AN ABSTRACT OF THE THESIS OF

YOSHIYUKI YAMAGISHI (Name) for the MASTER OF SCIENCE (Degree)

in MATHEMATICS (Major) presented on December 10, 1971 (Date)

Title: PRECEDENCE GRAMMARS AND RELATIONS BETWEEN PRECEDENCE LANGUAGES

Abstract approved: [redacted]
Curtis R. Cook

Context-free grammars are adequate for specifying the structure of almost all programming languages. But in such specifications, it is difficult and time consuming to analyze a statement in the programming language to recognize its structural components. Precedence grammars, a subset of the context-free grammars, were developed for quick and easy statement analysis and yet are adequate for describing the structure of most programming languages. We present a historical development of precedence grammars and show some of the formal language relations between the various classes of precedence languages.

Precedence Grammars and Relation Between Precedence Languages

by

Yoshiyuki Yamagishi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

June 1972

APPROVED:

[redacted]
Assistant Professor of Mathematics

in charge of major

Acting Chairman of Department of Mathematics

Dean of Graduate School

Date thesis is presented December 10, 1971

Typed by Clover Redfern for Yoshiyuki Yamagishi

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to Dr. Curtis R. Cook, my major professor, for his continuous encouragement and guidance and for the useful discussion and the considerable time he spent with me during the preparation of this thesis.

I would also like to thank Dr. Paul Cull for his helpful advice and for the time he donated to me both officially and unofficially.

Acknowledgments are also made to Dr. and Mrs. Harry E. Goheen, my host family, for their wonderful friendship. Also I wish to thank Dr. Goheen for being my initial advisor.

Finally, I wish to thank the many people who contributed to my advanced study and thesis preparation at Oregon State University.

TABLE OF CONTENTS

LIST OF FIGURES

PRECEDENCE GRAMMARS AND RELATION BETWEEN PRECEDENCE LANGUAGES

I. INTRODUCTION

Almost all commonly available translators or interpreters of high level programming languages have been specifically designed for a particular language. The natural extension to a translator or interpreter system which could accept programs written in many languages and output the appropriate (machine) code has been the focus of much current research. A first step in this direction was the use of a context-free grammar to specify the structure or syntax of programming languages. But this was not enough for a complete specification of the programming languages as a complete specification requires meaning to be associated with the syntax. A system which allows the description of the syntax and semantics of a programming language is called a syntax-directed system.

The major components of a syntax-directed system are the parser and the interpreter. The parser analyzes the source program statement to deduce its structure and the interpreter attaches meaning to the output of the parser. In this thesis we will concentrate on precedence grammars, a subclass of context-free grammars, designed for rapid parsing and yet adequate for the syntactic specification of most programming languages. We study these languages as

subclasses of deterministic languages and show relations between them.

The definitions and terminology used in this thesis are described in Chapter II. Chapter III is a historical survey of precedence and related grammars. In Chapter IV, two recently developed precedence grammar techniques are described. In Chapter V, we show relations between various classes of precedence grammars.

II. DEFINITION AND TERMINOLOGY

Most of the following definitions and terminology are taken from [1].

An alphabet or vocabulary V is any finite set of symbols. Let V^* denote the (countably infinite) set of all finite strings of symbols from alphabet V. Any finite length string of symbols from the alphabet is called a sentence. The empty sentence, ϵ, is the sentence consisting of no symbols. We use V^+ to denote the set V^* minus the empty sentence. A language over a vocabulary V is a subset of V^*. A grammar is a generative specification of a language.

A context-free grammar (CFG) is a 4-tuple $G = (V_N, V_T, P, S)$, where V_N is a finite set of symbols called nonterminals or variables, V_T is a finite set of symbols distinct from those in V_N called terminals, i.e., $V_N \cap V_T = \phi$, $V_N \cup V_T = V$, P is a finite set of productions or rewiting rules. Each production being an ordered pair (A, ω), normally we write $A \rightarrow \omega$, where $A \in V_N$ and $\omega \in V^*$. For a production $A \rightarrow \omega$, A is called the left side and ω the right side of the production. S is the distinguished member of V_N called the starting symbol or root.

In this thesis we adopt the notational convention that, unless otherwise stated, Latin capital letters (e.g., A, B, C, ...) stand for

nonterminals, lower case letters (e.g., a, b, c, . . .), digits and special symbols (e.g., +, -, (,), *, etc.) stand for terminals, and lower case Greek letters denote string of terminal and nonterminal symbols. A terminal string is one consisting entirely of terminals. The length of a string α (the number of symbols in α) is denoted by $|\alpha|$. The notation α^n will stand for the n-fold concatination of a string α with itself. $\alpha^0 = \epsilon$ and $\alpha^n = \alpha\alpha^{n-1}$.

The relation $\omega \Rightarrow z$ holds (with respect to a set of productions, P) if there are strings α, β and μ and a variabls A such that $\omega = \alpha A \beta$, $z = \alpha\mu\beta$ and $A \rightarrow \mu$ is a production in P. We say ω directly derives z and conversely z directly reduces to ω for this relation. The relation $\omega \overset{*}{\Rightarrow} z$ holds (with respect to a set of productions) if there is a finite sequence of strings $\omega_0, \omega_1, \omega_2, \ldots, \omega_n$ $(n \geq 1)$ such that $\omega = \omega_0$, $\omega_n = z$, and $\omega_{i-1} \Rightarrow \omega_i$ for $0 \leq i \leq n$; $\omega = \omega_0 \Rightarrow \omega_1 \Rightarrow \omega_2 \Rightarrow \ldots \Rightarrow \omega_n = z$. We say ω derives z (z is a derivation from ω) and conversely z reduces to ω (ω is a reduction from z). We write $\omega \overset{*}{\Leftarrow} z$ to mean $\omega = z$ or $\omega \overset{*}{\Rightarrow} z$. We choose as our canonical derivation the right derivation, i.e., the derivation in which each step is of the form $\rho A \beta \rightarrow \rho\omega\beta$, where β is a terminal string.

A sentential form is any string derivable from S. The language defined by context-free grammar G is $L(G) = \{\omega \in V_T^* \mid S \overset{*}{\Rightarrow} \omega\}$.

Although a context-free grammar may have useless productions,

we assume that every grammar in this paper has no useless productions; i.e., we assume that for each production $A \rightarrow \omega$, there exists a derivation $S \overset{*}{\Rightarrow} \sigma A \beta \Rightarrow \sigma \omega \beta \overset{*}{\Rightarrow} \sigma z \beta$, where σ, z and β are (possible empty) terminal strings. There is an effective procedure for finding an equivalent grammar with no useless production [9]. Also we assume every context-free grammar has exactly one starting symbol S which can appear only on the left side of the production.

Example. Let CFG $G = (V_N, V_T, P, S)$, where $V_N = \{S, N, D, M\}$ $V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $P = \{S \rightarrow M$, $M \rightarrow MD$, $M \rightarrow N$, $D \rightarrow N$, $D \rightarrow 0$, $N \rightarrow 1$, $N \rightarrow 2$, $N \rightarrow 3$, $N \rightarrow 4$, $N \rightarrow 5$, $N \rightarrow 6$, $N \rightarrow 7$, $N \rightarrow 8$, $N \rightarrow 9\}$. The grammar G defines the language

$$L(G) = \{\text{all unsigned integers without leading zero}\}.$$

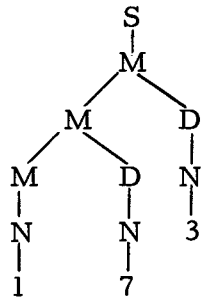
In general, there may be several different derivations of the same terminal string, since the order of substitution in the string may vary. For example we see two different derivations which derive the same integer number in grammar G.

$$S \Rightarrow M \Rightarrow MD \Rightarrow MDD \Rightarrow NDD \Rightarrow 1DD \Rightarrow 1ND \Rightarrow 17D \Rightarrow 17N \Rightarrow 173 \quad (1.1)$$

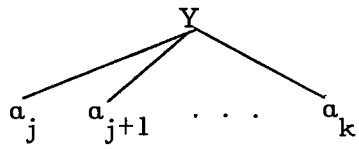
$$S \Rightarrow M \Rightarrow MD \Rightarrow MN \Rightarrow M3 \Rightarrow MD3 \Rightarrow MN3 \Rightarrow M73 \Rightarrow N73 \Rightarrow 173 \quad (1.2)$$

The sequence of derivation in (1.1) is substitution for the leftmost

nonterminal and in (1.2) is substitution for the rightmost, i.e., canonical derivation. We may represent any sentential form in at least one way as a derivation tree or syntax tree reflecting the steps in its derivation. For example above two derivations can be represented by:



The handle of the tree is the leftmost set of adjacent leaves forming a complete branch. In other words, if $a_1, a_2, \ldots, a_t$ are the leaves of the tree (where $a_i \in V$), we look for the smallest k such that the subtree has the form



for some j and Y. For a string $\alpha = a_1 a_2$, $H(\alpha) = a_1$ is called the head of α and likewise $T(\alpha) = a_2$ is called the tail of α, where $|\alpha| = |a_1| + |a_2|$. We may use subscripts to indicate length of the head and tail of a string α, i.e., $H_i(\alpha)T_{n-i}(\alpha) = \alpha$ and $H_0(\alpha)T_n(\alpha) = \alpha$ $H_n(\alpha)T_0(\alpha) = \alpha$, where $n = |\alpha|$.

Loosely speaking, a parse of a sentential form is some indication of how that string was derived. Although a grammar is the means

of generating a language, it may be used to recognize the sentence. For the recognition procedure of a language, we need a description of the language which is provided by a grammar which defines the language. A parsing algorithm is a recognition procedure which recognizes sequences of successively high level substructures until it finally can determine whether or not a given string is in the language. There are three general types of parsing algorithms. One type is bottom-up which starts with terminals and works up the syntax tree. The second is top-down which starts with the entire sentence at the top of the tree and works down to the terminals. The third type is a combination of the top-down and bottom-up. The class of grammars described in this thesis is designed for bottom-up parsing. A canonical parse is a parse which scans the sentential form α from left to right and obtains in reverse order the sequence of productions used in a canonical derivation of α. e.g.,

$173 <= N73 <= M73 <= MN3 <= MD3 <= M3 <= MN <= MD <= M <= S$

Clearly every parse has only one canonical parse, but not conversely. A sentential form may have more than one derivation. A sentence having no unique canonical parse in a given grammar is said to be ambiguous. A sentence is unambiguous if it has unique canonical parse. A grammar is ambiguous if some sentence is ambiguous with respect to it and conversely, a grammar is unambiguous if each of its sentences is unambiguous.

III. PRECEDENCE AND BOUNDED CONTEXT GRAMMARS

Although context-free grammars are adequate for describing the syntax of programming languages, they may be ambiguous and parsing algorithms are very slow and inefficient. Restricting the grammar to an unambiguous subclass of CFG which are adequate for describing most computer languages avoids the ambiguity problem. Two general schemes for making the parsing faster and more efficient are the precedence techniques and the local context techniques. The operator precedence grammars [6], precedence grammars [17] and extended precedence grammars [4] are examples of the first scheme and transition matrix [4]. (m, n) bounded context grammars [3, 5, 12, 17] and LR(k) grammars [1, 13] are examples of second scheme. Also there are some studies of combinations of the two types [15].

This chapter is a historical survey of precedence and related grammars. The practical techniques for parsing are described in Section 1 to 4. A practical technique is one that has been or is being used to write a compiler. Sections 1 to 3 also provide the background for right and weak precedence grammars which will be introduced in the next chapter. In Sections 5 and 6, more general, powerful and complicated grammars are described.

3.1. Operator Precedence Grammars [6]

An operator grammar is a CFG in which no production has a form $U \rightarrow \alpha U_1 U_2 \beta$, where $U_1, U_2 \in V_N$.

In an operator grammar, there are three relations some or all of which may hold between terminal symbols T_1 and T_2.

(1) $T_1 \doteq T_2$ if there is a production $U \rightarrow \alpha T_1 T_2 \beta$ or $U \rightarrow \alpha T_1 U_1 T_2 \beta$, where $U_1 \in V_N$.

(2) $T_1 \gtrdot T_2$ if there is a production $U \rightarrow \alpha U_1 T_2 \beta$ and a derivation $U_1 \stackrel{*}{=>} \gamma$, where $U_1 \in V_N$ and T_1 is the rightmost symbol of γ.

(3) $T_1 \lessdot T_2$ if there is a production $U \rightarrow \alpha T_1 U_1 \beta$ and a derivation $U_1 \stackrel{*}{=>} \gamma$, where $U_1 \in V_N$ and T_2 is the leftmost terminal symbol of γ.

If at most one relation holds between any ordered pair T_1, T_2 of terminal symbols, then the grammar is called an operator precedence grammar.

To parse of a string, the relations $\lessdot$, $\doteq$ and $\gtrdot$ between terminals are recorded in a $N_T \times N_T$ matrix, where N_T is the number of terminals of the grammar. The parse of a sentence is quite straightforward. Symbols $T_0, T_1, \ldots$ are pushed onto the stack until the relation $T_i \gtrdot T$ holds between the top stack terminal symbol T_i and the terminal symbol T just read. Search the

stack for first occurrence of the relation $T_j \lessdot T_{j+1}$ for some $0 \leq j < i$. Then $T_{j+1} \cdots T_i$ is a handle. Pop the handle from the stack and place on the stack the leftside of the production whose right-side matches the handle. Repeat this process comparing top stack symbol and T.

Example [11]. $G = (\{S, E, A, B\}, \{+, *, (,), a, \perp\} P, S\})$,

$P = \{S \rightarrow \perp E \perp, E \rightarrow E+A, E \rightarrow A, A \rightarrow A*B, A \rightarrow B, B \rightarrow (E), B \rightarrow a\}$,

where $\perp$ is special terminal start and end marker. The relations $\gtrdot$, $\doteq$ and $\lessdot$ can be kept in an $N_T \times N_T$ matrix as Figure 1, where $\odot$ means no relation

T_1 \ T_2	(	a	*	+	)	$\perp$
)	$\odot$	$\odot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
a	$\odot$	$\odot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
*	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\odot$
$\perp$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\odot$	$\doteq$

Figure 1. Operator precedence matrix.

An error will be detected whenever no relation $\odot$ holds between the top stack terminal symbol and the incoming terminal symbol. For a handle α, there may be more than one nonterminal to which it may be reduced, since there is no restriction that the right side of production be unique. This problem will be passed to the

semantic routine for appropriate action. Another objection is that the parsing process stops when the sentential form consists entirely of nonterminals.

This technique has been used in quite a few ALGOL, MAD and FORTRAN compilers. The technique is very easy to understand, flexible and efficient.

3.2. Precendence Grammars (Simple Precedence Grammars) [17]

Wirth and Weber generalized Floyd's operator precedence grammar to include relations between all pairs of symbols, terminals and nonterminals. The following precedence relations may hold between any pair of symbols $S_1, S_2 \in V$.

(1) $S_1 \doteq S_2$ if and only if there exists a production of the form $U \rightarrow \alpha S_1 S_2 \beta$.

(2) $S_1 \lessdot S_2$ if and only if there exists a production of the form $U \rightarrow \alpha S_1 U_1 \beta$ and there exists a derivation $U_1 \overset{=}{>}{}^{*} S_2 \gamma$.

(3) $S_1 \gtrdot S_2$ if and only if there exists a production of the form $U \rightarrow \alpha U_1 S_2 \beta$ (or $U \rightarrow \alpha U_1 U_2 \beta$) and there exists a derivation $U_1 \overset{*}{\Rightarrow} \gamma S_1$ (and $U_2 \overset{*}{\Rightarrow} S_2 \delta$).

A context-free grammar G satisfying the following two conditions, is called a simple precedence grammar.

(1) There are no productions such that

$$A \rightarrow \alpha, \quad B \rightarrow \alpha, \quad \text{where} \quad A \neq B, \quad \alpha \in V^{*}.$$

(2) There is a unique prcedence relation between A and B such that

$$S \overset{*}{\Rightarrow} \alpha AB\beta, \quad \text{where} \quad A, B \in V.$$

To parse a string, we store the precedence relations in a matrix that has as many row and columns as symbols in the vocabulary V and add a special terminal symbol "$\perp$" which marks the ends of the string. The parse is very similar to Floyd's. Push symbols $T_0 T_1 \cdots$ onto a stack until the relation $T_i \gtrdot T$ holds between top stack symbol T_i and terminal symbol T just read. Search the stack for the first occurrence of the relation $T_j \lessdot T_{j+1}$ for $0 \leq j < i$. Thus $T_{j+1} \cdots T_i$ is a handle. Replace the handle with leftside of unique production $U \rightarrow T_{j+1} \cdots T_i$ and place U on the stack. Repeat the process comparing U and T. If the relation $\odot$ (no relation) holds between the top stack symbol and T, then stop with error.

As an example [11], let $G = (\{S, A, B\}, \{a, b_1, b_2, \perp\}, P, S)$, $P = \{S \rightarrow \perp A \perp, A \rightarrow Bb_2, B \rightarrow b_1, B \rightarrow Ba, B \rightarrow BA\}$. Then the matrix is given by Figure 2. The matrix is much larger than Floyd's, since a relation may hold between any two symbols.

The above parsing algorithm always yields the unique canonical parse for a sentence which belongs to the simple precedence language because a handle is uniquely determined, and hence the

production to be applied is unique [17]. Thus theoretically, the technique is very sound and efficient, and it is more reliable than Floyd's, since the relation may hold between any two symbols.

S_1 \ S_2	A	B	a	b_1	b_2	⊥
A	⋗	⋗	⋗	⋗	⋗	≐
B	≐	⋖	≐	⋖	≐	⊙
a	⋗	⋗	⋗	⋗	⋗	⊙
b_1	⋗	⋗	⋗	⋗	⋗	⊙
b_2	⋗	⋗	⋗	⋗	⋗	⋗
⊥	≐	⋖	⊙	⋖	⊙	⊙

Figure 2. Simple precedence matrix.

3.3. Extended Precedence Grammars [4]

McKeeman extended Wirth and Weber's technique by separating the precedence matrix into two tables and by looking at more context so that fewer precedence conflicts arise. Therefore his technique will accept a larger class of context-free grammars.

In Wirth and Weber's algorithm, we only need to find a ⋗ relation when looking for the rightmost symbol of a handle. Also the next symbol read is always a terminal symbol. Thus to find the rightmost symbol of a handle, we need an $N_T \times N$ matrix, where N_T is the number of terminal symbols and N is number of symbols. To find the leftmost symbol of a handle, we need $N \times N$ matrix which is the

same size as Wirth and Weber's.

Right precedence relation (the relation to find a rightmost symbol of handle) can be defined as follows.

Let $L(U) = \{A \mid U \overset{*}{\Rightarrow} A\alpha\}$

$R(U) = \{A \mid U \overset{*}{\Rightarrow} \alpha A\}$

$L_T(U) = \{U\}$ for $U \in V_T$ or

$L_T(U) = \{A \mid U \overset{*}{\Rightarrow} A\alpha, A \in V_T\}$ for $U \in V_N$

then for $A \in V$ and $\alpha, \beta \in V^*$.

(1) $S_1 \lessdot S_2$ if and only if there exists a production of the form $U \rightarrow \alpha A_1 U_1 \beta$ and $S_2 \in L_T(U_1)$.

(2) $S_1 \gtrdot S_2$ if and only if there exists a production of the form $U \rightarrow \alpha U_1 U_2 \beta$ and $S_1 \in R(U_1)$ and $S_2 \in L_T(U_2)$.

Left precedence relation (the relation to find a leftmost symbol of a handle) can be defined as

(1) $S_1 \doteq S_2$ if and only if there exists a production of the form $U \rightarrow \alpha S_1 S_2 \beta$.

(2) $S_1 \gtrdot S_2$ if an only if there exists a production of the form $U \rightarrow \alpha S_1 U_1 \beta$ and $S_2 \in L(U_1)$.

If the precedence relation cannot be determined uniquely, such as $U \rightarrow \alpha S_1 U_1 \beta$ and $S_1 \in R(S_1)$, $S_2 \in L_T(U_1)$, then conflict relation $\gtrless$ can be entered into both matrices. The conflict will be

resolved by the parser examining more context.

As an example [11], we have two matrices (let us call them the right precedence table and the left precedence table) for the grammar used in the previous section. No conflicts occur in this grammar. See Figure 3.

S_1 \ S_2	a	b_1	b_2	⊥
A	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$
B	$\doteq$	$\doteq$	$\doteq$	⊙
a	$\gtrdot$	$\gtrdot$	$\gtrdot$	⊙
b_1	$\gtrdot$	$\gtrdot$	$\gtrdot$	⊙
b_2	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
⊥	⊙	$\doteq$	⊙	⊙

Right precedence table

S_1 \ S_2	A	B	a	b_1	b_2	⊥
A	⊙	⊙	⊙	⊙	⊙	$\doteq$
B	$\doteq$	$\lessdot$	$\doteq$	$\lessdot$	$\doteq$	⊙
a	⊙	⊙	⊙	⊙	⊙	⊙
b_1	⊙	⊙	⊙	⊙	⊙	⊙
b_2	⊙	⊙	⊙	⊙	⊙	⊙
⊥	$\doteq$	$\lessdot$	⊙	$\lessdot$	⊙	⊙

Left precedence table

Figure 3. Extended precedence matrix.

The parsing algorithm uses Wirth and Weber's two arguments precedence relation as long as possible, and if a precedence conflict arises (a) the top two stack symbols T_{i-1}, T_i and the T just read are used to decide whether T should be put onto the stack or whether T_i is the rightmost symbol of a handle and reduction should take place, or, (b) similarly, in order to go back in the stack to find the rightmost symbol of a handle, three symbols instead of two are used. When looking to the right on the string using the right precedence table if $\gtrdot$ relation holds, a list of triples is searched to find

the value of the following three-argument function P1 and take the proper action according to the value

$$P1(T_{i-1}, T_i, T) = \begin{cases} \text{true} \ldots T_i \gtrdot T & (T_i \text{ is tail of a handle}) \\ \text{false} \ldots T_i \lessdot T & (T_i \text{ is not tail of a handle}); \end{cases}$$

when looking to the left on the string (go back in the stack), the left precedence table is used and if $\lessdot$ relation holds, the function P2 is used

$$P2(T_{j-1}, T_j, T_{j+1}) = \begin{cases} \text{true} \ldots T_{j-1} \lessdot T_j & (T_j \text{ is a head of a handle}) \\ \text{false} \ldots T_{j-1} \doteq T_j & (T_j \text{ is not a head of a handle}). \end{cases}$$

McKeeman has written a complier for a dialect of PL/1 on the IBM 360 using this technique. The matrix are about 90 x 45 and 90 x 90, while roughly 450 triples are necessary [4].

3.4. Transition Matrix [4]

This parsing technique for operator grammars was first introduced by Samelson and Bauer in 1960 [4]. This method uses two stacks, operand stack and operator stack. The string is searched from left to right distinguishing between identifiers and operators. Incoming identifiers are pushed onto the operand stack, while an incoming operator is compared with the top symbol of the operator

stack. If the reduction cannot be performed, the incoming operator is pushed onto the operator stack. If a reduction can be performed, a subroutine is called to perform the appropriate reduction and place the result onto the operand stack.

The difference between this method and the operator precedence technique is that this method uses a matrix of subroutine names instead of a precedence matrix. The top operator stack element and incoming symbol determine element of the matrix which is a name of subroutine to execute; this subroutine then performs the necessary reduction or pushes the symbol onto the operand or operator stack. Therefore the productions do not have to be searched at each step to determine the reduction to make.

A matrix for ALGOL is about 50 x 40, with about 500 subroutines [4]. This is perhaps the fastest technique but its drawbacks are the space used and large number of subroutines needed to implement it. Operator precedence and transition technique are the most well known and have undoubtedly been used in many other compilers.

3.5. (m, n) Bounded Context Grammars [3, 5, 12]

A grammar is an (m, n) bounded context grammar ((m,n)BCG) if and only if a handle is uniquely determined by the m symbols to its left and the n symbols to its right. Thus at each step, the parser searches a sentential form from left to right for a handle by considering at

most m symbols to the left on the stack and n terminals to the right of a possible handle. The operator precedence, simple precedence and extended precedence grammars are essentially $(1,1)$ bounded context grammars.

Parsing an (m,n) BCG for $m > 1$, $n > 1$ make unreasonable demands on computer time and storage space. Therefore (m,n)BCG has not been used in an actual compiler. Eickel [3] gave a $(1,1)$BCG parsing algorithm in 1963. He rewrote the productions so that every production has at most two symbols on the right side. Then the stack's top two symbols T_i and T_{i-1} and the incoming terminal T must uniquely determine the step to be taken. Thus for each triple (T_{i-1}, T_i, T) one and only one of the following conditions must hold.

(1) T_1T_2 is a handle and reduction $U \leftarrow T_1T_2$ may be executed.

(2) T_2 is a handle and reduction $U \leftarrow T_2$ may be executed.

(3) T must be pushed onto the stack.

(4) T_1T_2 may not be appear as a correct substring of a sentential form.

This algorithm has been programmed and tested but not used to write a compiler.

3.6. (m,n) Precedence Grammars [17]

In their paper [17] Wirth and Weber suggested extending idea

of a precedence relation between two symbols S_1 and S_2 to a precedence relation between two strings α and β. Thus we have $\doteq$, $\lessdot$ and $\gtrdot$ relations between two strings α and β, where $|\alpha| \leq m$ and $|\beta| \leq n$. The (m,n) precedence grammar is of course also (m,n) bounded context grammar, and the simple precedence grammar in Section 3.2 is (1.1) precedence grammar [8].

In operator precedence, precedence and extended precedence techniques, each time a handle is recognized, the productions must be searched to find the productions whose right sides match the handle. Therefore, all these methods are slow unless a quick and efficient table search can be performed. Also the operator precedence technique requires an $N \times N$ matrix for the precedence relations, and extended precedence require $N \times N$ and $N \times N_T$ matrices. We need two bits of computer storage to represent the elements of precedence matrix. This turns out to be quite a sizeable amount of computer storage. For instance, the ALGOL 60 report [16] has about 120 nonterminals and about 110 terminals. Hence an operator precedence matrix for ALGOL 60 requires 105, 800 bits. One other drawback of precedence grammars is that most syntax specifications of programming languages must be substantially altered to put them into proper form.

IV. RIGHT PRECEDENCE AND WEAK PRECEDENCE GRAMMARS

We are always interested in developing grammars with a more efficient parsing technique, where efficient means (a) rapidly parsed, (b) small computer storage usage, and (c) grammar generates a wider class of languages. In this chapter we demonstrate that we do not need a left precedence table to parse simple precedence grammars. This technique is a more efficient one in the sense of (a) and (b). A similar technique was used in developing right precedence grammars and weak precedence grammars. Both of these grammars are an extension of the simple extended precedence grammars. After the right precedence grammar and weak precedence grammar are defined we show that they generate a larger class of languages than simple precedence languages.

4.1. Improved Simple Precedence Technique

Recall that Wirth and Weber recorded the precedence relations in a table. We can think of the table as two tables, one for looking for the rightmost symbol of the handle and the other for the leftmost symbol of the handle. Now we will show that we can abolish the left precedence portion of this table and thereby decrease the size of this table.

Before defining a right precedence grammar, let us show that

for a simple precedence grammar we do not need a left precedence table. The handle turns out to be the longest right side of a production that matches the contents of the top part of the stack. We use the right precedence table to find the rightmost symbol of the handle. Suppose we find the rightmost symbol of the handle at the position S_i in the sentential form $S_0S_1 \cdots S_nS_{n+1}$, i.e., $S_i \gtrdot S_{i+1}$, with $S_{i+1}, \ldots, S_{n+1} \in V_T$, $S_0, S_1, \ldots, S_i \in V$. Let $A \rightarrow \alpha$ be the production with the longest right side that matches the contents of the stack where α has the form $S_pS_{p+1} \cdots S_{i-1}S_i$. If this is the only production whose right side matches the stack contents, then α is the handle and a reduction can be performed. Suppose there is another production $B \rightarrow \sigma$ whose right side matches the contents of the stack. Then σ must be of the form $S_qS_{q+1} \cdots S_{i-1}S_i$ for $q > p$. If σ is a handle $S_{q-1} \lessdot S_q$ must hold. But $A \rightarrow \alpha$ implies $S_{q-1} \doteq S_q$ contradicting the fact that grammar is a simple precedence grammar. Thus $S_pS_{p+1} \cdots S_{i-1}S_i$ must be a handle. Hence parsing simple precedence languages using this method requires only a right precedence matrix ($N \times N_T$ matrix size) and the handle corresponds to the longest right side of a production matching the top stack elements. Therefore this method of parsing precedence languages is faster and requires less space than Wirth and Weber's simple precedence parsing technique.

4.2. Right Precedence Grammars [11]

In this section we will define right precedence grammars and show that they correspond to the improved technique of the previous section. We also show that simple precedence grammars are a proper subclass of right precedence grammars.

Let $G = (V_N, V_T, P, S)$ be a context-free grammar. When G satisfies the following condition, then we say G is a right precedence grammar (RPG).

(1) There are no productions such that

$A \rightarrow \alpha$, $B \rightarrow \alpha$, where $A \neq B$ and $\alpha \in V^*$.

(2) There exists a unique right precedence relation (refer to Chapter III, Sec. 3) between two symbols A and B such that

$S \overset{*}{\Rightarrow} \alpha AB\beta$, where $A, B \in V$ and $\alpha, \beta \in V^*$.

(3) If there exists productions such that

$A \rightarrow \alpha\beta$, $B \rightarrow \beta$

then there exists no productions such that

$C \rightarrow \gamma H_i(\alpha) D\delta$, $D \overset{*}{\Rightarrow} T_{n-i}(\alpha) B\delta'$, where $n = |\alpha|$

$n \geq i > 0$ and $\alpha, \beta, \gamma, \delta, \delta' \in V^*$.

To parse a string, the right precedence relation $\lessdot$ and $\gtrdot$ are recorded in $N \times N_T$ matrix. The parse of a sentence is very simple. Symbols $S_1 S_2 \ldots$ are pushed onto the stack until the

relation $S_i \gtrdot S$ holds between the top stack symbol S_i and the symbol S just read. Find the production $U \rightarrow U_j \cdots U_1$ with longest right side such that $S_i = U_1$, $S_{i-1} = U_2, \ldots, S_{i-j+1} = U_j$, i.e., $S_{i-j+1} \cdots S_{i-1} S_i$ is the handle. Replace the handle with leftside of the unique production $U \rightarrow U_j \cdots U_1$ and place U on the stack. Repeat the process comparing the top stock U with S. An error will be detected whenever no relation $\odot$ holds between the top stack symbol and incoming symbol and no production matches top stock elements. The flow chart for parsing is given in Figure 4.

When $S_i \gtrdot S$ relation occurs during parsing, ambiguity is possible only if two or more productions have right parts matching the top of stack. However, restrictions (1) and (3) make it impossible to have two derivations of the same sentential form. Thus a right precedence grammar is unambiguous.

Suppose we have productions such that

$$A \rightarrow \alpha\beta, B \rightarrow \beta, \alpha, \beta \in V^+$$

and $T(\alpha)\beta$ matches the top stack elements $S_{i-p} \cdots S_i$, where $T(\alpha) \neq \alpha$, $T_1(\beta) = S_i$, $\alpha\beta$ does not match the top stack elements, and $S_i \gtrdot S_{i+1}$. Then above parsing method allows us to pick up β for the handle and a reduction can be performed. This is not allowed in a simple precedence grammar because the two productions imply that

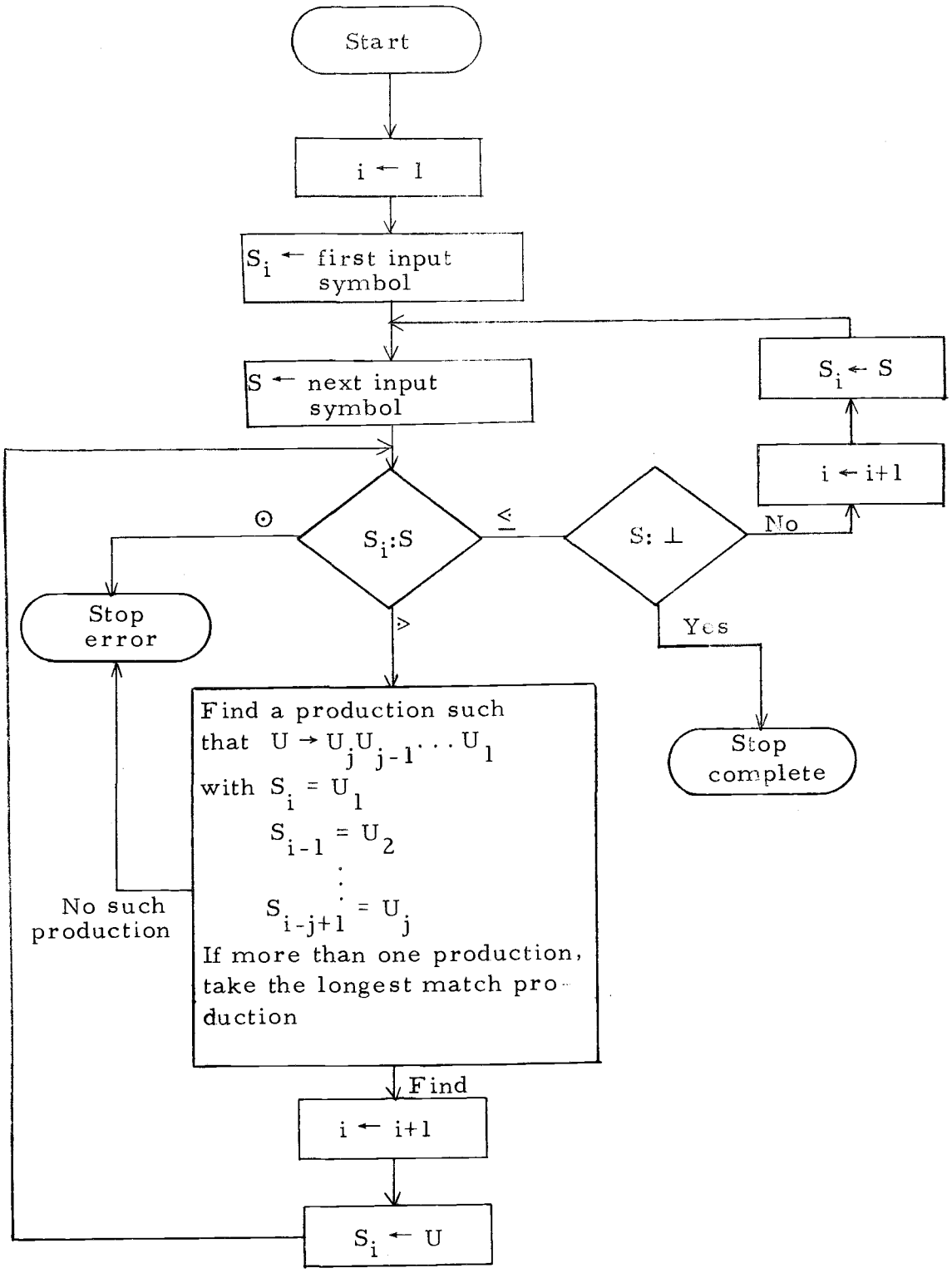


Figure 4. Flow chart for right precedence parser.

$T_1(\alpha) \doteq H_1(\beta)$ and $T_1(\alpha) \lessdot H_1(\beta)$ and only one relation can hold between any two symbols in a simple precedence grammar. Also if restriction (3) is allowed, $H_i(\alpha) \lessdot T_{n-i}(\alpha)$ must hold; however we have $H_i(\alpha) \doteq T_{n-i}(\alpha)$ from condition $A \rightarrow \alpha\beta$. This is a contradiction. Restriction (1) is the same as (1) of simple precedence grammar, and (2) of RPG is a weaker restriction than (2) of the simple precedence grammar. This observation results lead us to the following theorem:

Theorem 4.1. Every simple precedence grammar is a right precedence grammar.

As an example, let $G = (\{S, E, A, B\}, \{+, *, (,), a, \perp\}, P, S)$ where $P = \{S \rightarrow \perp E \perp, E \rightarrow E+A, E \rightarrow A, A \rightarrow A*B, A \rightarrow B, B \rightarrow (E), B \rightarrow a\}$. Although G is not a simple precedence grammar, it is the right precedence grammar [11]. See Figure 5.

4.3. Weak Precedence Grammars [10]

Ichbian [10] introduced the weak precedence grammar (WPG). Weak precedence grammars are generalizations of the simple precedence grammars and are very similar to right precedence grammars. They define two weak precedence relations "<" and ">" which may hold between any two symbols. The relation "<" is the same as relations $\lessdot$ and $\doteq$ in simple precedence grammars, and

S_1 \ S_2	+	*	(	)	a	⊥
E	$\lessdot$	⊙	⊙	$\lessdot$	⊙	$\lessdot$
A	$\gtrdot$	$\lessdot$	⊙	$\gtrdot$	⊙	$\gtrdot$
B	$\gtrdot$	$\gtrdot$	⊙	$\gtrdot$	⊙	$\gtrdot$
+	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙
*	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙
(	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙
)	$\gtrdot$	$\gtrdot$	⊙	$\gtrdot$	⊙	$\gtrdot$
a	$\gtrdot$	$\gtrdot$	⊙	$\gtrdot$	⊙	$\gtrdot$
⊥	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙

Right precedence table

S_1 \ S_2	E	A	B	+	*	(	)	a	⊥
E	⊙	⊙	⊙	$\doteq$	⊙	⊙	$\doteq$	⊙	$\doteq$
A	⊙	⊙	⊙	⊙	$\doteq$	⊙	⊙	⊙	⊙
B	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙
+	⊙	$\lessdot$	$\lessdot$	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙
*	⊙	⊙	$\doteq$	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙
(	$\lessdot$	$\lessdot$	$\lessdot$	⊙	⊙	$\lessdot$	⊙	$\lessdot$	⊙
)	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙
a	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙	⊙
⊥	$\lessdot$	$\lessdot$	$\lessdot$	⊙	⊙	$\lessdot$	⊙	⊙	⊙

Left precedence table

There are conflicts in left precedence table with element (⊥, E), ((, E) and (+, A). The left precedence table is presented for reference.

Figure 5. Precedence matrix for right precedence parser.

relation $>$ same as $\gtrdot$.

A weak precedence grammar is defined as follows.

(1) There is at most one weak precedence relation between any two symbols of the vocabulary V.

(2) There exist no two productions with the same right side.

(3) Whenever there are two productions of the form, $U_1 \rightarrow xSy$ (where the string x may be empty) and $U_2 \rightarrow y$, there is no weak precedence relation between S and U_2.

The parsing algorithm for weak precedence grammars is exactly the same as the right precedence algorithm. The handle is determined by the pattern matching, that is the handle corresponds to the longest right part of a production that matches to the top elements of the stack. The longest match rule holds in this parsing method the same as the right precedence parsing method because of definition (3). Although the top stack element matches to two or more productions right side, the definition (3) does not allow to apply shorter production.

Two theorems in [10] show that a weak precedence grammar is unambiguous and that every simple precedence grammar is a weak precedence grammar.

Definition (1) implies that we need $N \times N$ precedence matrix size for a parser. However, Ichbian also pointed out that the relevant portion of the precedence matrix is exactly same as a right precedence table.

V. FORMAL LANGUAGE RELATIONS

In this chapter, we will define deterministic and nondeterministic pushdown accepters, deterministic languages, and LR(k) grammars and show the relations between them. We show that every right precedence language is a deterministic language. Also we expand and update an inclusion tree for classes of precedence and bounded context grammars which appears in [4].

5.1. Nondeterministic and Deterministic Pushdown Automata and Relation to Context-Free Grammars

A nondeterministic pushdown automaton (NPDA) is very important device for formalization of languages. The NPDA is essentially a finite automaton an input tape (with left to right search) and pushdown store (with first in last out rule).

A pushdown automaton M is defined as follows:

$$M = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

(1) K is a finite set of states $\{q_0, q_1, q_2, \ldots, q_n\}$,

(2) Σ is a finite set of inputs (correspond to terminal symbols),

(3) Γ is a finite set of pushdown alphabet (correspond to nonterminal symbols),

(4) q_0 in K is the initial state,

(5) Z_0 in Γ is a particular pushdown symbol called the start symbol. Z_0 appears initially on the pushdown store,

(6) F contained in K is the set of final states,

(7) δ is a mapping $\delta:(K \times (\Sigma \cup \epsilon) \times \Gamma) \rightarrow (K \times \Gamma^*)$.

At each step we have a triple (q_i, a_j, Z) where q_i is a state i, a_j is an input symbol, and Z is the top pushdown store symbol. The initial step must be (q_0, a_1, Z_0). With the help of mapping

$$\delta(q_i, a_j, Z) = \{(P_1, \gamma_1), (P_2\gamma_2) \ldots (P_n, \gamma_n)\}$$

where P_ℓ, $1 \leq \ell \leq m$ are in K, γ_ℓ is in Γ^*, $1 \leq \ell \leq m$ and state q_i can enter state P_ℓ, for some ℓ, replace Z by γ_ℓ and advance input head by one. When a_j is ϵ, the input head is not advanced.

The language can be accepted in two ways. For a pushdown automaton M we define $T(M)$, the language is accepted by final state as the set of strings that cause M to enter a final state when the whole string has been read. Also we define $N(M)$, the language accepted by empty store as the set of strings that cause M to empty its pushdown store when the whole string has been read. Two types of acceptance are equivalent [9].

Now let us see the relations between context-free grammars and nondeterministic pushdown automata.

Theorem 5.1. If L is a context-free language, then there exists a NPDA M such that L = T(M) [9].

Theorem 5.2. If L is accepted by some NPDA, then L is a context-free language [9].

Thus a class of languages accepted by NPDA is precisely the class of context-free language.

A deterministic pushdown automaton (DPDA) is a pushdown automaton which has only one choice of move for any triple. A language is a deterministic language if it is accepted by some DPDA. Deterministic languages are very important because they are unambiguous. Various classes of precedence grammars are unambiguous [4, 10, 11, 17]. We want unambiguous languages because meaning is associated with a parse and if language is ambiguous, more than one meaning can be associated with the sentence.

5.2. DPDA and Right Precedence Languages

Now let us show that a right precedence language is a deterministic language. The proof of this result consists of constructing from the right precedence grammar a DPDA which simulates the parsing process.

There are several practical techniques possible to find the longest match on the right side of the productions. Here we present a

list processing technique. We construct a binary tree with three fields in each node. The first field has a symbol which appears on the right side of a production. The second and third fields are individually LEFT LINK and RIGHT LINK. RIGHT LINK is used in case of symbol match. LEFT LINK is used when the symbol is not matched. The root of a tree, entry, must be the rightmost symbol of a production. Each branch has a nonterminal symbol of the corresponding production. Suppose we have productions with common rightmost symbol such as $P_1 \rightarrow A$, $P_2 \rightarrow BA$, $P_3 \rightarrow CBA$, $P_4 \rightarrow DBA$ and $P_5 \rightarrow CA$, and a unique production $P_6 \rightarrow DEF$. Then trees are represented on Figure 6.

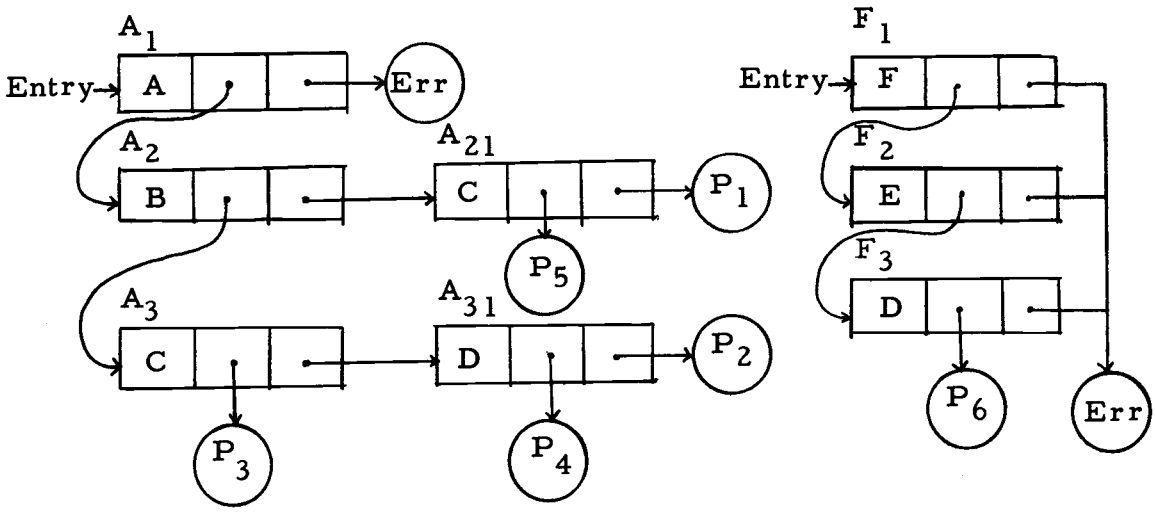


Figure 6. Binary tree presentation of the productions.

Suppose we have a sentential form $\cdots S_{i-1} S_i S_{i+1} \cdots$, and $S_i \gtrdot S_{i+1}$ holds during parsing, then jump to binary tree with root symbol S_i and start the matching process to the left on the sentential form working down the tree. We name each node as shown in Figure 6. The addresses have the common name which is an entry symbol. For the successor of LEFT LINK, increase the last subscript by one. For the successor of RIGHT LINK, add the new subscript to the right. The root always has subscript 1.

Let RPG $G = (V_N, V_T, P, S)$. Let us define DPDA $M = \{K, V_T, V, \delta, q_0, Z_0, \{q_F \cup q_{error}\}\}$, where $V = V_T \cup V_N$ from G.

(1) Initialization; read an input symbol and replace Z_0

$$\delta(q_0, V_t, Z_0) = (q_1, V_t)$$

for each $V_t \in V_T$.

(2) Input string and place input symbol on stack until $\gtrdot$ relation holds between input symbol and top stack symbol. For each $V_t \in V_T$, each $Z \in V$

$$\delta(q_1, V_t, Z) = \begin{cases} (q_1, ZV_t) & \text{if relation is } \lessdot \\ (q_{error}, \text{stop}) & \text{if relation is } \doteq \\ (q_{V_t Z_1}, Z) & \text{if relation is } \gtrdot \end{cases}$$

(3) Find the longest match process using binary tree. For each $V_t \in V_T$, and for each binary tree with entry symbol U and for all possible nodes

$$\delta(q_{V_t U_{ij\ldots k}}, \epsilon, Z) = \begin{cases} (q_{V_t U_{ij\ldots k+1}}, \epsilon) & \text{for the left link} \\ (q_{V_t U_{ij\ldots k1}}, Z) & \text{for the right link} \\ (q_{V_t}, P) & \text{for the left link which is end of match} \\ (q_{V_t}, ZP) & \text{for the right link which is end of match} \\ (q_{error}, \text{stop}) & \text{for left and right link in error} \end{cases}$$

where P is the left side of production and each subscript of U is greater than or equal to 1 and less than n for some n = the longest length of right side of production in some binary tree.

(4) Precedence relation comparision between replaced symbol and input symbol. For every $V_t \in V_T$ and every $Z \in V$, and for every entry U

$$\delta(q_{V_t}, \epsilon, Z) = \begin{cases} (q_1, ZV_t) & \text{for } Z \lessdot V_t \text{ and } V_t \neq \perp \\ (q_{S\ 1}, ZV_t) & \text{for } Z \lessdot V_t \text{ and } V_t = \perp \\ (q_{V_t U_1}, Z) & \text{for } Z \gtrdot V_t \\ (q_{error}, \text{stop}) & \text{for } Z \odot V_t \end{cases}$$

(5) Final state

$$\delta(q_S, \epsilon, Z) = (q_F, \text{stop})$$

It should be clear from the construction that $L(G)$ is set of words accepted by M. Thus we can say every right precedence language is a deterministic language.

5.3. LR(k) Grammars and Deterministic Languages [13]

LR(k) grammars were first developed by Knuth in 1965 [13]. A grammar is said to be translatable from left to right with bound k (LR(k) grammar) if and only if any handle is always uniquely determined by entire string to its left and k terminal symbols to its right. Suppose we have a string $X_1X_2\cdots X_nX_{n+1}\cdots X_{n+k}a$, where $X_{n+1}\cdots X_{n+k}a$ is a part of string we have not examined yet, but all possible reductions have been made at the left of string so that the right boundary of the handle must be at the position X_r $(r \geq n)$. Then by looking k terminal symbols ahead $X_{n+1}\cdots X_{n+k}$, we want to know if there is a handle whose right boundary is at position X_n, if so reduction will be performed with corresponding production, if not move to the right and read a new terminal character of string to be translated.

Each LR(k) grammar is clearly unambiguous, since every derivation tree must have the same handle, and have a canonical form. Furthermore nearly all known unambiguous context-free grammars are LR(k) grammars. Kunth [13] gave two algorithms for deciding whether or not a grammar is LR(k) for a given k. Thus the LR(k)

condition may be regarded as the most powerful general test for unambiguity that is now available.

He, also, proved a number of interesting consequences about the relationship between LR(k) grammars and DPDA or deterministic languages. Let us see these results.

Theorem 5.3. If G is an LR(k) grammar, then G is unambiguous [9].

Theorem 5.4. There is an algorithm to determine whether context-free grammar G is LR(k) for some k [9].

Theorem 5.5. If $G = (V_N, V_T, P, S)$ is a LR(k) grammar, then L(G) is a deterministic language [9].

Theorem 5.6. Every deterministic language is generated by some LR(1) grammar [9]. (Thus a language is LR(1) if and only if it is LR(k) for some k).

Moreover Hopcroft [9] proved,

Theorem 5.7. If L is a deterministic language, then there exists an LR(O) grammar G such that $L(G) = L\perp$, where $\perp$ is a special end marker [9].

In Chapter IV, we showed that RPG are unambiguous and that the right precedence languages are deterministic languages. Thus we have the following theorem.

Theorem 5.8. If $G = (V_N, V_T, P, S)$ be a RPG, then there is an LR(1) grammar G' which generates exactly $L(G)$.

Graham [8] obtained several interesting relations between (m, n) bounded context grammars, (m, n) precedence grammars and deterministic languages.

Theorem 5.9. Let G be an (m, n) precedence grammar. Then G is an (m, n) bounded context grammar.

Theorem 5.10. Let G be an (m, n) bounded context grammar. Then there is a $(1, 1)$ bounded context grammar G' which generate exactly $L(G)$.

Theorem 5.11. Let $G = (V_N, V_T, P, S)$ be a $(1, 1)$ bounded context grammar. Then G can be transformed into an equivalent $(2, 1)$ precedence grammar $G' = (V_N', V_t, P', S)$.

Theorem 5.12. Every deterministic language is generated by a $(2, 1)$ precedence grammar.

The next theorem follows from Theorem 5.12 and the result that every right precedence language is a deterministic language.

Theorem 5.13. The language generated by an RPG is generated by a $(2, 1)$ precedence grammar.

DeRemer [1] developed the simple LR(k) grammars (SLR(k)) and their parsers. He said that his technique is superior to the precedence techniques both in the speed of parser construction and in the size and speed of the parsers produced. His SLR(1) techniques required from one-tenth to one-twentieth the time required by mixed strategy precedence (MSP) techniques [15] to construct parsers for practical grammars, and the resulting SLR(1) parsers require about two-thirds the space and time required for the corresponding MSP parsers.

5.4. Inclusion Charts

Feldman and Gries [4] showed the inclusion tree for the classes of grammars accepted by the particular constructors discussed in their paper (see Figure 7). They assumed that operator precedence grammars do not have identical right side productions, otherwise inclusion does not hold. The (1, 1) bounded and extended precedence grammars both use triples, the advantage for (1, 1) bounded grammars arises from the automatic intermediate reductions performed, which essentially allows more context.

Now let us improve their chart using the recent developments presented above. The improved chart is actually two charts. Figure 8 is the inclusion tree for the class of grammars and Figure 9 is the inclusion chart for the class of languages.

In Figure 8, I have identified weak and right precedence

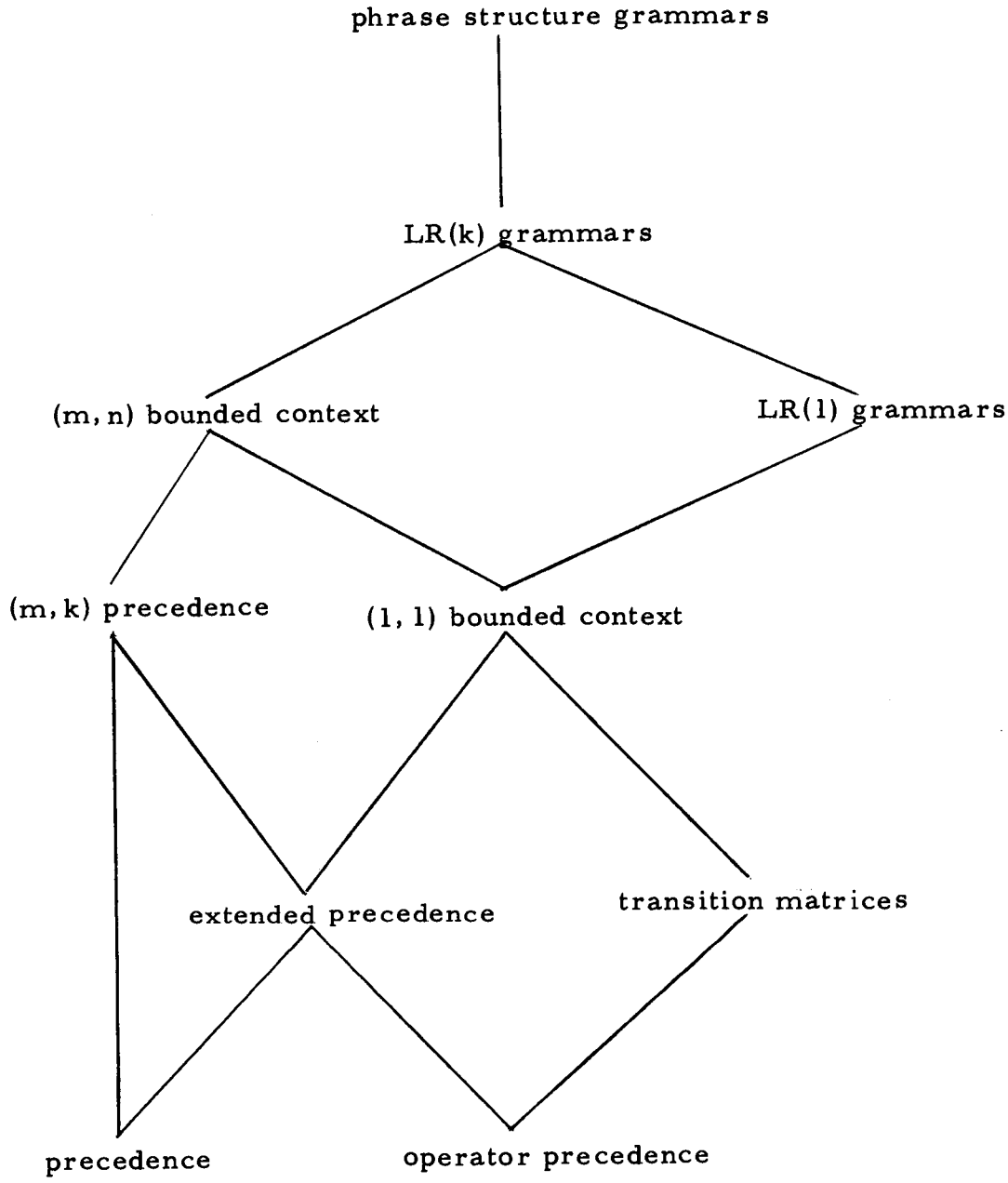


Figure 7. Feldman and Gries's inclusion tree for the class of grammars [4].

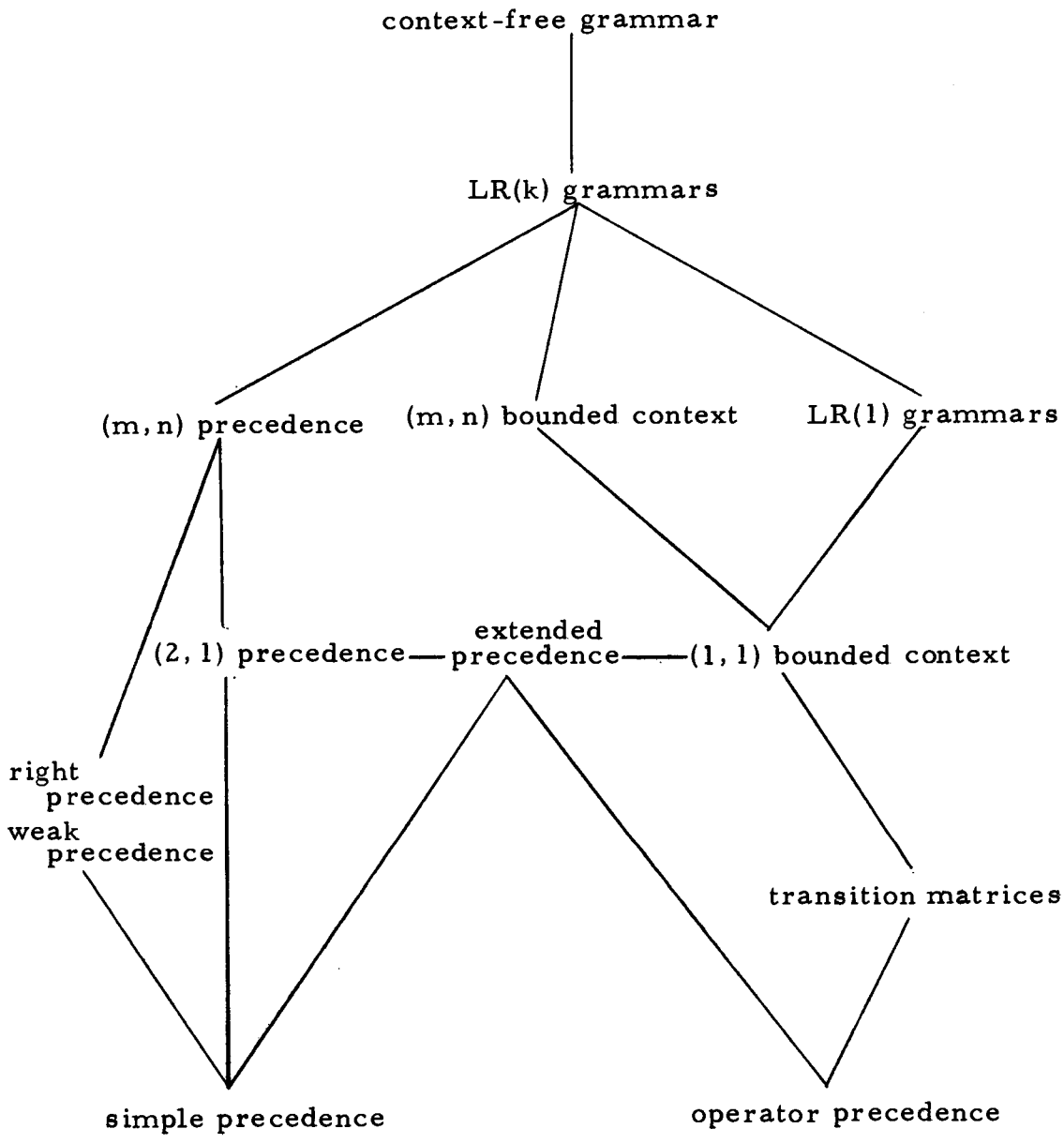


Figure 8. Revised inclusion tree for the class of grammars.

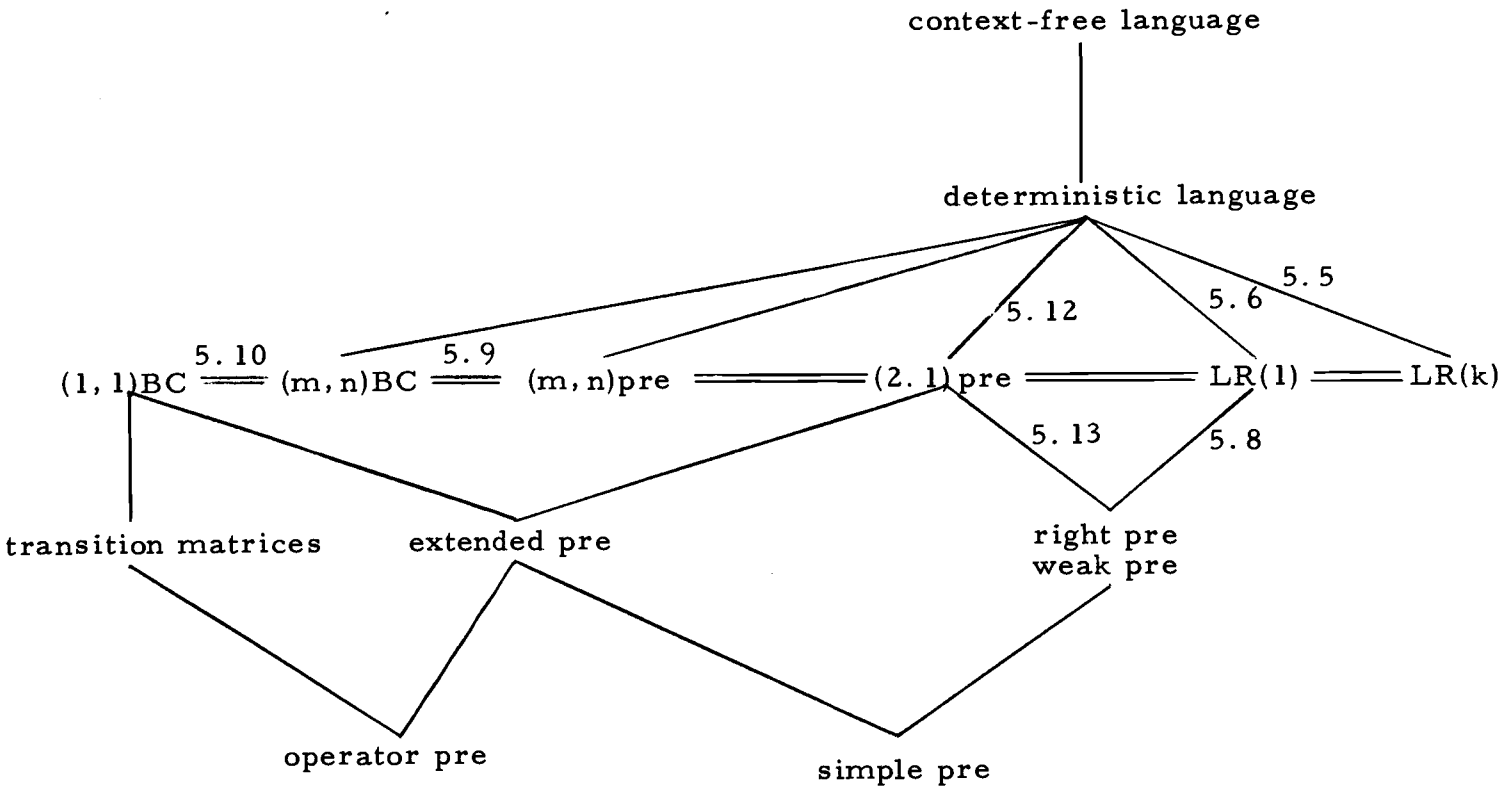


Figure 9. Inclusion chart for various class of languages.

grammars. Both grammars determine a handle by the longest match of the rightside of a production. The parsing algorithm is the same for both grammars, although the precedence relation holds between all pairs of symbols in weak precedence grammars. The definitions of both grammars essentially differ only in the manner in which they specify that the handle in a parse is uniquely determined by the longest of the rightside of a production. I have been unable to prove that they are equivalent. Both grammars fall somewhere between (m,n) precedence and simple precedence grammars.

The $(2,1)$ precedence and extended precedence grammar seem to be equivalent grammars since the extended precedence uses triples whenever conflicts arise. Thus extended precedence is $(2,1)$ precedence when conflicts arise. Also Theorem 5.11 implies $(2,1)$ precedence, extended precedence and $(1,1)$ bounded are equivalent.

The relations between extended precedence and operator precedence, and transition matrices and $(1,1)$ bounded or operator precedence are mentioned in [4].

Figure 9 shows the inclusion chart for various classes of languages generated by grammars. The numbers associated with lines refer to the appropriate theorem in this paper. A double line means two classes of language are equivalent.

The $(2,1)$ precedence and extended precedence grammar seems to generate the same class of language, but I cannot prove this.

BIBLIOGRAPHY

1. DeRemer, Franklen L. Simple LR(k) grammar. Communication of the Association for Computing Machinery 14:453-460. 1971.

2. Earley, Jay. An efficient context-free passing algorithm. Communication of the Association for Computing Machinery 13:94-102. 1970.

3. Eickel, J. (ed.), et al. A syntax controlled generator of formal language processors. Communication of the Association for Computing Machinery 6:451-455. 1963.

4. Feldman, Jerome and David Gries. Translator writing system. Communication of the Association for Computing Machinery 11:77-113. 1968.

5. Floyd, Robert W. Bounded context syntactic analysis. Communications of the Association for Computing Machinery 7:62-67. 1964.

6. Floyd, Robert W. Syntactic analysis and operator precedence. Journal of the Association for Computing Machinery 10:316-333. 1963.

7. Gauthier, R. and Ponto, S. Designing system programs. Englewood Cliff, N.J. Prentice-Hall, Inc., 1970. 269 p.

8. Graham, Susan L. Extended precedence languages, bounded right context languages and deterministic languages. IEEE Conference record of 1970 eleventh annual symposium on switching and automata theory. 175-180. 1970.

9. Hopcroft, John E. and Ulleaman, Jeffery D. Formal language and their relation to automata. Reading, Massachusetts, Addison-Wesley, 1969. 242 p.

10. Ichhian, J.D. and Morse, S.P. A technique for generating almost optimal Floyd-Evans productions for precedence grammar. Communication of the Association for Computing Machinery 13:501-508. 1970.

11. Inoue, Kenzo. Simple right precedence grammar. Joohooshiori 11:449-456. 1970.

12. Irons, E.I. "Structural Connections" in formal languages. Communication of the Association for Computing Machinery 7:67-72. 1964.

13. Knuth, Donald E. On the translation of languages from left to right. Information and Control 8:607-639. 1965.

14. Korenjak, K.J. A practical method for constructing LR(k) processors. Communication of the Association for Computing Machinery 12:613-623. 1969.

15. McKeeman, W.H., Horning, J.J. and Wortman, D.B. A compiler generator. Englewood Cliff, N.J. Prentice-Hall, Inc., 1970. 526 p.

16. Rosen, Saul (ed.), et al. Programming systems and languages. New York, McGraw-Hill, 1967. 734 p.

17. Wirth, N. and Webber, H. EULER - a generalization of ALGOL, and its formal definition: Part I, Part II. Communication of the Association for Computing Machinery 9:13-25, 89-99, 1966.

APPENDIX

APPENDIX

Example of right precedence grammar accepted by deterministic pushdown automaton. Let

$$G = (\{S, E, A, B\}, \{+, *, (,), a, \perp\}, P, S)$$

where P is

$S \rightarrow \perp E \perp$

$E \rightarrow E + A$

$E \rightarrow A$

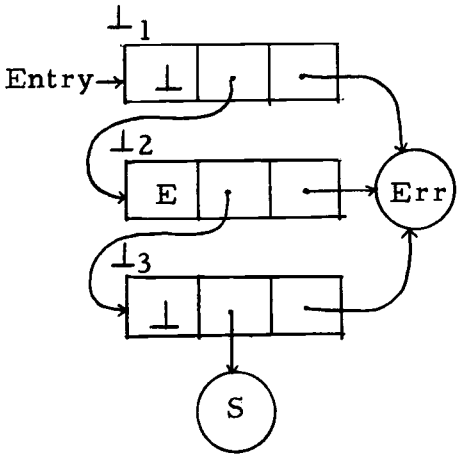
$A \rightarrow A * B$

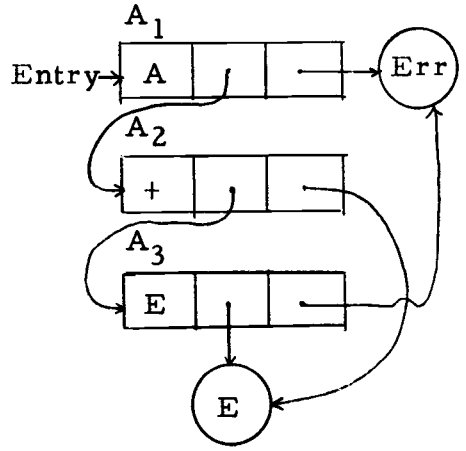
$A \rightarrow B$

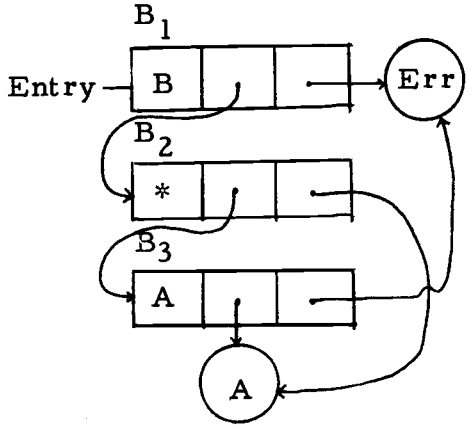
$B \rightarrow (E)$

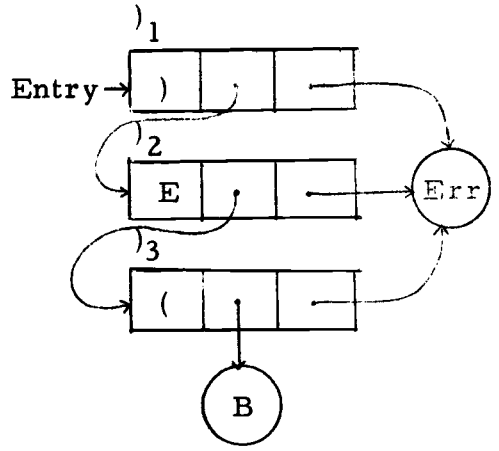
$B \rightarrow a$

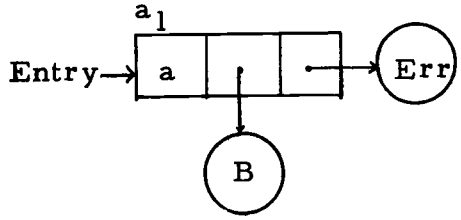
	+	*	(	)	a	$\perp$
E	$\lessdot$	$\odot$	$\odot$	$\lessdot$	$\odot$	$\lessdot$
A	$\gtrdot$	$\lessdot$	$\odot$	$\gtrdot$	$\odot$	$\gtrdot$
B	$\gtrdot$	$\gtrdot$	$\odot$	$\gtrdot$	$\odot$	$\gtrdot$
+	$\odot$	$\odot$	$\lessdot$	$\odot$	$\lessdot$	$\odot$
*	$\odot$	$\odot$	$\lessdot$	$\odot$	$\lessdot$	$\odot$
(	$\odot$	$\odot$	$\lessdot$	$\odot$	$\lessdot$	$\odot$
)	$\gtrdot$	$\gtrdot$	$\odot$	$\gtrdot$	$\odot$	$\gtrdot$
a	$\gtrdot$	$\gtrdot$	$\odot$	$\gtrdot$	$\odot$	$\gtrdot$
$\perp$	$\odot$	$\odot$	$\lessdot$	$\odot$	$\lessdot$	$\odot$


⊥1
Entry
⊥
⊥2
E
Err
⊥3
⊥
S


A1
Entry
A
Err
A2
+
A3
E
E


B1
Entry
B
Err
B2
*
B3
A
A


)1
Entry
)
2
E
Err
3
(
B


a1
Entry
a
Err
B

Input string . . ⊥a*a+a⊥

DPDA Stack	DPDA	Comment
Z_0		
⊥	$\delta(q_1, \perp, Z_0) = (q_1, \perp)$	
⊥a	$\delta(q_1, a, \perp) = (q_1, \perp a)$	relation is a $\lessdot$ ⊥
⊥a	$\delta(q_1, *, a) = (q_{*a_1}, a)$	relation is a $\gtrdot$ ⊥
⊥B	$\delta(q_{*a_1}, \epsilon, a) = (q_*, B)$	match and end
⊥B	$\delta(q_*, \epsilon, B) = (q_{*B_1}, B)$	relation is a $\gtrdot$ ⊥
⊥	$\delta(q_{*B_1}, \epsilon, B) = (q_{*B_2}, \epsilon)$	match but not end
⊥A	$\delta(q_{*B_2}, \epsilon, \perp) = (q_*, \perp A)$	not match but end
⊥A*	$\delta(q_*, \epsilon, A) = (q_1, A*)$	relation is A $\lessdot$ *
⊥A*a	$\delta(q_1, a, *) = (q_1, *a)$	relation is * $\lessdot$ a
⊥A*a	$\delta(q_1, +, a) = (q_{+a_1}, a)$	relation is a $\gtrdot$ +
⊥A*B	$\delta(q_{+a_1}, \epsilon, a) = (q_+, B)$	match and end
⊥A*B	$\delta(q_+, \epsilon, B) = (q_{+B_1}, B)$	relation B $\gtrdot$ +
⊥A*	$\delta(q_{+B_1}, \epsilon, B) = (q_{+B_2}, \epsilon)$	match but not end
⊥A	$\delta(q_{+B_2}, \epsilon, *) = (q_{+B_3}, \epsilon)$	match but not end
⊥A	$\delta(q_{+B_3}, \epsilon, A) = (q_+, A)$	match and end
⊥A	$\delta(q_+, \epsilon, A) = (q_{+A_1}, A)$	relation is A $\gtrdot$ +
⊥	$\delta(q_{+A_1}, \epsilon, A) = (q_{+A_2}, \epsilon)$	match but not end
⊥E	$\delta(q_{+A_2}, \epsilon, \perp) = (q_+, \perp E)$	not match but end
⊥E+	$\delta(q_+, +, E) = (q_1, E+)$	relation is E $\lessdot$ +
⊥E+a	$\delta(q_+, a, +) = (q_1, +a)$	relation is + $\lessdot$ a

DPDA Stack	DPDA	Comment
$\perp E+a$	$\delta(q_+, \perp, a) = (q_{+a_1}, a)$	relation is $a \gtrdot \perp$
$\perp E+B$	$\delta(q_{\perp a_1}, \epsilon, a) = (q_\perp, B)$	match end end
$\perp E+B$	$\delta(q_\perp, \epsilon, B) = (q_{\perp B_1}, B)$	relation is $B \gtrdot \perp$
$\perp E+$	$\delta(q_{\perp B_1}, \epsilon, B) = (q_{\perp B_2}, \epsilon)$	match but not end
$\perp E+A$	$\delta(q_{\perp B_2}, \epsilon, +) = (q_\perp, +A)$	not match but end
$\perp E+A$	$\delta(q_\perp, \epsilon, A) = (q_{\perp A_1}, A)$	relation is $A \gtrdot \perp$
$\perp E+$	$\delta(q_{\perp A_1}, \epsilon, A) = (q_{\perp A_2}, \epsilon)$	match but not end
$\perp E$	$\delta(q_{\perp A_2}, \epsilon, +) = (q_{\perp A_3}, \epsilon)$	match but not end
$\perp E$	$\delta(q_{\perp A_3}, \epsilon, E) = (q_\perp, E)$	match and end
$\perp E \perp$	$\delta(q_\perp, \epsilon, E) = (q_{S\perp_1}, E\perp)$	relation is $E \leq \perp$ and $V_t = \perp$
$\perp E$	$\delta(q_{S\perp_1}, \epsilon, \perp) = (q_{S\perp_2}, \epsilon)$	match but not end
$\perp$	$\delta(q_{S\perp_2}, \epsilon, E) = (q_{S\perp_3}, \epsilon)$	match but not end
S	$\delta(q_{S\perp_3}, \epsilon, \perp) = (q_S, S)$	match and end
	$\delta(q_S, \epsilon, S) = (q_F, S)$	