ;)
X_2 → ⟨body⟩⟨stat⟩	(⟨body⟩, ;)
⟨stat⟩ → ⟨program⟩	(⟨body⟩, end), (⟨body⟩, ;)
⟨stat⟩ → ⟨assignment⟩	(⟨body⟩, end), (⟨body⟩, ;)
⟨assignment⟩ → X_3 ⟨expr⟩	(X_3, end), (X_3, ;)
X_3 → ⟨var⟩: =	(⟨body⟩, : =)
⟨expr⟩ → ⟨simple expr⟩	(X_3, ;), (X_3, end), (X_9,])
⟨expr⟩ → X_4 ⟨expr⟩	(X_4, end), (X_4, ;)
X_4 → X_5 else	(X_3, else), (X_9, else)
X_5 → ⟨if clause⟩⟨simple expr⟩	(⟨if clause⟩, else)
⟨simple expr⟩ → ⟨term⟩	(⟨if clause⟩, else), (X_3, ;), (X_3, end), (X_9,]), (⟨if clause⟩, +), (X_4, +), (X_3, +), (X_9, +), (X_{11}, =)

⟨simple expr⟩ → X_6 ⟨term⟩	(X_6, +), (X_6, else), (X_6, ;), (X_6, end), (X_6, =), (X_6, then), (X_6,])
X_6 → ⟨simple expr⟩ +	(⟨if clause⟩ , +), (X_4, +), (X_3, +), (X_9, +), (X_{12}, +), (X_{11}, +)
⟨term⟩ → ⟨ factor⟩	(X_6, +), (X_6, =), (X_6, then), (X_6, else) (X_6, *), (⟨ if clause ⟩ , *), (X_{12}, *), (X_4, *), (X_9, *), (X_3, *), (⟨if clause ⟩, else), (X_3, ;), (X_3, end), (X_9,]), (⟨ if clause ⟩ , +), (X_4, +), (X_3, +), (X_9, +), (X_{11}, =)
⟨term ⟩→ X_7 ⟨factor ⟩	(X_7, +), (X_7, =), (X_7, then), (X_7, else) (X_7,]), (X_7, ;), (X_7, end)
X_7 → ⟨term ⟩*	(X_6, *), (⟨if clause ⟩ , *), (X_{12}, *), (X_{11}, *) (X_4, *), (X_3, *), (X_9, *)
⟨factor ⟩ → ⟨var ⟩	(X_7, *), (X_7, +), (X_7, =), (X_7. then),
⟨factor ⟩→ ⟨number ⟩	(X_7, else), (X_6, +), (X_6, =), (X_5, then), (X_6, else), (X_6, else), (X_6, *), (⟨ if clause ⟩ ,*), (X_{12}, *), (X_4, *), (X_9, *), (X_3, *), (⟨if clause ⟩, else), (X_3, ;), (X_3, end), (X_9,]), (⟨ if clause ⟩ , +), (X_4, +), (X_3, +), (X_9, +), (S_{11}, =)
⟨factor⟩ → X_8]	(X_7,]), (X_6,]), (⟨if clause ⟩,]), (X_{12},]) (X_{11},]), (X_4,]), (X_9,]), X_3,])
X_8 → X_9 ⟨expr⟩	(X_9,])
X_9 → [	(X_9, [), (X_7, [), (X_6, [), (⟨ if clause ⟩ , [) (X_{12}, [), (X_{11}, [), (X_3, [), (X_4, [)

⟨if clause⟩ → X_{10} then	(X_3, then), (X_4, then), (X_9, then)
X_{10} → X_{11} ⟨relation⟩	(X_{11}, then)
X_{11} → if	(X_3, if), (X_4, if), (X_9, if)
⟨relation⟩ → X_{12} ⟨simple expr⟩	(X_{12}, then)
X_{12} → ⟨simple expr⟩ =	(X_{11}, =)
⟨var⟩ → A	(X_7, A), (⟨body⟩, A), (X_6, A), (X_4, A), (X_9, A), (X_3, A), (X_{12}, A), (X_{11}, A), (⟨if clause⟩, A)
⟨number⟩ → ⟨number⟩⟨digit⟩	(⟨number⟩, 1),...,(⟨number⟩, 9) (⟨number⟩, *), (⟨number⟩, +) (⟨number⟩, =), (⟨number⟩, then) (⟨number⟩, else), (⟨number⟩, ;) (⟨number⟩, end)
⟨digit⟩ → 0|1|...|9	(⟨number⟩, 0),..., (⟨number⟩, 9), (⟨if clause⟩, 0),...,(⟨if clause⟩, 9), (X_7, 0),..., (X_7, 9), (X_6, 0),..., (X_6, 9), (X_4, 0),..., (X_4, 9), (X_{11}, 0),..., (X_{11}, 9), (X_{12}, 0),..., (X_{12}, 9),

From the table of rules and contexts, a flow chart of the PDA that accepts L(G') can be constructed. This flow chart is abbreviated in that, for a given state, only those contexts necessary for determining a transition are presented. Thus, when no ambiguity will be introduced, only the stack symbol or the input string symbol is used for determining which of several possible transitions can occur. In the flow chart, the array N, with index i is used to represent the symbols of the stack, and the array S, with index j, represents the symbols of the input string. Note that "error exits", i.e., the instances of case (0) in the leftmost parsing algorithm, are omitted from the flow chart. An error is assumed to exist at any transition in which the appropriate symbols are not present on either the input string or the stack.

The next step after synthesizing a PDA as in Figure 1 is to design a translator for the language accepted by that PDA. To do this, we employ a notation similar to that used in [13], and available in numerous versions in the current literature. The basic idea of the notation is to introduce <u>rules of translation</u> in a one-to-one correspondence with the rules of the original grammar. These rules of translation describe the effect of translating the right parts of the syntactic rules with which they are associated. As an example, we might have the following pairing in some grammar:

<u>Syntactic Rule</u>:	<u>Rule of Translation</u>:
⟨term⟩ → ⟨term⟩ * ⟨factor⟩	⟨term⟩⟨factor⟩ <u>multiply</u>

This pairing of rules can be represented as a <u>translation grammar</u> $G^t = (G, O, f)$, where $G = (V, T, P, S)$ is the programming language syntax, O is a translated program vocabulary, and f is a one-to-one mapping from P onto PXO^*. The rule of translation given in the above example is easily recognized as one rule for converting from standard arithmetic notation to reverse Polish notation. In the translated sequence '⟨term⟩ ⟨factor⟩ <u>multiply</u>', the translated objects corresponding to ⟨term⟩ are written out in the sequence determined by the rules of translation associated with the syntactic rules derived from ⟨term⟩, and likewise with ⟨factor⟩ . In general, if a rule of translation is identical to the right part of its associated syntax rule, we write the symbol 'I' in place of the rule of translation. If the rule of translation is the symbol 'e', then the right part of its associated syntax rule is not written out by the trnaslator.

We can next present the simple programming language given above as a translation grammar:

	Syntactic Rules:	Rules of Translation:
G:	⟨program⟩ → ⟨body⟩ ⟨stat⟩ end	I
	⟨body⟩ → begin	I
	⟨body⟩ → ⟨body⟩⟨stat⟩ ;	I
	⟨stat⟩ → ⟨program⟩	I
	⟨stat⟩ → ⟨assignment⟩	I
	⟨assignment⟩ → ⟨var⟩:=⟨expr⟩	⟨var⟩ ⟨expr⟩ assign
	⟨expr⟩ → ⟨simple expr⟩	I
	⟨expr⟩ → ⟨if clause⟩ ⟨simple expr⟩ else ⟨expr⟩	⟨if clause⟩⟨simple expr⟩ then ⟨expr⟩ else
	⟨simple expr⟩ → ⟨term⟩	I
	⟨simple expr⟩ → ⟨simple expr⟩ + ⟨term⟩	⟨simple expr⟩⟨term⟩ add
	⟨term⟩ → ⟨factor⟩	I
	⟨term⟩ → ⟨term⟩ * ⟨factor⟩	⟨term⟩⟨factor⟩ multiply
	⟨factor⟩ → ⟨var⟩	I
	⟨factor⟩ → ⟨number⟩	numberoperand ⟨number⟩
	⟨factor⟩ → [⟨expr⟩]	⟨expr⟩
	⟨if clause⟩ → if ⟨relation⟩ then	⟨relation⟩ if

⟨relation⟩ → ⟨simple expr⟩$^{(1)}$=⟨simple expr⟩$^{(2)}$	⟨simple expr⟩$^{(1)}$ ⟨simple expr⟩$^{(2)}$ equals
⟨var⟩ → A	variableoperand A
⋮	⋮
⟨var⟩ → Z	variableoperand Z
⟨number⟩ → ⟨digit⟩	I
⟨number⟩ → ⟨number⟩⟨digit⟩	10 ⟨number⟩ multiply ⟨digit⟩ add
⟨digit⟩ → 0	I
⋮	⋮
⟨digit⟩ → 9	I

The details of the translator grammar can be explained briefly: Essentially, arithmetic expressions and relations are translated into reverse-Polish strings through the rules of translation. Conditional expressions are rearranged so that if, then, and else become labels in the translated program, and the device that interprets the translated program contains routines for passing to the statements directly following if, then, or else as appropriate. Since the effect of interpreting the translated program is to coalesce assignment statements into a single resultant operand that is the "value" of the assigned expression, the semicolon ";" that separates program statements is written into the translated program so that the interpreting mechanism can erase the resultant operand of an assignment. begin and end are likewise written in sequence into the translated program so that the interpretor of this program can maintain a list of valid identifiers corresponding to the program's nested block structure.

The translator of Figure 2 is thus a relatively straightforward extension of the PDA in Figure 1, with the additional structure arising from the appropriate rules of translation. The sequencing of operators to follow pairs of operands is accomplished by noting that state transitions such as the one that recognizes the sequence

$\langle$simple expr$\rangle$ + $\langle$ term $\rangle$

in Figure 1 are appropriate points for writing out operators (here, "add") into the translated program. Likewise, a rule of transition such as

numberoperand $\langle$number$\rangle$

requires some temporary storage in the translator to store the symbols that comprise $\langle$number$\rangle$, finally writing out the translated sequence

```
Code ('numberoperand')
Code (temporarystore).
```

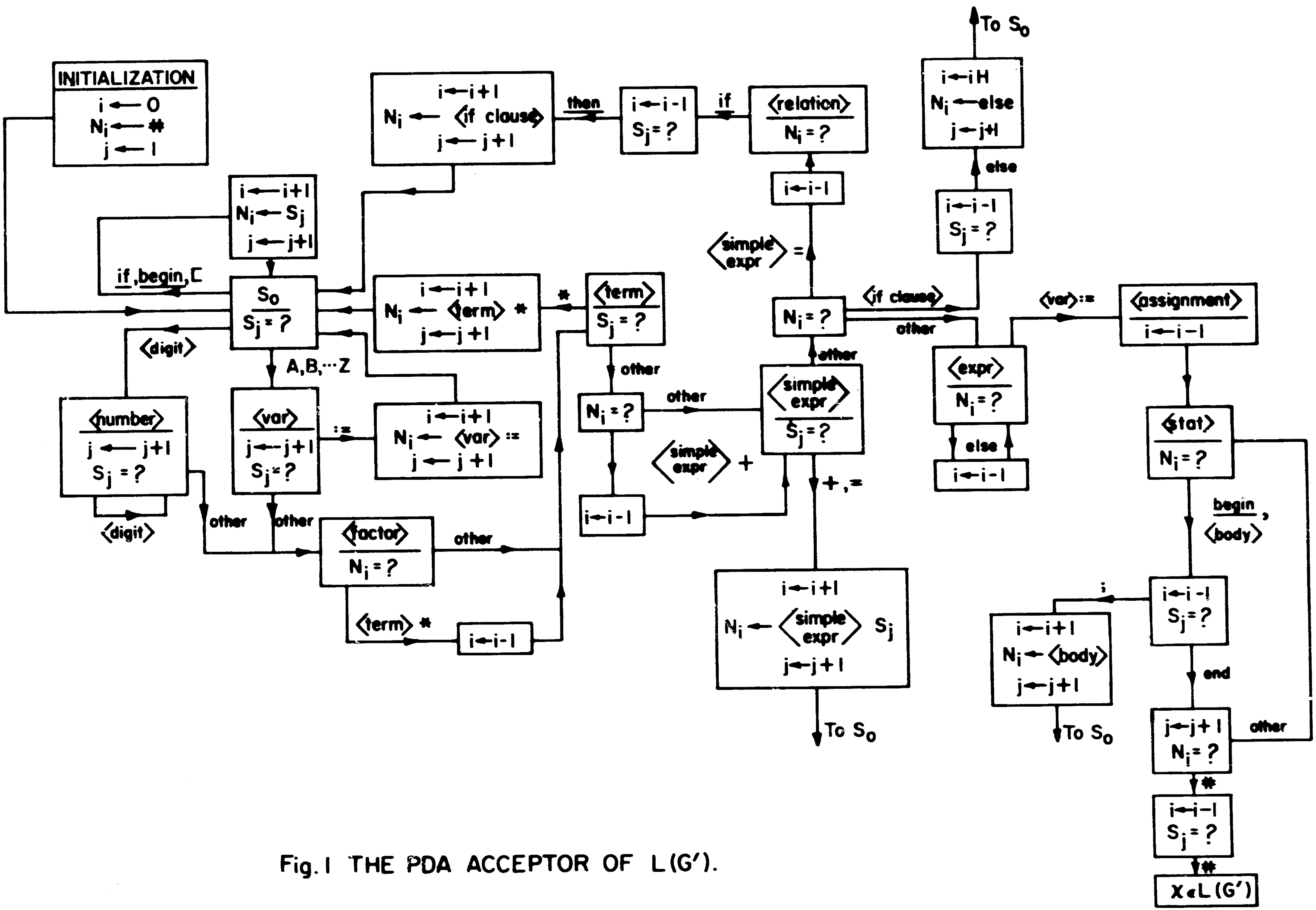

Fig. I THE PDA ACCEPTOR OF L(G').

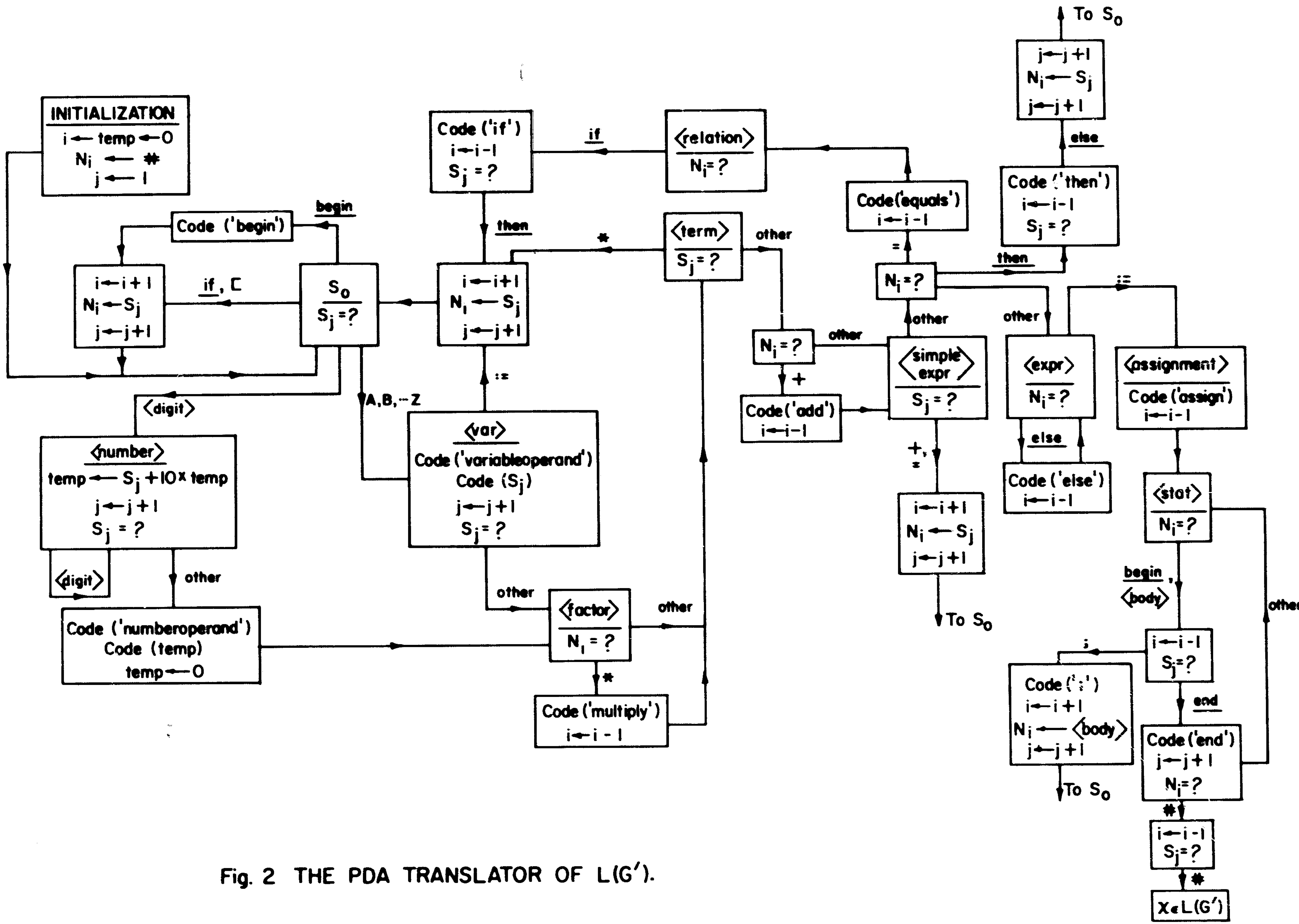

Fig. 2 THE PDA TRANSLATOR OF L(G').

Deterministic and Extended Deterministic Automata

A pushdown automaton A is called deterministic if M is a (partial) mapping from Q x T x N into Q x N x D. This is equivalent to saying that, for every configuration C_1 of machine A, there is at most one configuration C_2 such that $C_1 \vdash C_2$. Hence, for every x in L(A), there exists exactly one leftmost parse, and L(A) is unambiguous. However, the notion of a deterministic PDA is a relatively restricted one, and in the following paragraphs, we present one method for extending this notion.

A machine A will be called extended deterministic (abbreviated XD) if, for every string x in L(A), there exists only one sequence of configurations $C_{1x}, C_{2x}, \ldots, C_{nx}$ such that

$$C_{1x} = (\# \; S_0 \; x \; \#), \; C_{nx} = (\# \; F \; \#)$$

and $\quad C_{ix} \vdash C_{i+1,x} \qquad$ for $i = 1, \ldots, n-1$.

Obviously, a deterministic PDA is extended deterministic, but not conversely. It is also clear that the class of XD-PDA's corresponds to the class of unambiguous CFL's. However, it is well known that no general algorithm exists for testing a CFL for ambiguity. Rather than addressing ourselves to the solution of an unsolvable problem, we choose to look at a more restricted version of the problem: Namely, in terms of our leftmost parsing algorithm, in which grammars can the necessity of backtracking during a computation be eleminated? Essentially, our solution to this problem involves an algorithm for extending our notion of a PDA to include the capability of keeping track of all alternatives that arise during a computation over an input string and for reading the input tape one symbol at a time without backing up.

Multiple Configurations

A multiple configuration C' of some machine A is a triple

$$C' = ((\# \, v_1, \ldots, \# \, v_m) \; S_c \; a \; w \; \#),$$

where $v_1, \ldots, v_m$ are in N* and S_c is in P(Q)-Q, where P(Q) is the set of subsets of Q, aw is in T* as before, and the number of states in S_c is m.

Given a PDA, let $\vdash$ be the relation on

$$(\# \, N^* \times Q \times T^* \, \#) \cup (\# \, N^* \times \ldots \times \# \, N^* \times (P(Q)-Q) \times T^* \, \#)$$

defined as follows:

I. Let $C_1 = (\# \, v \, g \, S_1 \, a \, w \, \#)$, where g is in N, a is in T, S_1 is in Q, v is in N*, and w is in T*.

(a) Let $S_c = \{c_i\colon [(c_i, b_i, 1) \in M(S_1, a, g)]\}$.
Then, $C_2' = ((\# \, v \, b_1, \ldots, \# \, v \, b_k) \; S_c \; a \; w \; \#)$
and $C_1 \vdash C_2'$. We say that
$M(S_1, a, g) = (S_c, (b_1, \ldots, b_k), 1)$.

(b) Let $S_c = \{c_i\colon [(c_i, b_i, 2) \in M(S_1, a, g)]\}$.
Then, $C_2' = ((\# \, v \, g \, b_1, \ldots, \# \, v \, g \, b_k) \; S_c \; w \; \#)$
and $C_1 \vdash C_2'$. We say that
$M(S_1, a, g) = (S_c, (b_1, \ldots, b_k), 2)$

(c) Let $S_c = \{c_i\colon [(c_i, b_i, 3) \in M(S_1, a, g)]\}$.
Then, $C_2' = ((\# \, v \, g \, b_1, \ldots, \# \, v \, g \, b_k) \; S_c \; a \; w \; \#)$
and $C_1 \vdash C_2'$. We say that
$M(S_1, a, g) = (S_c, (b_1, \ldots, b_k), 3)$

II. Let $C_1' = ((\# v_1 b_1, \ldots, \# v_t b_t) S_a d \; w \#)$, where the b_i are in N, the v_i are in N*, S_a is in P(Q)-Q, d is in T, and w is in T*.

(a) Let

$$S_B = \{B_i : (\exists a_j) [(a_j \in S_a) \& (B_i, c_i, 3) \in M(a_j, d, b_j)]\}.$$
$$\cup \{a_k : (a_k \in S_a) \& [[Q \times N \times 2 \supseteq M(a_k, d, b_k)] \vee [Q \times N \times 1 \supseteq M(a_k, d, b_k)]]\}$$

Then,

$$C_2' = ((\# v_{i1} b_{i1} c_{i1}, \ldots, \# v_{ij} b_{ij} c_{ij}, \# v_{k1} b_{k1}, \ldots, \# v_{km} b_{km}) S_B \; w \#)$$

and $C_1' \vdash C_2'$. We say that

$$M(S_a, d, (b_{i1}, \ldots, b_{ih})) = (S_B, (c_{i1}, \ldots, c_{ij}), 3)$$

(b) Let

$$S_B = \{B_i : (\exists a_j)[(a_j \in S_a) \& [(B_i, c_i, 1) \in M(a_j, d, b_j)]] \& \sim (\exists a_k)[(a_k \in S_a) \& (Q \times N \times 3 \supseteq M(a_k, d, b_k)]\}$$
$$\cup \{a_k : (a_k \in S_a) \& Q \times N \times 2 \supseteq M(a_k, d, b_k)) \& \sim (\exists a_j)[(a_j \in S_a) \& (Q \times N \times 3 \supseteq M(a_j, d, b_j)]\}$$

Then,

$$C_2' = ((\# v_{i1} c_1, \ldots, \# v_{iu} c_u, \# v_{k1} b_{k1}, \ldots, \# v_{kt} b_{kt}) S_B \; d \; w \#)$$

and $C_1' \vdash C_2'$. We say that

$$M(S_a, d, (b_{i1}, \ldots, b_{iv})) = (S_B, (c_1, \ldots, c_u), 1)$$

(c) Let

$$S_B = \{B_i : (\exists a_j)[(a_j \in S_a) \& [(B_i, c_i, 2) \in M(a_j, d, b_j)] \& \sim (\exists a_k)[(a_k \in S_a) \& (Q \times N \times 3 \supseteq M(a_k, d, b_k)] \& \sim (\exists a_h)[(a_h \in S_a) \& (Q \times N \times 3 \supseteq M(a_h, d, b_h)]\}$$

Then,

$C_2' = ((\# \; v_{i1} \; b_{i1} \; c_1, \ldots, \# \; v_{ip} \; b_{ip} \; c_p) \; S_B \; w \; \#)$

and $C_1' \vdash C_2'$. We say that

$$M(S_a, d, (b_{i1}, \ldots, b_{i1})) = (S_B, (c_1, \ldots, c_p), 2)$$

We see that the transitions from a single state to a multiple configuration and from one multiple configuration to another or to a single state preserve the actions of the leftmost parsing algorithm. That is, a reduction sequence over a string x is contained in any reduction sequence involving multiple configurations. The extra stacks used during a multiple computation simply keep track of additional possibilities until all but one sequence of configurations is eliminated.

In part I of the definition, note that the transitions are not uniquely defined if, for a state s and a particular pair of symbols (a, b),

$$(S \times N \times 1 \supseteq M(s, a, b)) \; \& \; (S \times N \times 2 \supseteq M(s, a, b))$$
$$\vee \; (S \times N \times 3 \supseteq M(s, a, b) \; \& \; (S \times N \times 2 \supseteq M(s, a, b))$$

In both of these circumstances, the necessity of simultaneously erasing and not erasing the same input tape symbol (a) during one transition is not compatible with our algorithm. The presence of such a transition may imply that lookahead techniques should be used to decide which of the two or more transitions should take place. These methods will be discussed in another paper.

If there exists a state q in some multiple configuration S_q, and such that q is descended from two or more states $y_{i1}, \ldots, y_{iq}$ in S_y for which

$$M(S_y, a, b) = (S_q, (c_1, \ldots, c_n), d)$$

then $\vdash$ is not uniquely defined for this transition. This is because there is no longer a one-to-one association of pushdown store strings and states of S_q. In such a case, more than one possible leftmost parse may exist for a string of the machine's language. When these two degenerate cases arise during the construction of an XD-PDA, it is instructive to rewrite the PDA as a rightmost parsing algorithm to see whether the same problems arise when parsing strings of a language from right to left.

For a PDA in which multiple configurations are definable, we say that $C_{i1} \overset{*}{\vdash} C_{im}$ when there exists a sequence of (possibly multiple) configurations $C_{i1},\ldots,C_{im}$ such that

$$C_{ik} \vdash C_{i,k+1} \qquad \text{for } k = 1,\ldots, m-1.$$

The language of an automaton A in which multiple configurations are definable is then given by

$$\begin{aligned} L(A) = \{x: (x \in T^* - \{e\}) \;\&\; [[(\# S_0\, x\, \#) \overset{*}{\vdash} (\# F \#)] \\ \vee\, [\# S_0\, x\, \#) \overset{*}{\vdash} ((\# v_1,\ldots, \# v_n)\, S_F\, \#)] \\ \&\; (S_F \in P(Q)-Q) \\ \&\; (\exists\, q_i)[(q_i \in S_F) \;\&\; (q_i = F) \;\&\; (v_i = e)]]\} \end{aligned}$$

With these preceding definitions in mind, we can then state the following theorems that are proved in [10].

<u>Theorem 1</u>. Let A be a PDA for which $\vdash$ is uniquely defined for all multiple configurations. Let

$$A' = (Q', t, N', M', D, \# S_0, F)$$

with $Q' \subseteq P(Q)$, $N' \subseteq N \cup N \times \ldots \times N$, and M' the original M of A together with the transitions defined on multiple configurations.

Then, $L(A) = L(A')$.

That this is so follows from the observation that, for $\vdash$ uniquely defined on multiple configurations of A, all computations of A over some string x in L(A) are contained in a single computation of A' over that string. Conversely, no computation of A' over some string x will succeed unless x is in L(A).

Theorem 2. Let A be a PDA having multiple configurations for which $\vdash$ is uniquely defined. Then L(A) is unambiguous.

That this is so arises from the fact that the conditions for uniqueness of $\vdash$ also insure unique leftmost parses of a particular language.

A Simple XD-PDA

The following is a grammar of Irons [6] chosen to illustrate simply the techniques that we have developed for constructing XD-PDA's.

$$
\begin{array}{lll}
G: & S \rightarrow a\ E \mid B\ d & \\
 & E \rightarrow b\ D & B \rightarrow A\ c \\
 & D \rightarrow c\ f & A \rightarrow a\ b
\end{array}
$$

In terms of our PDA model, this grammar results in an XD acceptor as illustrated in Table 2. In this table an asterisk * is used to indicate that a particular symbol in that position need not be read.

Table 2. The Acceptor for L(G)

Present State	Input Symbol	Stack Symbol	Next State	Stack Symbols Written	Instruction D
S_o	b	*	S_o	X_2	2
	c	*	X_3	e	2
	a	*	(S_o,X_4)	(X_1,e)	2
(S_o,X_4)	b	*	(S_o,A)	(X_2,e)	2
	c	*	X_3	e	2
	a	*	(S_o,X_4)	(X_1,e)	2
(S_o,A)	b	*	S_o	X_2	2
	c	*	(X_3,B)	(e,e)	2
	a	*	(S_o,X_4)	(X_1,e)	2
(X_3,B)	d	#	S	e	2
	f	X_2	D	e	2
D	#	X_2	E	e	1
E	#	X_1	S	e	1
S	#	#	-	-	-

Upper Bounds on Storage and Computation Times

In this section, we propose to derive upper bounds on stack length and number of configurations required for a computation by a deterministic PDA. Let $x = a_1 \ldots a_n$ be an input string to some PDA, and let $y = \# x_1 \ldots x_m$ represent the string of symbols on the stack at some point during a computation. Then, after symbol a_k is erased by the PDA ($k = 1, \ldots, n$), there are at most $(k + 1)$ symbols on the stack. This is so because, by the definition of a PDA,

(a) For each input symbol erased, at most one symbol can be written on the stack.

(b) For each stack symbol erased, at most one symbol can take its place.

Hence, in particular, there are never more than $(n + 1)$ symbols on the stack, where n is the length of the input string.

In the case of a PDA with multiple-state configurations, there can never be more than k stacks active at once, where k is the number of states of the PDA. Hence, for such a PDA, there is an upper bound of $k(n + 1)$ symbols stored on stacks during a computation.

In order to arrive at an upper bound for computation time of a PDA, we must first discover certain additional properties of the PDA model introduced in this paper. A good beginning is to discover an upper bound on the number of actions that can be taken by a PDA without erasing an input string symbol during a computation.

Let p be the number of rules in the grammar for the PDA such that the rules form a chain

$$A^{(i)} \rightarrow A^{(i+1)} \qquad i = 1, \ldots, p-1,$$

such that all the rules of the chain have at least one context in common, and such that p denotes the length of the longest chain of rules of this sort in the grammar. Then, without the erasure of a symbol from the stack, at most p state transitions can occur during a computation.

If q_k symbols are on the stack after input string symbol a_k has been erased, at most

$$(1 + q_k)\ (p + 1)$$

state transitions can occur before a_{k+1} is erased. If only w_k symbols are removed from the stack, then at most

$$(1 + w_k)\ (p + 1) \quad , \qquad w_k = 0, 1, \ldots, q_k$$

state transitions can take place before a_{k+1} is erased.

We can then ask what total number of symbols can be erased from the stack during any computation. I.e., what is the maximum value of

$$\sum_{k=1}^{n} w_k \quad ?$$

to answer this question, we note again that our PDA model only allows a new stack symbol to be written as a result of the erasure of an input string symbol. Since for an input string of length n no more than n new symbols can be written on the stack, no more than n symbols can be extracted from the stack during any computation.

Hence,

$$\sum_{k=1}^{n} w_k \leq n.$$

Finally, we can arrive at an upper bound on the number of configurations that can appear during a computation of a PDA over a string x of length n.

$$MAX \leq (n + 1) + (p + 1)\ (w_1 + \ldots + w_n + n)$$

or $$MAX \leq n(2p + 3) + 1$$

We also know that

$$MAX \leq n + 1,$$

where this lower bound is reached when the PDA acceptor of some language has an empty stack vocabulary, i.e., is a finite-state acceptor. Hence,

$$n + 1 \leq MAX \leq n(2p + 3) + 1$$

This upper bound on the number of configurations during a computation also holds for the XD-PDA's having multiple-state configurations. This is because the computations of the nondeterministic PDA from which the XD version was constructed are all included in the computations of the XD version.

Bibliography

1. Eickel, J., Paul, M., Bauer, F. L., and Samelson, K. A Syntax-Controlled Generator of Formal Language Processors. Comm. ACM 6 (August 1963), 451-455.

2. Ginsburg, S., Greibach, S., and Harrison, M. Stack Automata and Compiling. J ACM 14 (Jan. 1967), 172-201.

3. Ginsburg, S. The Mathematical Theory of Context-Free Languages. McGraw-Hill, New York, 1966.

4. Greibach, S. Inverses of Phrase Structure Generators. Doctoral Dissertation, Harvard University, Cambridge, Mass., 1963.

5. Haines, L. H. Generation and Recognition of Formal Languages. Doctoral Dissertation, M. I. T., Cambridge, Mass., 1965

6. Irons, E. T. "Structural Connections" in Formal Languages. Comm. ACM 7 (February 1964), 67-72.

7. Knuth, D. E. On the translation of languages from left to right. Information and Control 8 (Dec. 1965), 607-639.

8. Matthews, G. H. Discontinuity and Assymmetry in Phrase Structure Grammars. Information and Control 6 (June 1963), 137-146.

9. Schneider, V. B. The Design of Processors for Context-Free Languages. National Science Foundation Memorandum, Department of Industrial Engineering and Management Science, Northwestern University, Evanston, Illinois, August, 1965.

10. Schneider, V. B. Pushdown-Store Processors of Context-Free Languages. Doctoral Dissertation, Northwestern University, Evanston, Illinois, 1966.

11. Schneider, V. B. Syntax-Checking and Parsing of Context-Free Languages by Pushdown-Store Automata. Proceedings of the Spring Joint Computer Conference (1967), 685-690.

12. Shamir, E. Mathematical Models of Languages. Proceedings of the I. F. I. P. Congress (1965), 71-75.

13. Wirth, N. and Weber, H. EULER: A Generalization of ALGOL and Its Formal Definition: Parts I and II. Comm. ACM 3 (Jan.-Feb. 1966).