PREFIX PRECEDENCE GRAMMAR

by

Jeff Nee Yang

Memorandum No. ERL-M451

16 July 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PREFIX PRECEDENCE GRAMMAR*

by

Jeff Nee Yang

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

I. Introduction

A class of context-free grammars called prefix precedence grammars is defined. This class is shown to include simple precedence grammars as a proper subset. The construction of a fast, small, and extensible parser for these grammars is described. Four inspection rules are given which allow one to check whether a given grammar is a prefix precedence grammar. These features make prefix precedence grammars useful in the design of extensible programming languages [2].

II. Terminology

A language is a set of sentences. Each sentence is a finite string of symbols from an alphabet set Σ. Hence a language is a subset of the set Σ^* of all finite strings of symbols from Σ. And a grammar is a means of defining a language (i.e. specifying which strings are sentences in the language) and ascribing structure to its sentences. In this paper, we specify a language, L(G), by a context-free grammar, G, which is defined as a finite set of productions (or rewriting rules) of

---

*This paper is a development of one of the ideas proposed in [2]. Somewhat similar ideas have been developed independently by M. Geller in his work on production prefix grammars.

Research sponsored by the Naval Electronic Systems Command Contract N00039-71-C-0255.

the form $A \rightarrow x$ with the following properties:

1. $A \in N$, where N is a finite set of nonterminal symbols which are used in defining the language. $N \cap \Sigma = \phi$.
2. x is a nonempty string whose symbols are in the set $N \cup \Sigma$.

A is called the left part and x the right part of the production $A \rightarrow x$. There is a special symbol in N called the starting symbol and denoted by S. Figure 1 is an example of a grammar with $\Sigma = (a,*,+)$, $N = (E,T)$, and $S = E$.

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow a * T$

$T \rightarrow a$

FIG 1. Grammar G1: $N = (E,T)$; $\Sigma = (a,*,+)$; $S = E$

Unless otherwise specified, uppercase letters at the beginning of alphabet (A,B,C,...) are used for nonterminal symbols in N, lowercase letters at the beginning of alphabet (a,b,c,...) for symbols in Σ, uppercase letters at the end of alphabet (W,X,Y,...) for symbols in V ($V = N \cup \Sigma$), and lowercase letters at the end of alphabet (u,v,w,x,...) for strings in V.

In order to show how a context-free grammar defines a language, we need some further definitions. We say that y is a direct derivative of z (written $z \Rightarrow y$) by applying the production $A \rightarrow x$ if there are (possibly empty) strings u and v such that $z = uAv$ and $y = uxv$. The substring x is called a phrase of string y. The transitive closure of "$\Rightarrow$" is denoted by "$\overset{*}{\Rightarrow}$". $z \overset{*}{\Rightarrow} y$ if there exist strings $Z_0, Z_1, Z_2, \ldots, Z_n$

such that $Z = Z_0$, $Z_0 \Rightarrow Z_1 \ldots Z_{i-1} \Rightarrow Z_i$ and $Z_i = y$. y is called the derivative of z, and the sequence $Z = Z_0 \Rightarrow Z_1 \Rightarrow \ldots Z_i = y$ is a derivation of y from z. We write $Z \overset{+}{\Rightarrow} y$, if $Z \overset{*}{\Rightarrow} y$ and $Z \neq y$. The derivatives of the starting symbol S are called the sentential forms. A sentence is a sentential form consisting only of terminals. The language L(G) of a grammar G is defined as the set of sentences.

$$L(G) = \{x \mid S \overset{*}{\Rightarrow} x \wedge x \in \Sigma^*\}$$

In the grammar G1 of Fig. 1, L(G1) is the set of all arithmatic expressions using operator + and *, and the operand a. The sentence a+a*a can be derived from E as follows: $E \Rightarrow E + T \Rightarrow E + a*T \Rightarrow E + a*a$ $T + a*a \Rightarrow a + a*a$. Any sentential form can be represented by a syntax tree reflecting its derivation. The syntax tree of a + a*a is shown in Fig. 2. In Fig. 2a, we put phrase markers at the beginning and the end of each production in order to illustrate the phrase structure of the sentence and to give some idea of how the sentence is derived from the starting symbol. Thus, given a context-free grammar one can generate sentences of the language by deriving them and their syntax trees from the starting symbol. Compilers, on the other hand, have the opposite problem: given a sentence r and a grammar G, construct a derivation of r and find a corresponding syntax tree. This is the process of parsing of the sentence.

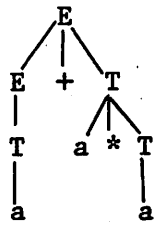

Fig. 2. Syntax Tree of a+a*a

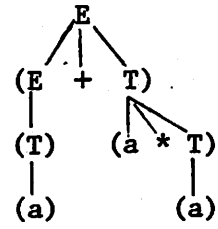

Fig. 2a. (((a)) + (a*(a)))

III. Bottom Up Parsing

The parsing algorithm that we shall be considering belongs to the class of bottom up parsing methods which analyze the input sentence r and construct the syntax tree of r from bottom to top. A bottom up parsing algorithm can be thought of as the iteration of the following two steps applied to a string $y \in V^*$. Initially y is in Σ^*.

1. Find the leftmost phrase, x, of y (detection of phrase).
2. Determine the production involved, say $A \rightarrow x$, and replace x by A to obtain a new y (reduction of phrase).

Assume we know all the phrase markers of the input string (e.g. string in Fig. 2a). Detection of phrase in a bottom up parsing algorithm now consists of scanning the input string from left to right until the first) is encountered and then retreating back to the last (. Between these two parentheses is a phrase and we reduce this phrase to the corresponding nonterminal by consulting the production set. Fig. 3 shows the history of bottom up parsing of the input string a+a*a.

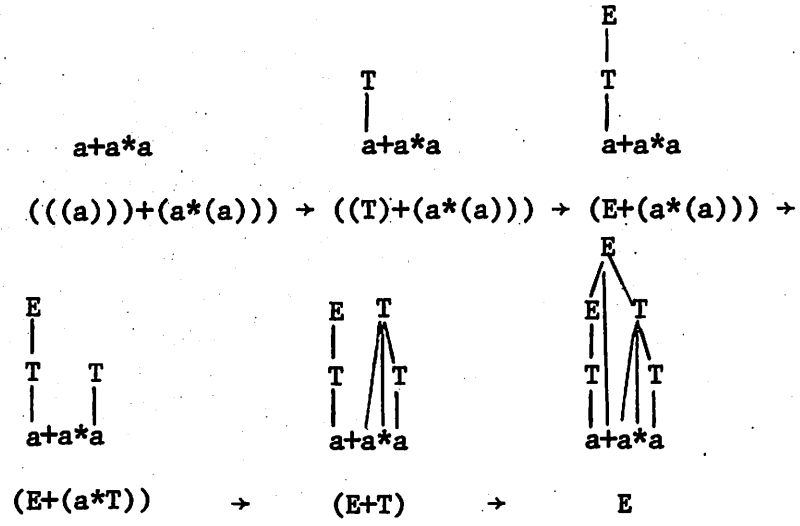

Fig. 3. History of bottom up parsing of the input string a+a*a.

In general, if the given grammar is uniquely invertible (i.e., no two productions have the same right part), reduction can be done easily once the phrase is detected. Hence detection of the phrase, in other words, finding phrase markers, is the major problem in parsing.

A class of context-free grammars called simple precedence grammars was first described by Wirth and Weber [1]. It has the property that all the phrase markers can be detected by comparing two adjacent symbols in the sentential form. Grammar G1 is a simple precedence grammar and its precedence matrix is shown in Fig. 4. Where $\langle\cdot$ corresponds to left phrase marker, $\cdot\rangle$ corresponds to right phrase marker, $\doteq$ means no phrase marker, and blank means error.

	E	T	a	+	*	Λ
E				$\doteq$		$\cdot\rangle$
T				$\cdot\rangle$		$\cdot\rangle$
a				$\cdot\rangle$	$\doteq$	$\cdot\rangle$
+		$\doteq$	$\langle\cdot$			
*		$\doteq$	$\langle\cdot$			
Λ	$\langle\cdot$	$\langle\cdot$	$\langle\cdot$			

Fig. 4. Precedence Matrix of Grammar G1 (Λ represents empty string).

To parse a sentence of a simple precedence grammar, we scan the input string from left to right and compare each pair of adjacent symbols in sequence. If the precedence relation between these two symbols is $\doteq$, then the scanning head is moved one character to the right. If the relation is $\langle\cdot$, then a $\langle\cdot$ is inserted between these two symbols and the

head moved to the right. If the relation is $\cdot\rangle$, then the scanning head is retreated back to the left of the last $\langle\cdot$ and the phrase is reduced to the corresponding nonterminal.

The idea of simple precedence grammar can be generalized as follows: instead of comparing two adjacent symbols to detect the phrase, we use all the symbols back to the last $\langle\cdot$ (i.e., the prefix of the right part of some production) to detect the phrase since eventually this information will be used to reduce the phrase.

IV. Prefix Precedence Grammars

Definition. There are three <u>prefix</u> precedence relations, $\lessdot$, $\doteq$, and $\cdot\rangle$, between a prefix w of some production and a symbol Y. They are defined as follows.

1. $w \doteq Y$ if $A \to wYu$ is a production.
2. $w \langle\cdot\ Y$ if $A \to wBu$ is a production and $B \overset{+}{\Rightarrow} Yu$.
3. $W \cdot\rangle\ Y$ if $A \to w$ is a production and $Y \in F(A)$, where $F(A) = \{b \mid b \in \Sigma,\ S \overset{*}{\Rightarrow} yAbz\}$.

Definition. A context-free grammar is a <u>prefix precedence grammar</u> iff the following two conditions are satisfied:

1. At most one prefix precedence relation holds between a pair of prefix and symbol.
2. If two productions $A \to x$ and $B \to x$ have the same right part, then $F(A) \cap F(B) = \phi$.

Grammar G1 is a prefix precedence grammar. The prefix precedence relation of grammar G1 is shown in Fig. 5. Each row corresponds to a prefix and we refer to it as a state. The corresponding parsing flowchart is shown in Figure 6.

		a	+	*	E	T	Λ
1	E		≐				>
2	E+	<				≐	
3	E+T		>				>
4	T		>				>
5	a		>	≐			>
6	a*	<				≐	
7	a*T		>				>
0	Λ	<			<	<	

Figure 5.

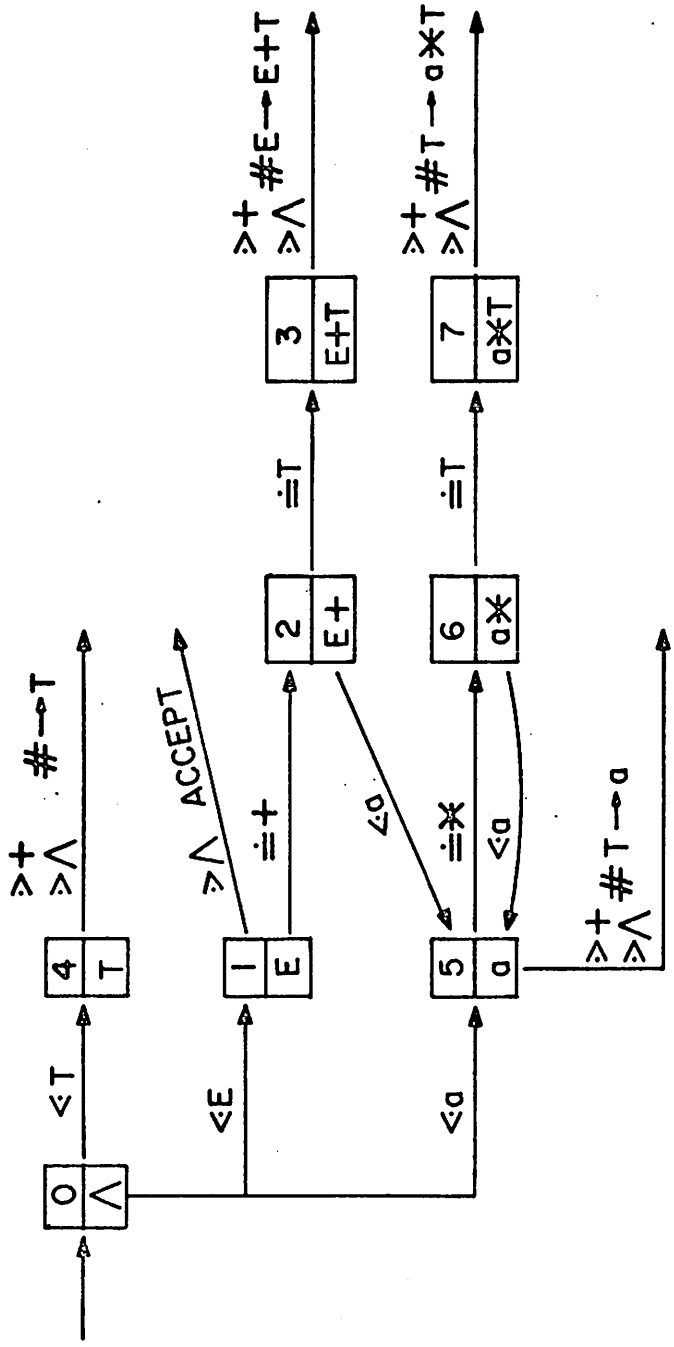

Figure 6.

During the parsing process we maintain a stack, denoted by

$$U\ S_w | Yu \qquad \ldots\ldots(1)$$

The portion to the left of the vertical line consists of the names of states; this represents the portion of a string which has already been scanned. To the right of the vertical line is the remaining input string. S_w is the current state and Y is the current input character.

Initially we are in state S_0, the stack to the left of the vertical line in (1) contains only S_0, and the string to be parsed appears to the right. Inductively, at any state during the parsing process and with the stack contents as given by (1), there are three possible next steps:

1. If $S_w \lessdot Y$, then delete Y and put S_y (next state) to the right of S_w,

$$US_w | Yu \rightarrow US_wS_y | u$$

2. If $S_w = Y$, then delete Y and change the current state S_w to S_{wy},

$$US_w | Yu \rightarrow US_{wy} | u$$

3. If $S_w \gtrdot Y$ and $Y \in F(A)$, then output $A \rightarrow w$, insert A to the right of Y, and pop off the current state (i.e., delete S_w),

$$US_w | Yu \rightarrow U | AYu$$

This process is iterated until the input string is reduced to a single S or an error is detected. Figure 7 shows the stack contents in the process of parsing the input string a*a+a.

State Stack	Input Stack	Output
0	a*a+a	
05	*a+a	
06	a+a	T → a
065	+a	
06	T+a	T → a*T
07	+a	
0	T+a	
04	+a	
0	E+a	E → T
01	+a	
02	a	
025		
02	T	T → a
03		
0	E	E → E+T
01		Accept

Fig. 7. Parsing History of Input String a*a+a.

Note that in our algorithm each state corresponds to a prefix of some production. During the parsing process, each prefix w is transformed to a single state name S_w and stored in the stack. Hence a smaller storage space is required compared with the simple precedence parsing. Note also that, unlike SLR(1) parsing, if $S_w \doteq Y$ we simply change the current state from S_w to S_{wy} without putting S_w in the stack. Thus our algorithm requires less storage space and is also faster than SLR(1) parsing.

V. Inspection Rules.

In this section, four inspection rules are given which allow one to check the conflict of prefix precedence. This feature is useful when designing a simple grammar for a given programming language. An equivalent definition of prefix precedence grammar defined by these four inspection rules is given as follows.

Theorem 1. A context-free grammar is a prefix precedence grammar iff the following four conditions are satisfied:

1. If two productions have the same prefix w (w is not empty),
   $A \to wXu$ and $B \to wYv$,
   then $X \overset{*}{\not\Rightarrow} Yz$ and $Y \overset{*}{\not\Rightarrow} Xy$.
2. If one production is the prefix of another production,
   $A \to w$ and $B \to wXu$,
   then $F(A) \cap I(X) = \phi$. Where $I(X) = \{b \mid b \in \Sigma, X \overset{*}{\Rightarrow} ba\}$
3. If A is a left recursive nonterminal (i.e. $A \overset{+}{\Rightarrow} Au$), then there is no production in the form $B \to yAz$, where y is not empty.
4. If two productions have the same right part, $A \to w$ and $B \to w$, then $F(A) \cap F(B) = \phi$.

Proof. Since rule 4 is identical to condition 2 of the first definition, we will just consider the equivalence of condition 1 to rules 1, 2 and 3 in the following proof.

Only if: In this part we will show that a given grammar violating any one of the inspection rules must have a prefix precedence conflict. Let us consider the conditions one by one.

(1) Assume that $A \to wXu$ and $B \to wYv$ are two productions, and that $X \overset{+}{\Rightarrow} Yz$. Then by the definition of prefix precedence we have $w \doteq Y$ and $w \lessdot Y$.

(2) Assume that $A \to w$ and $B \to wYv$ are two productions and that $b \in \{F(A) \cap I(X)\}$. Then we have $w \gtrdot b$ and $w \lessdot b$ ($w \doteq b$ if X is b).

(3) Assume that A is a left-recursive nonterminal, and that $B \to wAu$ is a production (w is not empty). Then we have $w \doteq A$ and $w \lessdot A$.

If: In this part we will show that any prefix precedence conflict will cause an immediate violation of the inspection rules.

(1) Suppose that $w \doteq b$ and $w \cdot\rangle b$. This means that there exist A, B, and u such that $A \to w$ and $B \to wbu$ are productions and $b \in F(A)$. Thus inspection rule 2 is violated.

(2) Suppose that $w \langle\cdot b$ and $w \cdot\rangle b$. This means there exist A, B, Y, u, and v such that $A \to w$ and $B \to wYu$ are productions, $Y \overset{+}{\Rightarrow} bv$ and $b \in F(A)$. Thus inspection rule 2 is violated.

(3) Suppose that $w \doteq X$ and $w \langle\cdot X$. This means there exist A, B, Y, u, v, and z such that either $A \to wXu$, $B \to wYv$ are productions and $Y \overset{+}{\Rightarrow} Xz$ or $A \to wXu$ is a production and X is a left-recursive non-terminal. Hence either inspection rule 1 or inspection rule 3 is violated.

Example: Grammar G2: $N = (S)$; $\Sigma = (b,d)$; $S = S$.

$S \to bbS$

$S \to bd$

$S \to d$

It can be checked immediately by using the above inspection rules that G2 is a prefix precedence grammar. G2 is not a simple precedence grammar or BRC(m,1) grammar for any m [4].

Theorem 2. Every simple precedence grammar is a prefix precedence grammar.

Proof. Let Tail(w) be the last symbol of the string w and Y be a symbol. We will prove the contrapositive. It is obvious that if there exists a prefix precedence relation ($\langle\cdot$, $=$, or $\cdot\rangle$) between w and Y, then the corresponding precedence relation ($\langle\cdot$, $=$, or $\cdot\rangle$) exists between Tail(w) and Y [1]. Therefore, if a grammar has a prefix precedence conflict

between w and Y, it must have a precedence conflict between Tail(w) and Y. Hence a given grammar is not simple precedence if it is not prefix precedence.

VI. Lookback Prefix Precedence Grammars.

In this section we continue the development of our parsing machine construction technique. We analyze the class of lookback prefix precedence grammars whose sentences can be parsed during the deterministic left-to-right scan with each parsing decision being made on the basis of the knowledge of both the lookahead symbol and lookback state name.

Example. Consider the parsing flowchart shown in Figure 8. It corresponds to grammar G3 which contains the following productions

(1) S → adAd	(4) S → bdBb
(2) S → adBc	(5) A → ee
(3) S → bdAc	(6) B → ee

Production 5 and 6 have the same right part and F(A) = F(B) = {c,d}. Hence G3 is not a prefix precedence grammar. At state 8 the lookahead symbol alone is not enough to decide whether to reduce using production 5 or 6. However, we could make the parsing decision associated with state 8 by looking at both our left and right contexts after arriving there. If we look to our left and see "3" (note that according to our parsing algorithm, state 3 or 4, not 7, are the possible left states of state 8 in the stack) then, if we look to our right and see d, #5 is the correct reduction, but if we see c, #6 is correct. On the other hand, if we see "4" to our left then the correspondences are d with #6 and c with #5.

For G3 and many other grammars we can define the function C(A)

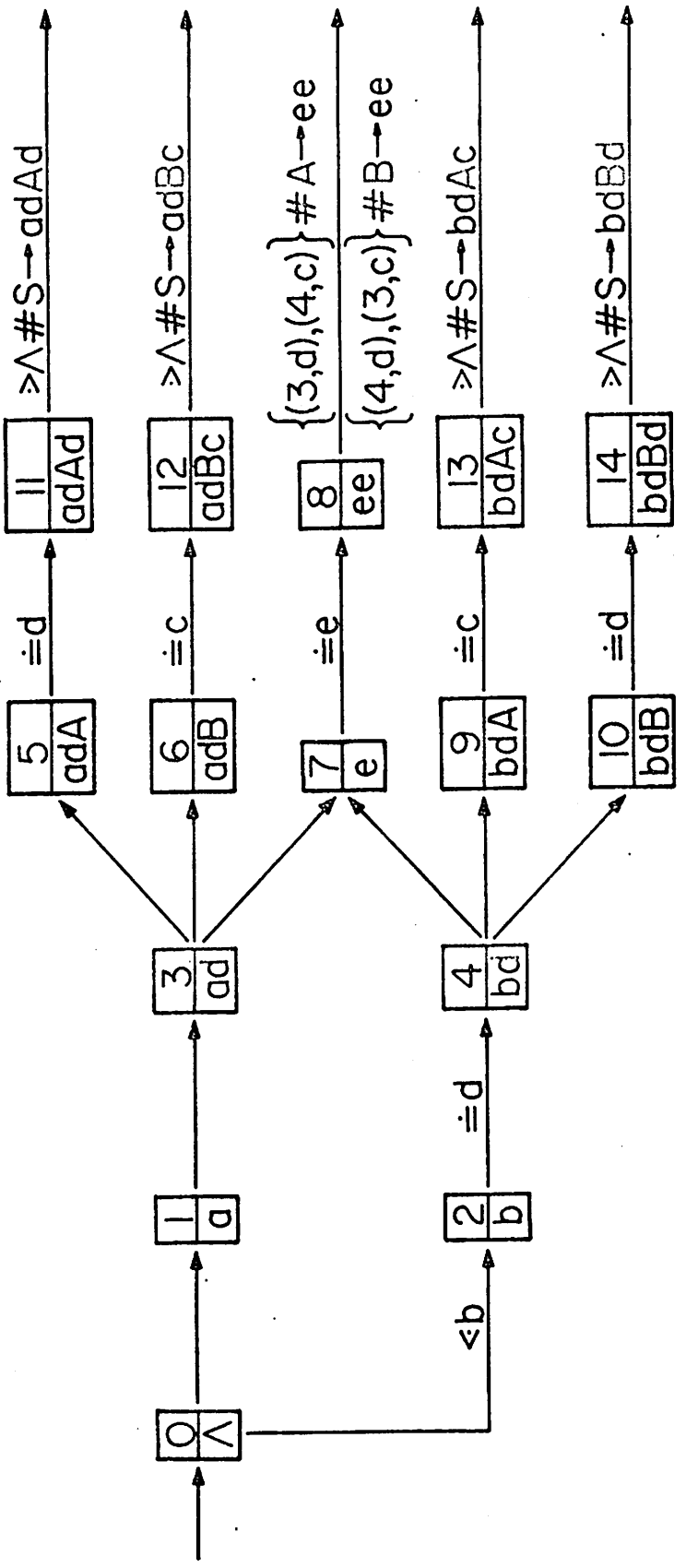

Fig. 8. Parsing Flowchart for Grammar G3.

for some nonterminal A whose value is a set of ordered pairs of predecessor states and lookahead symbols. The definition requires the following three preliminary definitions: (1) P(A), the predecessors of nonterminal A, is the set of all states of the parsing machine which emanate an A-transition arc, (2) $S(S_\omega|A)$, the successors of A-transition from state S_w, and (3) $\{(I,\Sigma)\}$ denote the set of pairs whose first components are state names and whose seconds are in Σ.

Definition Let A be a nonterminal of the given grammar. Then C(A) = $\{(S_\omega,b)|S_\omega \in P(A)$ and there exists a prefix precedence relation between $S(S_\omega|A)$ and b$\}$.

Example. In grammar G3, C(A) = {(3,d), (4,c)} and C(B) = {(3,c), (4,d)}. This result can be obtained easily from the parsing flowchart in Figure 8.

We now define the lookback prefix precedence grammar whose sentence can be parsed by our modified algorithm.

Definition. A context-free grammar is a lookback prefix precedence grammar iff the following four conditions are satisfied.

1. If two productions have the same prefix w (w is not empty), $A \rightarrow wXu$ and $B \rightarrow wYv$, and $X \overset{+}{\Rightarrow} Yz$ for some z, then $P(A) \cap P(B) = \phi$.
2. If one production is the prefix of another production, $A \rightarrow w$ and $B \rightarrow wXu$, and $F(A) \cap I(X) \neq \phi$, then $P(A) \cap P(B) = \phi$.
3. If A is a left recursive nonterminal, then there is no production in the form $B \rightarrow yAz$ with nonempty prefix y.
4. If two productions have the same right part, $A \rightarrow w$ and $B \rightarrow w$, then $C(A) \cap C(B) = \phi$.

It can be shown that every LR(K) grammar has an equivalent lookback prefix precedence grammar. The lengthy proof which involves several transformation grammars is ommitted here. Grammar G3 is an example of lookback prefix precedence grammar. The parsing flowchart of grammar G3 is shown in Figure 8. Note that G3 is not a SLR(K) grammar of a LALR(K) grammar [5].

The idea of making a parsing decision by using both left and right contexts surrounding the decision point was first described by Floyd's bounded context grammars [4]. Lookback prefix precedence grammars among other grammars such as SMSP grammars [6], BRC grammars [4] and L(m)R(k) grammars [5], employ similar parsing strategy. However, the C(A) sets of the lookback prefix precedence grammars is less difficult to compute than those of BRC grammars and L(m)R(k) grammars. Also, in our algorithm the lookback symbols are state names rather than symbols in V and hence provide more information. For example, grammar G3 is BRC (2,1) or L(4)R(1). Thus two or four lookback symbols are needed if these parsing methods are used.

VI. Conclusion.

The class of prefix precedence grammars is broad enough to describe most of the programming languages. The inspection rules are simple and straightforward, they can be applied at any stage during the designing and modifying of a language. The parser for prefix precedence grammar is small and fast. Each state of the parser corresponds to a prefix of some production, hence small changes in a grammar (e.g. addition or deletion of a production) only cause small corresponding modifications of the parser (addition or deletion of a few corresponding states). All

these features make prefix precedence grammars a useful model for programming languages.

Acknowledgement

The author wishes to express his deep gratitude to his research advisor, Professor Lotfi A. Zadeh for his guidance and encouragement throughout the preparation of this report; thanks are also due to Professor Jay Earley, Dr. Eric Cho, and Mr. Rowland Johnson for their helpful discussions and suggestions.

References

1. Wirth, N., and H. Weber (1966). EULER - a generalization of ALGOL and its formal definition, Parts 1 and 2. Comm. ACM 9: 1, 13-23 and 9: 2, 89-99.

2. Yang, J. N., (May,1973). Simple grammars and design of programming languages, Qualifying Exam. Note, U.C. Berkeley.

3. DeRemer, F. L. (1971). Simple LR(K) grammars. Comm. ACM 14: 7, 453-460.

4. Floyd, R. W. (1964). Bounded context syntactic analysis. Comm. ACM 7: 2, 62-67.

5. DeRemer, F. L. (1969). Practical translators for LR(k) languages. Ph.D. Thesis, MIT, Cambridge, Mass.

6. Aho, A. V., and J. D. Ullman (1972). Weak and mixed strategy precedence parsing. J. ACM 19: 2, 225-243.