A MODIFIED MARKOV ALGORITHM AS A LANGUAGE PARSER

BY

Jacob Katzenelson and Elie Milgrom

Memorandum No. ERL-M363

22 June 1972

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Jacob Katzenelson is on sabbatical leave from the Technion, Israel Institute of Technology, Haifa, Israel.

Elie Milgrom is with the Department of Applied Mathematics, University of Louvain, 3030 Heverlee, Belgium.

Research sponsored by the National Science Foundation, Grant GK-10656X2.

ABSTRACT

The Markov Algorithm is modified to form a tool for the specification of the syntax and for the parsing of programming languages. The properties of the Modified Markov Algorithm (MMA) are explored and sufficient conditions are given under which the order of the MMA's rules can be changed and the composition of algorithms is equivalent to appending. The article also describes transformations of the MMA to algorithms which are equivalent to the MMA but are faster and discusses the relations between a special case of the MMA and the context bounded grammars.

The above results are used to synthesize parsers from a given description of the language. The method involves partitioning the language to sub-languages, providing each with a MMA parser, composing the parsers to a single parser and transforming the result to a single MMA. If possible the last MMA is then transformed into an equivalent algorithm to increase parsing speed. The article illustrate the method by constructing a syntax parser for a subset of Algol 60 where syntax is interpreted in a wide sense-including scope of variables, etc.

Our main conclusion is that the use of the MMA for specifying the syntax and parser are quite straightforward. Moreover, the rules of the resulting MMA are quite readable in the sense that they closely correspond to the way that a programmer interprets a program. This is attributed to the use of an algorithm to specify the language, the use of context and environment dependent rules by the MMA and the help of the theoretical results mentioned above.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

Section I: Introduction

Several works have recently used algorithms derived from the Markov Algorithm (MA) [1, 2, 3] as language parser and as a method for defining the syntax of computer languages. Caracciolo [4] uses an extension of the MA for defining string processing languages. De Bakker [5] uses a computation scheme which is quite similar to the MA to formally define the syntax and the semantics of Algol 60. The works of Bell [6] and Milgrom [7] are concerned with extensible languages. In these works algorithms derived from the MA are used as a parsers which can be modified and extended dynamically to handle new syntactic constructs.

Markov Algorithms have several interesting properties which make their use attractive for such applications. First, the MA is a general computing scheme; it can do whatever other computing schemes can do, and is equivalent, for example, to the Turing Machine. This means that the MA can be used to specify the syntax (and semantics) of any language which can be defined by an algorithm on a finite set of symbols. Second, the Markov Algorithm uses a list of rewriting rules associated with a list of priorities. The rules look quite similar to BNF and their order is close to our intuitive idea of priorities between operators and operations of a programming language. However, contrarily to BNF rules, the MA rules and their priorities always define the parser, i.e., they define a Markov algorithm which tests whether a string belongs to the language or not.[†] This property is shared with the rules of Production Language (Ch. 7, [8]).

[†]The result, however, might be undecidable.

This article presents a modified MA (MMA) and explores its properties which are related to the specifications of the syntax of programming languages and the corresponding parsers. By the syntax of a programming language we mean a set of rules which defines which sequences of symbols constitute a legal program [5]. This definition of syntax includes requirements on the scope of names and labels, the matching of types, etc. which are sometimes considered as part of the semantics of the language.

We consider the following to be our main conclusion: We have used the MMA to specify the syntax and the parser for Algol 60 and for a language for the manipulation of graphs. We have found that the writing of such definitions is reasonably simple and in many cases straightforward; moreover we found that the resulting rules are quite readable in the sense that a user can look at the rules and can quite easily determine the effect of each rule and the interrelation between the rules.

We believe that the above properties are of special importance in the area of problem-oriented languages. Here as a result of the specific orientation and the relatively small number of users the languages have to be designed by the application oriented people rather than by the language specialists. The simplicity of the synthesis procedure which leads to the parser and the readability of the rules are of great value in this case.

On the other hand, the questions of simplicity and users convenience are related to 'human engineering' and they are somewhat subjective. In such cases the question is whether this simplicity is a result of the familiarity of the authors with the MMA or whether there is a more fundamental reason for it. Although we cannot completely rule out the first possibility, we claim that the relative simplicity and convenience in the use of the MMA are linked to the properties of the MMA. Some of these properties are mentioned above as properties of the MA. Other properties are the following: the use of context-sensitive rules and the number of non-terminal symbols; to a large extent the form of the rules is similar to the way a programmer thinks about a program when he reads it; in the examples we worked out, we found a large subset of rules whose order did not matter; the order of those rules could be changed without affecting the nature of the language.

Other results included in the work are sufficient conditions for the interchangability of the order of rules in the MMA (and the MA); the relation between MMA grammars which satisfy these conditions and (m,n) bounded-context grammars, and the transformation of the MMA to an equivalent form which corresponds to a faster parser.

The next section contains the definition of the MMA, a formal definition of languages using the MMA, the syntax tree, etc. Section III contains several theorems related to the sufficient conditions for changing the order of the rules and for the composition of MMA's. Section IV is concerned with the Algol 60 example and discusses the method which is used to construct the example; the rules themselves appear in the appendix. Section V contains a summary and conclusions.

Section II: The Markov Algorithm and the Language Definition

2.1 The Algorithm

Let us first define the Markov Algorithm as it is conventionally defined.

We assume the existence of a non-empty finite set of symbols called the alphabet. A word or a string is any finite sequence of symbols of the alphabet. The empty sequence of symbols is called the empty word, or empty string, and is denoted by Λ. Symbols are denoted by upper case characters and strings by lower case characters; $\underline{p}\ \underline{q}$ denotes the concatenation of the strings $\underline{p}$ and $\underline{q}$. We say that the string $\underline{t}$ occurs in the string $\underline{q}$ if there exist strings $\underline{u}$ and $\underline{v}$, possibly empty, such that $\underline{q} = \underline{utv}$.

A rule is an expression of the form

$$\underline{p} \rightarrow \underline{q}$$

$$\text{or} \quad \underline{p} \rightarrow . \underline{q}$$

the arrow and the dot are not part of the alphabet. The expression

$$\underline{p} \rightarrow (.) \ \underline{q}$$

denotes either $\underline{p}\rightarrow\underline{q}$ or $\underline{p}\rightarrow.\underline{q}$. $\underline{p}$ is called the input-pattern or antecedent of the rule and $\underline{q}$ is called the output-pattern or consequent of the rule.

A grammar is a finite ordered list of rules

$$\underline{p}_i \rightarrow (.) \ \underline{q}_i \qquad \text{where } i=1, 2, \ldots, I.$$

We say that a rule i has a higher priority than a rule j if $i < j$.

Given any string $\underline{s}$ (called the input string), a grammar defines the following algorithm:

Step 1: Set $i = 1$.

Step 2: Consider the i-th rule in the grammar and find the leftmost occurrence of $\underline{p}_i$ in $\underline{s}$; if no such occurrence is found, then go to step 4; otherwise proceed.

Step 3: An occurrence has been found. Replace it by $\underline{q}_i$; if the i-th rule contains a dot, terminate; otherwise, go to step 1.

Step 4: Set $i = i+1$. If $i > I$, then terminate; otherwise go to step 2.

We depart from the above definition by adding several features which do not limit the generality of the algorithm but make it more convenient for writing parsers. Some of these features are derived from Bell's PBNF [6]. The result is called the <u>Modified Markov Algorithm</u> (MMA).

The rules have the form

$$R_i \quad N_i \qquad \underline{p}_i \rightarrow(.) \ \underline{q}_i \mid C_i \mid P_i$$

where R_i is the name of the rule,

N_i is an integer called the priority level of the rule,

$\underline{p}_i$ and $\underline{q}_i$ are strings, the input- and output-patterns of the rule.

C_i is a predicate, and

P_i is a program.

We can have several rules with the same priority level. However, among rules with the same priority level there should not exist two rules whose input-patterns are of the form $\underline{x}$ and $\underline{ux}$, $\underline{u}$ may be empty, and whose predicates can be both true at the same time (see definition of the algorithm below).

Strictly speaking, the predicates and the programs do not increase the capability of the MA. Their function is to simplify the writing of the parser and to make the rules more readable. To explain the operation of the predicate C_i and the program P_i we define a concept which we call an environment which we denote by $\mathcal{E}$. The environment is a collection of variables and associated data-structures, etc. The predicate C_i is a Boolean function which tests the values of these variables and data structures. The program P_i operates on the environment, modifies the values of the variables etc. The predicate C_i can depend also on the value of string variables which appears in the input-pattern $\underline{p}_i$. In the following example $\underline{x}$ is a given string while $\underline{\xi}$ is a string variable:

$$R_i \quad N_i \quad \underline{\xi x} \rightarrow \underline{y} \ | \text{predicate: } \underline{\text{if}}\ \underline{\xi} = \text{'+' or '- +'} \ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}}.$$

Note that we do not give a formal definition for the environment or for the language in which the programs and predicates are expressed. Such definitions can be given by postulating a convenient processor as in [7] or extensions to the MA as in [5].

With the introduction of the environment, our concept of a grammar includes both the rules and the environment. Note, however, that to define a MMA both the grammar and the initial values of all the variables in the environment has to be given. We shall refer to these values as the initial environment.

The length of a string $\underline{p}$ is defined as the number of symbols in $\underline{p}$. The position of a substring $\underline{x}$ in a string $\underline{uxv}$ is defined as the length of $\underline{ux}$. Given a set of substrings of a string $\underline{s}$, let $\{p_i\}$ be the corresponding set of positions; let p_o be the smallest number in $\{p_i\}$. The leftmost strings in the set are defined to be the substring whose position is p_o.

Given any string s̲ and an initial value the environment $\mathcal{E}_0$ the Modified Markov Algorithm with I priority levels is defined as follows:

```
Step 1:  Set i = 1.
Step 2:  Consider the i-th priority level.
         Let $R_j$, j=1, ..., $j_0$ be the set of rules
         having this priority level. Over all j between
         1 and $j_0$, find the leftmost occurrence of $\underline{p}_j$
         in s̲ such the $C_j$ is TRUE; if no such occurrence
         has been found, then go to step 4; otherwise
         proceed.
Step 3:  An occurrence has been found; let this occurrence
         correspond to the rule $R_j$. Replace it by $\underline{q}_j$ and
         execute the program $P_j$. If $R_j$ contains a dot
         then terminate; otherwise go to step 1.
Step 4:  Set i = i+1. If i > I, then terminate;
         otherwise go to step 2.
```

Remark: In step 2 the leftmost occurrence of $\underline{p}_i$ is uniquely defined since among rules with the same priority level there should not exist two rules whose input patterns are of the form x̲ and u̲x̲, and whose predicates can be true at the same time.

2.2 The Language

In the following we follow the notations and the terms used in [7] ch. 3. The purpose of this section is the presentation of the differences between the conventional definitions and the definitions based on the MMA.

Given a grammar G, an initial environment $\mathcal{E}_0$, a string v̲, $\underline{v=xyz}$ and a string $\underline{w}$, $\underline{w=xsz}$. The string $\underline{w}$ is <u>directly derived</u> from $\underline{v}$, if in the process of applying to $\underline{v}$ the MMA with grammar G and an environment $\mathcal{E}_0$, the <u>first</u> rule which uses Step 3 reduces $\underline{y}$ to $\underline{s}$ and transforms $\underline{v}$ to $\underline{w}$. This is denoted by

$$\underline{v} \Rightarrow \underline{w} .$$

A <u>derivation</u> of $\underline{v}$ by G is a sequence of direct derivations effected by the MMA on $\underline{v}$.

A string w is <u>derived</u> from $\underline{v}$ (by G) if there exists a derivation of the form $\underline{v} \Rightarrow \underline{u}_0 \Rightarrow \underline{u}_1 \Rightarrow \ldots \Rightarrow \underline{w}$. This is written as $\underline{v} \Rightarrow + \underline{w}$.

We write $\underline{v} \Rightarrow* \underline{w}$ if either $\underline{v} = \underline{w}$ or $\underline{v} \Rightarrow + \underline{w}$.

We write $\underline{v} \Rightarrow. \underline{w}$ if the MMA derives $\underline{w}$ from $\underline{v}$ (by G) and terminates.

Let the alphabet $\underline{V}$ be partitioned to two disjoint sets: $\underline{V}_T$, the terminal symbols and $\underline{V}_N$, the nonterminal symbols. Let $\underline{V}_T^*$ denote the set of all strings over $\underline{V}_T$. Let S be a symbol in $\underline{V}_N$.

The language corresponding to the grammar G, the initial environment $\mathcal{E}_0$, the sets of symbols $\underline{V}$, $\underline{V}_T$ and the root S is defined as:

$$L(G, \mathcal{E}_0, \underline{V}, \underline{V}_T, S) = \{ \underline{x} | \underline{x} \in \underline{V}_T^* \text{ and } \underline{x} \Rightarrow. S\}$$

Given a string $\underline{x}$ such that $\underline{x} \in \underline{V}_T^*$ the question whether $\underline{x}$ is a member of a given L might be undecidable since for some G, some $\mathcal{E}_0$, and some $\underline{x}$ the MMA might not terminate.

Given G and $\underline{V}$, a string $\underline{w}$ is called <u>a sentential form</u> of G if $w \Rightarrow. S$; if $\underline{w}$ is an element of $\underline{V}_T^*$ it is called <u>a sentence</u>.

Given a grammar G and a sentential form $\underline{v}$. $\underline{y}$ is a <u>phrase</u> of $\underline{v}$ if there exist strings $\underline{s}$, $\underline{t}$, $\underline{u}$, $\underline{x}$, $\underline{z}$ (possible empty) such that $\underline{v} = \underline{xyz} \Rightarrow * \underline{syt} \Rightarrow \underline{sut}$ is a derivation of $\underline{v}$ by G. The leftmost phrase of a sentential form $\underline{v}$ is called the <u>handle</u> of $\underline{v}$.

2.3 The Syntax Tree

The syntax tree if s device which aids one to understand the syntax of sentences. We can define a similar concept here. However, since output patterns of the rules may contain more than one symbol we get a syntax graph rather than a syntax tree. The two terms are used interchangingly in the sequel. The concept is illustrated in Figure 1.

A syntax-graph consists of two kinds of nodes: symbol-nodes and replacement nodes. Each symbol node represents a symbol in a rather self-explanatory way. A replacement node represents a replacement performed by the algorithm on the input string (for example, nodes (a) and (b) in figure 1). Oriented branches connect each replacement-node with all the symbol-nodes which participate in the reduction. If a symbol is in the input-pattern of the replacement rule the branch is oriented from the symbol to the replacement node. If a symbol is in the output-pattern of the replacement rule the branch is oriented from the replacement node to the symbol.

Consider node (a) of Figure 1. All branches which enter this node are connected to symbol nodes which are incident to one branch only. Replacement nodes with this property are called leaves. The removal of a leaf is the removal of the corresponding node, all branches incident to it and all symbol-node which as a result of the above operation do not have any branch incident to them. Since each replacement-node correspond to a rule, we can speak about the priority, input-pattern, etc., of the node, meaning actually the priority, etc. of the rule which is associated with the node.

Section III: Some Properties of the Parser

This section includes several properties of the MMA. Some of these properties are included for completeness and for insight (termination, handle, relation with (m,n) bounded context languages). Other properties were found useful in the synthesis of grammars and in particular in the example of next section (the interchanging of the order of the rules and composition of MMA's) or in reducing the computer time requirements (the markers, the transformation to two stack algorithm).

3.1 Termination

We are interested in conditions imposed on the rules of the grammar which guarantee the termination of the MMA for any input string. Some simple sufficient conditions are given in [9]. These conditions require that for all rules $\ell(\underline{p}_i) \geq \ell(\underline{q}_i)$, and that a certain oriented graph derived from the grammar has no (oriented) loops; $\ell(\underline{q})$ denotes the number of symbols in the string $\underline{q}$. The graph is constructed in the following way:

1. A node is associated with each rule for which

$$\ell(\underline{p}_i) = \ell(\underline{q}_i) .$$

2. A directed branch is put between nodes i and j if $\underline{q}_i$ contains symbols which also appear in $\underline{p}_j$.

3.2 The Handle

Since the definition of a handle in the MMA sense is so similar to the definition of a handle in the conventional sense ([8] ch. 3) one is tempted to reduce the input string by first finding the handle, reducing it, finding the handle of the resulting string, etc. Figure 2 illustrates that this strategy does not necessarily produces the correct result for all grammars and all input strings.

On the other hand one can interpret theorems Ia and Ib of section 3.3 as giving sufficient conditions on the rules of the grammar under which the handle after handle strategy produces the correct results for any sentential form.

3.3 Changing Priority Levels of the MMA Grammar Rules

Theorems Ia, Ib and IIa are special cases of theorem IIb. We have chosen this rather lengthy method of presentation since we found that the direct presentation of the mass of conditions of theorem IIb tends to obscure rather than clarify the main issues and techniques.

Let i be a positive integer and $\underline{a}$ be a string of ℓ symbols. $\text{head}_i(\underline{a})$ $(\text{tail}_i(\underline{a}))$ is defined to be a string of the leftmost (rightmost) i symbols in $\underline{a}$. $\text{head}_i(\underline{a})$ and $\text{tail}_i(\underline{a})$ are not defined for $i > \ell$.

Two (non-empty) strings $\underline{a}$ and $\underline{b}$ <u>overlap</u> if one or more of the following holds:

(1) There exists an i, $i>0$, such that $\text{tail}_i(\underline{a}) = \text{head}_i(\underline{b})$;

(2) there exists a j, $j>0$, such that $\text{head}_j(\underline{a}) = \text{tail}_j(b)$;

(3) one of the strings is a substring of the other.

The concept of overlap between rules has to be defined with respect to the value taken by the environment $\mathcal{E}$, and the values of the variables which appear in the input-patterns of the rules.

Let R_i and R_j be two rules with input patterns $\underline{p}_i(\xi_i)$ and $\underline{p}_j(\xi_j)$ where ξ_i and ξ_j denote the variables appearing in $\underline{p}_i$ and $\underline{p}_j$. The two rules <u>overlap</u> if there exist (at least) two values, $\overline{\xi}_i$ and $\overline{\xi}_j$ and an environment $\mathcal{E}$ such that $\underline{p}_i(\overline{\xi}_i)$ and $\underline{p}_j(\overline{\xi}_j)$ overlap and both the predicates $c_i(\underline{p}_i(\overline{\xi}_i), \mathcal{E})$ and $c_j(\underline{p}_j(\overline{\xi}_j), \mathcal{E})$ are <u>true</u>.

It is clear that if the input-patterns of R_i and R_j contain no variables and the predicates are independent of $\mathcal{E}$ (the predicate is always true), then the above definition of overlap between two rule reduces to overlap between $\underline{p}_i$ and $\underline{p}_j$.

Theorem Ia

Let G_1 be a grammer such that each of its rules satisfies the following conditions:

(1) The patterns do not contain variables;

(2) the predicate does not depend on the environment;

(3) the rule is not "dotted";

(4) the root symbol appears in one rule only and only in the output pattern.

Let G_2 be a grammer derived from G_1 by changing the priority levels of rules of G_1.

If the rules of G_1 do not overlap each other and themselves, then any sentential form of G_1 (G_2) is also a sentential form of G_2 (G_1).

Proof

Let $\underline{s}$ be a sentential form of G_1 and let τ be the syntax-tree which corresponds to the reduction of $\underline{s}$ by G_1.

A match of the string $\underline{s}$ and a rule R_i is any substring of $\underline{s}$ which is equal to $\underline{p}_i$ (Note that the conditions of theorem Ia exclude any role of the environment).

From the construction of the tree follows two important properties: (property a.I) τ specifies $\underline{s}$ completely; one can construct $\underline{s}$ by collecting all symbol nodes of τ which have only one branch incident to them and the branch is oriented out of the node. Consider any leaf of τ and the associate rule R_i. The symbols of the symbol nodes whose branches enter the replacement node of the leaf form the string $\underline{p}_i$. This string is a substring of $\underline{s}$ and as a result of condition (2) of theorem Ia the string is a match of $\underline{s}$ and R_i. We say that this match corresponds to the leaf. Thus, the second property is: (property b.I) To any leaf of τ there corresponds a match in $\underline{s}$.

Notice that (b.I) holds only under the assumptions of the theorem as a result of the independence of the rules from $\mathcal{E}$, while (a.I) holds for any tree of a sentential form.

Given $\underline{s}$, τ and an integer n, $n \geq 0$, the following algorithm transforms $\underline{s}$ and τ into $\underline{s}_n$ and τ_n:

Algorithm AI

```
Step 1:  Set $i \leftarrow 0$; if i equals n terminate.
Step 2:  If $\tau$ does not contain leaves then terminate, otherwise
         choose one leaf of $\tau$, remove the leaf and reduce the
         corresponding match in $\underline{s}$ using the rule which corresponds
         to the chosen leaf.
Step 3:  Set $i \leftarrow i+1$; if i equals n terminate, otherwise go to
         step 2.
```

The algorithm requires some discussion. Let us apply AI to $\underline{s}$ and τ. If τ has leaves then step 2 chooses a leaf and transforms τ into another tree which is denoted by τ_1. (the index is the value of i after step 3). The leaf has a corresponding match (property b.I) and $\underline{s}$ is transformed into $\underline{s}_1$. If τ has leaves (and $n \geq 2$) step 2 chooses a leaf of τ_1 and transforms τ_1 into τ_2. Does a leaf of τ_1 have a corresponding match? The answer is positive and the same argument that leads to property b.1 results in: (property c.I) to any leaf of τ_1 there corresponds a match. Thus, we conclude that AI is defined for all n and that the following lemma holds:

<u>Lemma 1.I</u> Given a sufficiently large n, AI reduces $\underline{s}$ to the root symbol.

<u>Lemma 2.I</u> Let n be an integer, $n \geq 0$, and let $\underline{s}_n$ and τ_n be defined as above. (a) To any leaf of τ_n there corresponds a match in $\underline{s}_n$ and (b) To every match in $\underline{s}_n$ there corresponds a leaf in τ_n.

The first part of the lemma is property c.I. The second part requires a proof. Assume there exists a match to a rule R_i to which does not correspond a leaf. $\underline{s}_n$ can be reduced to the root symbol using AI. In this process all symbols of the match are removed from the string. Any rule which removes these symbols overlaps R_i, unless the rule is R_i itself and the match is removed as a whole. The first case implies a contradiction with the non-overlapping requirement and the second with the assumption that there is no leaf in τ_n which corresponds to the match. Thus, to each match in $\underline{s}_n$ there corresponds a leaf in τ_n which completes the proof. QED lemma 2.I

The proof of the theorem follows directly from the lemma. Apply G_2 to $\underline{s}$ and consider τ. At each step a match is found and is replaced. At each such step we remove the corresponding leaf. The process cannot terminate before all leaves are removed since each leaf implies a match. To prove that, consider the following argument: After i steps G_2 transforms $\underline{s}$ and τ into $\underline{s}_i$ and $\underline{\tau}_i$ which are the same string and tree which result from the application of AI to $\underline{s}$ and τ where in each step AI chooses the leaf which corresponds to the match that G_2 finds (lemma 2.I). Thus, from lemma 2.I, each leaf of τ_i implies a match and termination occurs only when the root symbol S is reached, which makes $\underline{s}$ a sentential form of G_2. QED Ia.

Corollary: Let $\underline{s}_1$ be the string derived from $\underline{s}$ by removing one leaf of τ. $\underline{s}_1$ is a sentential form of G_1

Theorem IIa

Let the alphabet consist of three disjoint sets ψ_1, ψ_2 and ψ_3.

Let G_1 be a grammar which consists of two sets of rules, α and β, and which satisfies the following conditions:

α rules: 1) The input-patterns consist of symbols of ψ_1 and ψ_2 only; output-patterns consist of symbols of ψ_2 only;

2) the α - rules do not overlap any rule of β;

3) the α - rules do not contain a 'dot' nor do they contain the root symbol;

4) the predicates of α - rules depend on the environment only and not on string variables.

β rules: 1) The input-patterns consist of symbols of ψ_2 and ψ_3 only; output-patterns consit of symbols of ψ_3 only;

2) the β-rules satisfy all the requirements imposed on the rules in theorem Ia;

3) the β-rules do not change the environment;

4) the output patterns of the β rules are non-empty.

Let G_2 be a grammar derived from G_1 by changing the priority levels of the β rules. Any sentential form of G_1 (G_2), $\mathcal{E}_0$, is a sentential form of G_2 (G_1), $\mathcal{E}_0$.

Remark: Observe that rules of α can overlap and can depend on the environment.

Proof:

Let s be a sentential form of G_1 when the initial environment is

$\mathcal{E}_0$ and let τ be the corresponding syntax tree.

Let a leaf which corresponds to a $\beta(\alpha)$ - rule be called a <u>$\beta(\alpha)$ - leaf</u> and a match which corresponds to a $\beta(\alpha)$-rule be called <u>$\beta(\alpha)$ - match</u>.

Let us apply G_1, $\mathcal{E}_0$ to $\underline{s}$. G_1, $\mathcal{E}_0$ uses the rules in the following sequence

$$R_{\beta_{0,1}}, \ldots, R_{\beta_{0,k1}}, R_{\alpha_1}, R_{\beta_{1,1}}, \ldots, R_{\beta_{1,k2}}, R_{\alpha_2}, \ldots$$

and <u>after</u> each application of a rule it creates the following nodes, inputs strings and environment values,

$$N_{\beta_{0,1}}, \ldots, N_{\beta_{0,k1}}, N_{\alpha_1}, N_{\beta_{1,1}}, \ldots, N_{\beta_{1,k2}}, N_{\alpha_2}, \ldots \quad ;$$

$$\underline{s}_{\beta_{0,1}}, \ldots, \underline{s}_{\beta_{0,k1}}, \underline{s}_{\alpha_1}, \underline{s}_{\beta_{1,1}}, \ldots, \underline{s}_{\beta_{1,k2}}, \underline{s}_{\alpha_2}, \ldots \quad ;$$

$$\mathcal{E}_0, \ldots, \mathcal{E}_0, \mathcal{E}_{\alpha_1}, \mathcal{E}_{\alpha_1}, \ldots, \mathcal{E}_{\alpha_1}, \mathcal{E}_{\alpha_2}, \ldots \quad ;$$

If at each reduction we remove the corresponding node from τ we get a sequence of trees

$$\tau_{\beta_{0,1}}, \ldots, \tau_{\beta_{0,k1}}, \tau_{\alpha_1}, \tau_{\beta_{1,1}}, \ldots, \tau_{\beta_{1,k2}}, \tau_{\alpha_2}, \ldots .$$

From conditions α-1 and β-1 of the theorem follows the following property: (property a.II) For any α-node N_α of τ, branches entering N_α come from symbols in $\underline{s}$ or from symbol-nodes such that the branches which enter them come from an α-nodes. (We say that an α-node <u>depends</u> on $\underline{s}$ and on α-nodes only; this is the definition of the dependency of one node on other nodes and on $\underline{s}$ which is used in the sequel).

Let m be the number of α-nodes in τ. From the definition of the MMA follows the following property: (property b.II) For any i, $i + 1 \leqslant m$,

$N_{\alpha_{i+1}}$ corresponds to the leftmost highest priority α-match in any of the strings $\underline{s}_{\alpha_i}, \underline{s}_{\beta_{i,1}}, \ldots \underline{s}_{\beta_{i,ki}}$.

Let $\underline{s}_k$ be any string which contains both α- and β-matches. $\underline{s}_{k+1}$ is derived from $\underline{s}_k$ by reducing one of the β-matches. From conditions α-1, β-1, and β-4 follows that the reduction cannot introduce to $\underline{s}_{k+1}$ any new α-match which does not appear in $\underline{s}_k$. From condition β-3 (environment) and (α-2) follows that the operation cannot remove an α-match from $\underline{s}_k$. Therefore, (property c.II) $\underline{s}_k$ has the same α-matches as $\underline{s}_{k+1}$.

Given an integer n, $n \geq 0$ the following algorithm transforms $\underline{s}$ and τ into $\underline{s}_n$ and τ_n:

Algorithm AII

```
Step 1:  set $i \leftarrow 0$ ; $j \leftarrow 1$; if i equals n then terminate.
Step 2:  Choose either a β-leaf or the $N_{\alpha_j}$ leaf. If such leaves
         are unavailable then terminate. Remove this leaf and
         reduce the corresponding match in $\underline{s}$ using the corresponding
         β-rule (for a β-leaf) or $R_{\alpha_j}$ for $N_{\alpha_j}$.
Step 3:  set $i \leftarrow i + 1$; If $N_{\alpha_j}$ has been chosen set $j \leftarrow j + 1$;
         if i equal n then terminate; otherwise go to step 2.
```

As in the discussion of AI we want to consider step 2 of AII and to show that this step can be executed for any n. For this it is enough to show that as long as the tree has leaves, to each β-leaf and to the N_{α_j} leaf there exist corresponding matches in the input string.

From the construction of the tree and condition β-2 of the theorem it is clear that to each β-leaf there corresponds a match. Consider the process at the beginning of step 2 with the indices i and j; let what is left of τ be denoted by τ_{ij} and let $\underline{s}_{ij}$ and $\mathcal{E}_{ij}$ be the input string and the environment at this point. Does $\underline{s}_{ij}$ contain a match to N_{α_j}?

N_{α_j} is present in τ_{ij}. Since all N_{α_i}, $i < j$, have been removed, all branches entering N_{α_j} are from symbols in $\underline{s}_{ij}$ (property a.II). In other words, N_{α_j} is a leaf and the input pattern of R_{α_j} appears in $\underline{s}_{ij}$. Since only α-rules change the environment and all rules of N_{α_i}, $i < j$, have been applied and in the same order as in the application by G_1, $\mathcal{E} = \mathcal{E}_{\alpha_{j-1}}$, which is exactly the same environment in which N_{α_j} was created by applying G_1. In other words, $\underline{s}_{ij}$ contains a match to R_{α_j}.

The following lemma follows immediately from the above discussion.

<u>Lemma 1.II</u> Given a sufficiently large n, AII reduces $\underline{s}$ to the root symbol.

<u>Lemma 2.II</u>

a) To any β-leaf of τ_{ij} there corresponds a β-match in $\underline{s}_{ij}$ and to any β-match in $\underline{s}_{ij}$ there corresponds a β-leaf in τ_{ij}.

b) Consider all α-leaves of τ_{ij} that have predicates which are true for $\mathcal{E}_{ij}$. If this set is not empty then among these leaves consider those with the highest priority level and let the leftmost leaf of the set be A.

$\underline{s}_{ij}$ contains a match which correspond to A.

Consider all matches of $\underline{s}_{ij}$ by α-rules when the environment is $\mathcal{E}_{ij}$. If this set is not empty then among these matches consider those matches with the highest priority level and let the leftmost match of the set be M.

τ_{ij} contains a leaf which corresponds to M and this leaf is A.

<u>Proof of lemma 2.II</u>

a) In discussion AII we proved that to any β-leaf there corresponds a match. The fact that to every β-match corresponds a leaf follows from an argument similar to the one used in Ia. Assume the existence of a β-match with no leaf. We can reduce $\underline{s}_{ij}$ to the root symbol using AII (lemma 1.II). This implies that all symbols of the match are reduced and that the rule which corresponds to the match overlaps other rules in the grammar; which contradicts conditions α-2 and β-2 of the theorem.

b) From property a.II follows that τ_{ij} and $\tau_{\alpha_{j-1}}$ have the same α-leaves. Since N_{α_j} is the leftmost highest priority α-leaf of $\tau_{\alpha_{j-1}}$ (property b.II) A is nothing else but N_{α_j}. In discussing AII we proved that $\underline{s}_{ij}$ contains a match which corresponds to N_{α_j} which means that A has a corresponding match in $\underline{s}_{ij}$. Let us denote this match by M_A.

We shall prove now that $\underline{s}_{ij}$ and $\underline{s}_{\alpha_{j-1}}$ (and $\underline{s}_{\beta_{j-1,1}}$ to $\underline{s}_{\beta_{j-1,kj-1}}$) have the same set of α-matches. This proof concludes the proof of part b since, by construction, M_A is the leftmost highest priority match of $\underline{s}_{\alpha_{j-1}}$ and M, by definition, has the same property with respect to $\underline{s}_{ij}$; in other words M and M_A are the same match.

Let AII obtain $\underline{s}_{\alpha_{j-1}}$ by reducing the nodes (leaves) of τ in the following sequence:

$$N_{\alpha_1}, N_{\alpha_2}, N_{\alpha_3}, \ldots, N_{\alpha_{j-1}}, N_{\beta_{0,1}}, \ldots, N_{\beta_{0,k1}}, N_{\beta_{1,0}}, \ldots, N_{\beta_{j-1,kj}};$$

where all α-nodes appear at the head of the sequence. The above sequence is a feasible reduction scheme for AII since: the α-nodes are reduced in the original order and to each, in its turn, corresponds a match; nodes $N_{\beta_{0,1}}$ to $N_{\beta_{0,k1}}$ may depend on $\underline{s}$ only, nodes $N_{\beta_{1,1}}$ to $N_{\beta_{1,k2}}$ on $\underline{s}$ and N_{α_1}, etc. and therefore, when the turn of each β-nodes comes, the node is a leaf.

Let $\underline{s}_{\alpha}$ be the input string immediately after $N_{\alpha_{j-1}}$ is reduced. From property a.II follows that $\underline{s}_{\alpha}$ and $\underline{s}_{\alpha_{j-1}}$ have the same α-matches. In the same way it can be shown that $\underline{s}_{\alpha}$ has the same α-matches as $\underline{s}_{ij}$, which implies that M and M_A are the same. QED lemma 2.II

The proof of theorem IIa is similar to the proof of theorem Ia.

Consider now the application of G_2, $\mathcal{E}_0$ to $\underline{s}$. At each step we reduce either a β-match or the leftmost highest priority α-match. At each such reduction we remove the corresponding leaf from τ (lemma 2.II). The process cannot terminate as long as leaves are available since to each leaf there corresponds a match (lemma 2.II). Thus, on termination we are left with the root symbol only. QED IIa.

Corollary: Let $\underline{s}_1$ be a string derived from $\underline{s}$ by removing one β-leaf of τ or the leftmost highest priority α-leaf and reducing corresponding match. In the former case $\underline{s}_1$ is a sentential form of G_1, $\mathcal{E}_0$, in the second case $\underline{s}_1$ is a sentential form of G_1, $\mathcal{E}_{\alpha_1}$.

At this point we would like to extend the results of theorems Ia and IIa to include rules of the form

$$R_i \; N_i \quad \underline{\alpha}_i\underline{\pi}_i\underline{\beta}_i \rightarrow \underline{\alpha}_i\underline{\rho}_i\underline{\beta}_i \;|\; C_i(\underline{\alpha}_i, \underline{\beta}_i, \mathcal{E}) \;|\; P_i,$$

i.e. the patterns of the rules may contain string variables.

The main change is in the definition of the syntax-tree. The difficulty is illustrated in figure 3. Assume that $\underline{s}$ of figure 3 is a sentential form of some grammar G which contains the rules R_i and R_j and it is also a sentential form of any grammar derived from G by changing the priority levels of the rules. Observe that in figure 3(a) and (b) one of the matches is not a leaf and thus the lemmas on which the proofs of theorems Ia and IIA are based are not useful. We change the definition of the tree as illustrated in figure 3(c). Only $\underline{\pi}_i$ and $\underline{\rho}_i$ appear as the 'bottom' and 'top' parts of a leaf.

Let ξ_1 and ξ_2 be subsets of the alphabet.

Consider a rule of the form

$$R_i \quad N_i \quad \underline{\alpha}_i \underline{\pi}_i \underline{\beta}_i \rightarrow \underline{\alpha}_i \underline{\rho}_i \underline{\beta}_i \mid C_i(\underline{\alpha}_i, \underline{\beta}_i, \mathcal{E}) \mid P_i \quad ;$$

C_i is said to be <u>sensitive</u> to ξ_1 for $\underline{\alpha}_i$ and to ξ_2 for $\underline{\beta}_i$ if $C_i(\underline{\alpha}_i, \underline{\beta}_i)$ is <u>true</u> only for some $\underline{\alpha}_i$ and $\underline{\beta}_i$ which satisfy the following conditions:

1) Either all symbols of $\underline{\alpha}_i$ are elements of ξ_1, or, $\text{head}_1(\underline{\alpha}_i)$ is <u>any</u> symbol not in ξ_{1i}, $\xi_{1i} \subset \xi_1$, and all other symbols are elements of ξ_1.
2) Either all symbols of $\underline{\beta}_i$ are elements of ξ_2, or, $\text{tail}_1(\underline{\beta}_i)$ is <u>any</u> symbol not in ξ_{2i}, $\xi_{2i} \subset \xi_2$, and all other symbols are elements of ξ_1.

Note that $C_i(\underline{\alpha}, \underline{\beta})$ = <u>true</u> implies that $\underline{\alpha}$ and $\underline{\beta}$ satisfy the conditions; but $\underline{\alpha}$ and $\underline{\beta}$ which satisfy the conditions do not imply that $C_i(\alpha, \beta)$ is <u>true</u>.

Example: Let $\xi_1 = \xi_2 = \{*, +\} \qquad \xi_{1i} = \xi_{21} = \{*\}$;

$$R_i \; N_i \; a_1 a_2 \underline{\pi}_i b_1 \rightarrow a_1 a_2 \underline{\pi}_i b_1 \quad \Big| \; \text{if } a_2 = \text{'*'}, \text{ and } a_1 \neq \text{'*'} \; \underline{\text{and}} \; b_1 \neq \text{'*'} \; \underline{\text{then}} \; \underline{\text{true}} \; \underline{\text{else}} \; \underline{\text{false}}.$$

We would like to adapt a wider definition for the concept of overlap between two rules. The new definition does not change the proof of theorems Ib and IIb but extends the domain of grammars for which these theorems are valid.

Consider the following example: The rules of the grammar G_3 are

$R_1 \quad 0 \quad X\underline{\beta}_1 \rightarrow Y\underline{\beta}_1 \mid$ If β_1 = '*' then <u>true else false</u>;

$R_2 \quad 0 \quad \underline{\alpha}Z \rightarrow \underline{\alpha}_2 Y \mid$ If α_2 = '*' then <u>true else false</u>;

$R_3 \quad 0 \quad Y * Y \rightarrow S$;

R_1 and R_2 overlap. The string X*Z is a sentential form of G_3 which includes a R_1 match and R_2 match. Apply G_3 to X*Z and construct the syntax tree (new definition). The two matches have corresponding leaves. Now consider the proof of theorem Ia. The proof of Ia uses the overlap condition to prove that to each match there corresponds a leaf. Considering the example and the proof of Ia and Ib it seems that if the overlapping of the input patterns of two rules involves the $\underline{\alpha}_i$ $\underline{\beta}_i$ strings only then such an overlap can be permitted and with some additional conditions the theorems remain valid.

Consider rules of the form

$$R_i \; N_i \; \underline{\alpha}_i\underline{\pi}_i\underline{\beta}_i \rightarrow \underline{\alpha}_i\underline{\rho}_i\underline{\beta}_i \;\Big|\; C_i(\alpha_i, \beta_i, \mathcal{E}) \;\Big|\; P_i$$

which are sensitive to ξ_1 and ξ_2.

To simplify the definition we define the funcitons M_α and M_β :

If $\text{head}_1(\underline{\alpha}_j)$ is <u>any</u> element of $\bar{\xi}_{1j}$ then $M_\alpha(\underline{\alpha}_j)$ = <u>true</u> <u>else</u> $M_\alpha(\underline{\alpha}_j)$ = <u>false</u>. (The bar denotes the complement of a set).

If $\text{tail}_1(\underline{\beta}_j)$ is <u>any</u> element of $\bar{\xi}_{2j}$ then $M_\beta(\beta_j)$ = <u>true</u> <u>else</u> M_β = <u>false</u>.

R_i and R_j are said to <u>overlap (type 1)</u> if there exist $\mathcal{E}_0$, $\underline{\alpha}_i'$, $\underline{\beta}_i'$, $\underline{\alpha}_j'$, $\underline{\beta}_j'$ such that

1) $C_i(\underline{\alpha}_i', \underline{\beta}_i', \mathcal{E}) = C_j(\underline{\alpha}_j', \underline{\beta}_j', \mathcal{E})$ = <u>true</u>;

2) there exists a number m such that either

(2.1) $\text{tail}_m(\underline{\pi}_i\underline{\beta}_i') = \text{head}_m(\underline{\alpha}_j'\underline{\pi}_j)$

where $m \leq m_0$ and m_0 is given by the following table:

	$M_\alpha(\underline{\alpha}'_j)$ = true	$M_\alpha(\underline{\alpha}'_j)$ = false
$M_\beta(\underline{\beta}'_i)$ = true	$\min(\ell(\underline{\beta}'_i) + 1, \ell(\underline{\alpha}'_j) + 1)$	$\min(\ell(\underline{\beta}'_i), \ell(\underline{\alpha}'_j) + 1)$
$M_\beta(\underline{\beta}'_i)$ = true	$\min(\ell(\underline{\beta}'_i) + 1, \ell(\underline{\alpha}'_j)$	$\min(\ell(\underline{\beta}'_i), \ell(\underline{\alpha}'_j))$

or,

(2.2) same as (2.1) with j and i interchanged.

Example of overlap (type 1): Grammar G includes the following rules whose predicates are sensitive to $\xi_1 = \{Z_1\}$ in $\underline{\alpha}_i$ and $\xi_2 = \{Z_1, Z_2, Z_3\}$ in $\underline{\beta}_i$

R_1 0 $\alpha_1 AB \rightarrow \alpha_1 C$ | If α_1 = 'Z_1' then true else false;

R_2 0 $C\underline{\beta}'\underline{\beta}'' \rightarrow D\underline{\beta}'\underline{\beta}''$ | If (β' = 'Z_1') and ($\beta'' \neq$ 'Z_2') then true else false

For $\alpha_1 = Z_1$ $\beta' = Z_1$ there is overlap (type 1) in the following way:

R_1 :	$\vert Z_1 \vert AB$	$(\alpha_1 = Z_1)$
R_2 :	$CZ_1 \vert \underline{\beta}'' \vert$	$(\beta' = Z_1, \beta'' = Z_1, \beta'' \neq Z_2)$
R_2 :	$C \vert Z_1 \vert \underline{\beta}''$	$(\beta'' = A, \beta'' \neq Z_2)$.

Two rules R_i and R_j with input patterns $\underline{p}_i(\xi_i)$ and $\underline{p}_j(\xi_j)$ overlap if there exist two values ξ'_i and ξ'_j and a value of the environment such that (1) $\underline{p}_i(\xi'_i)$ overlaps $\underline{p}_j(\xi'_j)$ and both $C_i(\underline{p}_i(\xi'_i), \mathcal{E})$ and $C_j(\underline{p}_j(\xi'_j), \mathcal{E})$ are true; and (2) the overlap of R_i and R_j is not of type 1.

The above definition of overlap is the one used in the sequel.

Theorem Ib

Let G_1 be a grammar and let ξ_1 and ξ_2 be subsets of the alphabet. Let the rules of G be of the form

$R_i\ N_i\ \underline{\alpha}_i\underline{\pi}_i\underline{\beta}_i \rightarrow \underline{\alpha}_i\underline{\rho}_i\underline{\beta}_i \mid C_i(\underline{\alpha}_i, \underline{\beta}_i)$

and satisfy the following conditions: For all rules R_i

(1) the predicates are sensitive to ξ_1 for $\underline{\alpha}_i$ and to ξ_2 for $\underline{\beta}_i$ only;

(2) the symbols of ξ_1 and ξ_2 do not appear in $\underline{\rho}_i$;

(3) $\text{tail}_1(\underline{\pi}_i) \notin \xi_1$ and $\text{head}_1(\underline{\pi}_i) \notin \xi_2$;

(4) the predicates do not depend on the environment;

(5) R_i is not 'dotted' ;

(6) the root symbol appears in one rule only and only in the output pattern of that rule;

(7) the strings $\underline{\rho}_i$ and $\underline{\pi}_i$ are not empty.

Let G_2 be a grammar derived from G_1 by changing the priority levels of the rules.

If the rules of G_1 do not overlap each other nor themselves, then any sentential form of G_1 (G_2), $\mathcal{E}_0$, is a sentential form of G_2 (G_1), $\mathcal{E}_0$.

Proof

The proof has the same structure as the proof of theorem Ia. We use algorithm AI and lemmas 1.I and 2.I as stated in the proof of theorem Ia but with the terms 'tree' and 'leaf' having their modified definitions which admit string variables and which are discussed above.

Consider AI and let $\underline{s}$, $\underline{s}_i$, τ and τ_i be defined as in the proof of theorem Ia (bearing in mind the differences in the definitions mentioned above). Our first step is to consider step 2 of AI and show that it can be executed for any n. This is equivalent to proving that to any leaf of τ_i there corresponds a match in $\underline{s}_i$.

Let us introduce a small modification in AI and let the resulting algorithm be called AI*. Let the leaf chosen in step 2 corresponds to the replacement node N_i and let the corresponding rule be

$$R_i \; N_i \; \underline{\alpha}_i\underline{\pi}_i\underline{\beta}_i \rightarrow \underline{\alpha}_i\underline{\rho}_i\underline{\beta}_i.$$

In AI we reduce in $\underline{s}_i$ the match which corresponds to the leaf. In AI* we replace the string $\underline{\pi}_i$, which in $\underline{s}_i$ corresponds to N_i, by the string $\underline{\rho}_i$. The point, of course, is that AI* is not dependent on having a match corresponding to any leaf. Therefore, (1) the transformations of τ and $\underline{s}$ into τ_n and $\underline{s}_n$ are well defined for any n and (2) given a large enough n, AI* transforms $\underline{s}$ into the root symbol. We are going to show that under the conditions of the theorem any reduction of AI* corresponds to a reduction of a match (which implies that AI and AI* are the same).

Consider any replacement node N_j of τ. Let the corresponding rule be $R_j \; N_j \; \underline{\alpha}_j\underline{\pi}_j \rightarrow \underline{\alpha}_j\underline{\rho}_j$ (For simplicity we assuem $\underline{\beta}_j$ to be empty, since the proof with both $\underline{\alpha}_j$ and $\underline{\beta}_j$ non empty is essentially the same as the one given below). When G_1 is applied to the input string the actual reduction which corresponds to N_j is $\overline{Z}Z_1Z_2\underline{\pi}_j \quad \overline{Z}Z_1Z_2\underline{\rho}_j$ where $Z_1, Z_2 \in \xi_1$ and $\overline{Z} \notin \xi_{1j}$ (again the same proof can be extended to any non-zero number of symbols in $\underline{\alpha}_j$).

Consider any series of reductions of $\underline{s}$ and τ to the root symbol using AI*. Let our node N_j be the j-th replacement node to be removed. We claim that (a.1) for any series of reduction and for any $i \leq j$, Z_1 and Z_2 appear in $\underline{s}_i$ in the form $\ldots aZ_1Z_2\ldots$ where a is some symbol such that $a \notin \xi_{1j}$; (a.2) If N_j is a leaf of τ_i, $i \leq j$, then Z_1 and Z_2 appear in the form $\ldots aZ_1Z_2\underline{\pi}_i\ldots$.

Consider the following properties of $\underline{s}$ and of strings $\underline{s}_i$ derived from $\underline{s}$ by AI*. (Again, for simplicity, Z_1 and Z_2 are used instead of any two symbols of ξ_1 or ξ_2. In the following, Z_1 in $\underline{s}_i$ and Z_1 in $\underline{s}_j$ are the same occurrences of the same symbol; and the same remark holds for Z_2).

1) If Z_1 and Z_2 appear in $\underline{s}_i$ then they also appear in $\underline{s}$. This is a result of condition (2) of the theorem Ib.

2) If $\underline{s}_i = \ldots Z_1 Z_2 \ldots$ then AI* cannot transform it to $\underline{s}_j = \ldots Z_1 \underline{\alpha} Z_2 \ldots$ where $\underline{\alpha}$ is not empty (conditions (2) and (7), $\underline{\pi}_i$ is not empty).

3) If $\underline{s}_i = \ldots Z_1 \underline{\alpha} Z_2 \ldots$ then AI* cannot transform it to $\underline{s}_j = \ldots Z_1 Z_2 \ldots$. Since Z_1 and Z_2 cannot appear in a $\underline{\pi}_i$ without disappearing, then for any i, the reduction of $\underline{s}_i$ to $\underline{s}_j$ implies the reduction of $\underline{\alpha}$ to the empty string which contradicts condition (7).

4) If $\underline{s}_i = \ldots Z_3 Z_1 Z_2 \ldots$, where $Z_3 \in \xi_{1j}$ then AI* cannot reduce $\underline{s}_i$ to $\underline{s}_j = \ldots Z_1 Z_2 \ldots$ (Z_3 is reduced and removed from the string). Conceivably, this reduction might be done in one of two ways: $\underline{s}_i \Rightarrow^* \ldots Z_3 \ldots Z_1 Z_2 \ldots \Rightarrow^* \ldots Z_1 Z_2 \ldots$ which contradicts condition (7); $\underline{s}_i \Rightarrow^* \ldots Z_1 Z_2 \ldots$ which contradicts condition (3), and which proves the assertion.

5) At the creation of N_j by G_1 the input string was $\ldots \bar{Z} Z_1 Z_2 \underline{\pi}_j \ldots$.

From (1) (3) (4) and (5) follows that $\underline{s} = \ldots A Z_1 Z_2 \ldots$ where $A \notin \xi_{1j}$. From this result and from (2) and (3) follows that any $\underline{s}_i$ which contains both Z_1 and Z_2 is of the form $\underline{s}_i = \ldots B Z_1 Z_2 \ldots$ where $B \notin \xi_{1j}$.

It remains to show that as soon as N_j becomes a leaf the input string is $\underline{s}_k = \ldots B Z_1 Z_2 \underline{\gamma}\ \underline{\pi}_j \ldots$ with $\underline{\gamma}$ empty. Let the symbol node of Z_2 (Z_1) be connected by one branch to the replacement node N_{Z2} (N_{Z1}) where N_{Z1} and

N_{Z2} may be the same node. Since at the creation of N_j by G_1 the input string is $\ldots\overline{Z}Z_1Z_2\underline{\pi}_j\ldots$ it follows from condition (3) of the theorem that N_{Z2} depends on N_j (a directed path exists from N_j to N_{Z2}) and N_{Z1} (if different from N_{Z2}) depends on N_{Z2}.

Now assume that for some k, $\underline{s}_k = \ldots BZ_1Z_2\underline{\gamma}\ \underline{\pi}_j\ldots$ with $\underline{\gamma}$ non-empty. Consider the reduction of $\underline{s}_k$ to the root symbol using AI*. In this reduction N_{Z2} is reduced after N_j. Since both N_{Z2} and N_j do not depend on $\underline{\gamma}$ ($\underline{\gamma}$ was not there when N_{Z2} was created) $\underline{\gamma}$ has to be reduced to the empty string before N_{Z2} is reduced which contradicts condition (7) of the theorem.

This completes the proof of (a.1) and (a.2). There is only one more case to be considered: N_j is associate with a rule $R_j\ \underline{\alpha}_j\underline{\pi}_j \rightarrow \underline{\alpha}_j\underline{\pi}_j | C_j(\alpha_j)$ and at the application of G_1 to the input string the actual reduction which corresponds to N_j is $\overline{Z}\underline{\pi}_j \rightarrow \overline{Z}\underline{\rho}_j$ where $\overline{Z} \notin \xi_{1j}$. It has to be proved that if N_j is a leaf of some $\underline{s}_k$, $\underline{s}_k$ cannot be $\ldots Z\underline{\pi}_j\ldots$ where $Z\varepsilon\xi_{1j}$ and $\underline{\pi}_j$ is the string which corresponds to N_j.

Assume that there exists $\underline{s}_k\ \ldots Z\underline{\pi}_j\ldots$. Since AI* reduces $\underline{s}_k$ to the root symbol, it follows from condition (3) that the replacement node N_Z (the replacement node to which the symbol node of Z is connected by one branch) depends either on N_j or on nodes which do not depend on N_j but which depend on symbols to the right of $\underline{\pi}_j$. The second possibility implies that $\underline{\pi}_j$ can be reduced to the empty string which contradicts condition (7) of theorem Ib. The first possibility implies that in the application of AI* N_Z is reduced after N_j and both Z and $\underline{\pi}_j$ are present in the input string when N_j is created by G_1. Let $\underline{s}_{j-1}$ be the input string at that point, $\underline{s}_{j-1} = \ldots Z\underline{\gamma\pi}_j\ldots$ where $\text{tail}_1(\underline{\gamma}) = \overline{Z}$. From the format of $\underline{s}_k$ follows that N_Z does not depend on $\underline{\gamma}$ directly (no connection of $\underline{\gamma}$ and N_Z is done by one branch only). No node which depends on $\underline{\pi}_j$ can depend on $\underline{\gamma}$ since such a node has to be reduced after N_j which implies a non-empty string between Z and $\underline{\pi}_j$ in $\underline{s}_k$. Thus, in the reduction of $\underline{s}_{j-1}$ to the root symbol $\underline{\gamma}$ has to be reduced to the empty string which contradicts condition (7) of the theorem and rules out the first possibility.

From the above discussion follows that to every leaf of τ_i there corresponds a match in $\underline{s}_i$. Therefore we can return to the original definition of algorithm AI and prove lemmas 1.I and 2.I in the same way as in the proof of theorem Ia. Next, the proof of theorem Ib follows from AI and lemma 2.I in the same way as the proof of theorem Ia. QED Ib.

Corollary: Let $\underline{s}_1$ be the string derived from $\underline{s}$ by removing one leaf of τ. $\underline{s}_1$ is a sentential form of G_1.

Before presenting theorem IIb we have to define the sensitivity of P_i, the program part of a rule R_i, to ξ_1 and ξ_2. For simplicity assume ξ_2 is empty. If the predicate of R_i is sensitive, say, to ξ_1, C_i is true for some $\underline{\alpha}_i = \overline{Z}Z_1\ldots Z_n$ where $\overline{Z}$ is <u>any</u> element not in ξ_{1i}, $\xi_{1i} \subset \xi_1$. We allow the value of P_i to depend on the environment, on $\underline{\pi}_i$ and on $\underline{\alpha}_i$ and $\underline{\beta}_i$ but the dependency on $\underline{\alpha}_i$ and $\underline{\beta}_i$ has to be such that P_i is independent of the actual value of $\overline{Z}$. If these conditions are satisfied with respect to $\underline{\alpha}_i$ and $\underline{\beta}_i$ we say the $\underline{P}_i$ is sensitive to ξ_1 and ξ_2.

__Theorem IIb__

Let ψ_1 ψ_2 ψ_3, ξ_1 and ξ_2 be subsets of the alphabet where ψ_1, ψ_2 and ψ_3 are disjoint.

Let G_1 be a grammar consisting of two sets of rules α and β where the rules have the following form

$$R_i \; N_i \quad \underline{\alpha_i}\underline{\pi_i}\underline{\beta_i} \rightarrow \underline{\alpha_i}\underline{\rho_i}\underline{\beta_i} \;\; | \;\; C_i(\underline{\alpha_i}, \underline{\beta_i}, \mathcal{E}) \;\; | \;\; P_i$$

and which satisfy the following conditions:

For all rules R_i

1) the predicates and programs of all rules are sensitive to ξ_1 for $\underline{\alpha_i}$ and to ξ_2 for $\underline{\beta_i}$;

2) symbols of ξ_1 and ξ_2 do not appear in ρ_i;

3) $\text{tail}_1(\pi_i) \notin \xi_1$ and $\text{head}_1(\pi_i) \notin \xi_2$;

4) the rules are not 'dotted';

5) the strings $\underline{\pi_i}$ and $\underline{\rho_i}$ are not empty;

α rules: 1) The input-patterns consist of symbols of ψ_1 and ψ_2 only; output-patterns consist of symbols of ψ_2 only;

2) The α-rules do not overlap any rule of β;

3) The α-rules do not contain the root symbol.

β rules: 1) The input-patterns consist of symbols of ψ_2 and ψ_3 only; the output-patterns consist of symbols of ψ_3 only;

2) The β-rules satisfy all requirements imposed on the rules in theorem Ib;

3) The β-rules do not change the environment.

Let G_2 be a grammar derived from G_1 by changing the priority levels of the β-rules. Any sentential form of $G_1(G_2)$, $\mathcal{E}_0$, is a sentential form of $G_2(G_1)$, $\mathcal{E}_0$.

Proof:

The proof of theorem IIb closely follows the pattern established by theorems Ia, IIa, and Ib.

We first note that the conditions on which properties a.II, b.II, and c.II of the proof theorem IIa are based are included in theorem IIb and there are not additional conditions which limits these conditions' scope. Therefore, properties a.II, b.II, and c.II remain valid.

Next we define algorithm AII* in a way which is similar to the definition of AI* in the proof of theorem Ib. Conditions (1), (2), and (3), and (7) of Ib are included in IIb and are satisfied by both α- and β-rules. Therefore, as soon as a replacement node becomes a leaf, the appropriate $\underline{\alpha_i}$ and $\underline{\beta_i}$ appears adjacent to the left and right of $\underline{\pi_i}$. For $\underline{\alpha_i \pi_i \beta_i}$ to be a match, however, the environment has to have the correct value. Note now the AII* reduced the α-nodes in the same order as G_1 and that from condition (1) of IIb (programs) and the above discussion it follows that to every β-leaf and to N_{α_j} of τ_{ij} there corresponds a match. This implies that we can use AII rather than AII* and that lemma 1.II is valid in the case of theorem IIb as well.

Lemma 2.II follows from lemma 1.II and properties a.II, b.II, and c.II in the same way as in the proof of IIa. Once again the proof of the theorem IIb follows from the lemmas in a way similar to the proof of Ia, Ib, and IIb. QED IIb

3.4 Appending and Composition of MMA's

The appending and the composition of MA's are defined in Chapter 1 of Galler and Perlis [3] and as far as possible their notations are followed here.

Given two MA's, MA_1 and MA_2, which operate on the same alphabet, the appending of MA_2 to MA_1 results in the MA MA_3 which contains the rules of MA_1 and the rules of MA_2 ordered in the following way: first come all the rules of MA_1 in the same order as they appear in MA_1; next come all the rules of MA_2 in the same order as they appear in MA_2. The notation used for the result of appending MA_2 to MA_1 is $MA_1; MA_2$.

The composition of MA_1 and MA_2 is defined to be the following procedure: Apply MA_1 to the input string; On termination of MA_1 apply MA_2 to the string which resulted from the application of MA_1 to the input string. The composition of MA_1 and MA_2 is denoted by $MA_2 \circ MA_1$.

We need a concept related to composition which is best illustrated with reference to Figure 4. MA_1 (MA_1 box) is first applied to a string. On termination MA_2 is applied to the result (control is passed to box MA_2). On termination of MA_2 box 3 is activated. The operation of box 3 is the following: On the first visit to the box it transfers control to the MA_1 box. From the second visit onwards control is transferred to the MA_1 box only if a reduction has been made between the current the previous visits to the box; if no such reduction has been made box 3 terminates the process. We name this composition feedback composition and denote it by $[MA_2 \circ MA_1]$.

It is clear that the above definitions can be extended to several MA's. The extension to MMA is straightforward-the only addition required is that both MMA_1 and MMA_2 (or all algorithms concerned) have the same environment. With obvious meaning we can speak about the appending and the composition of grammars which have the same symbol as a root and which have the same environment $\mathcal{E}$.

We are concerned with the following problem: given two MMA's, $G_1, \mathcal{E}$ and $G_2, \mathcal{E}$, construct a MMA which is equivalent to $G_2 \circ G_1, \mathcal{E}$, (or to $[G_2 \circ G_1], \mathcal{E}$).

A standard procedure for finding an equivalent to $MA_2 \circ MA_1$, where MA_1 and MA_2 are MA's, is given in Chapter 1 of [3]. Essentially this procedure increases the alphabet, ξ, by adding a set of symbols ξ' such that to every symbol in ξ corresponds one and only one symbol in ξ' and vice versa. MA_2' is formed by changing each symbol in the rules of MA_2 to the corresponding symbol in ξ'. MA_1 is changed to MA_1' by introducing some new rules and eliminating the 'dots'. The effect of the changes is that MA_1' terminates on strings on which MA_1 terminates; however, before termination all symbols of ξ are changed to the corresponding symbols in ξ'. MA_3 consists of the appending of MA_2' to MA_1'.

It turns out that under the conditions which are essentially those imposed on the rules is theorems I and II $[MMA_2 \circ MMA_1]$ is equivalent to $MMA_1;MMA_1$, where equivalence here means that any sentential form of one is a sentential form of the other and vice versa. The proof of this fact is our next objective.

In the following, all MMA's are defined over the same alphabet and one symbol is designated as a root symbol so that the term 'sentential form' is meaningful.

The following lemma is useful although its proof is trivial.

<u>Lemma 3</u>:

Given are the MMA's $G_1, \mathcal{E}$, and $G_2, \mathcal{E}$. Let <u>s</u> and $\mathcal{E}$ be a string and an environment such that s̲ does not contain any matches to the rules of G_1 and contains at least one match to a rule of G_2.

If for any s̲ and $\mathcal{E}$ satisfying the above requirement no reduction by any rule of G_2 can introduce a match to a rule of G_1 then

(1) any sentential form of $G_2 \circ G_1$ is a sentential form of $G_1;G_1$ and vice versa.

(2) $[G_2 \circ G_1] = G_2 \circ G_1 = G_1;G_2$ where the equality means equivalence between the algorithms.

Theorem IIIa

Let $G_1, G_2, \ldots, G_n$ be MMA's all defined on the same $\mathcal{E}$. If $G_1;\ G_2;\ \ldots;\ G_n$ satisfies the conditions imposed on the rules of the grammar in theorem Ib, then

(a) any sentential form of $[G_n \circ \ldots \circ G_2 \circ G_1]$, $\mathcal{E}$, is a sentential form of $G_1;G_2;\ldots;G_n$, $\mathcal{E}$, and

(b) any sentential form of $[G_n \circ \ldots \circ G_2 \circ G_1]$, $\mathcal{E}$, is a sentential form of $G_1;G_2;\ldots;G_n$, $\mathcal{E}$.

Proof

For simplicity we consider the composition and appending of two MMA's only. First note that under the conditions of theorem Ib the environment does not play any role in the reduction process and therefore it can be ignored.

Part (a): Let s̲ and τ be the sentential form and syntax tree which corresponds to $[G_2 \circ G_1]$ (or, in the proof of part (b), to $G_1;G_2$). Let algorithm AI* operate on s̲ and τ and let the result be $\underline{s}_n$ and τ_n. In exactly the same way as in the proof of Ib we extablish that to any leaf of τ_n there corresponds a match in $\underline{s}_n$ and thus prove lemma 1.I, which, in turn is used to prove lemma 2.I.

Applying $G_1;G_2$ $[G_2 \circ G_1]$ for part (b) to $\underline{s}$ the termination with the root symbol follows directly from lemmas 1.I and 2.I, in the same way as in the proof of theorem Ib. QED

Theorem IIIb

Let $G_1, \ldots, G_n$ be MMA's all defined on the same environment $\mathcal{E}$.

If $G_1; G_2; G_3 \ldots; G_n$ satisfy the conditions on theorem IIb with the rules of G_1 the α-rules and $G_2 \ldots G_n$ the β-rules, then

(a) any sentential form of $[G_n \circ G_{n-1} \circ \ldots \circ G_2 \circ G_1]$, $\mathcal{E}$, is a sentential form of $G_1; G_2; \ldots; G_n$, $\mathcal{E}$, and

(b) any sentential form of $G_1; G_2; \ldots; G_n$, $\mathcal{E}$, is a sentential form of $[G_n \circ G_{n-1} \circ \ldots \circ G_1]$, $\mathcal{E}$.

Proof:

The first thing to note is that the conditions on the α-rules are such that after G_1 is applied and terminates no application of the β-rules can introduce α-matches to the string. Thus we can consider $[G_n \circ G_{n-1} \circ \ldots \circ G_2] \circ G_1$ rather than $[G_n \circ \ldots \circ G_1]$. Using theorem IIIa we get that the former is equivalent (from a sentential form point of view) to $G_2; G_3; \ldots; G_n$ composed with G_1 and from lemma 3 follows the equivalence with $G_1; G_2; \ldots; G_n$.

QED.

3.5 Computing Time and Transformation to Equivalent Algorithms

The MMA in its original form seems to be (a) slow, and (b) to require the presence of the whole string in memory at the same time. This section is a short summary of efforts to reduce the computer time requirements by transforming the MMA to forms which are equivalent to the MMA from the point of view of computation results but requires less computer time. A complete discussion of the subject will be given elsewhere.

Markers are used to reduce computation time. It was proved [9] that, under the termination conditions given in 3.1 the number of operations performed by an MA which uses markers (without the predicate and the program part) is linearly bounded by the initial length of the input string.

The idea is rather simple and can be easily explained by an example. Figure 5 shows a string in which, say, the third rule has just made a replacement. We put two markers, a left $\underline{3}$rd marker and a right $\underline{3}$rd marker, at both ends of the new substring. Formally, our next step should be to start scanning with the first rule from the beginning of the string. The marker enables us to start with the $\underline{1}$st rule not from the beginning, but as indicated at figure. If no match if found, scanning by the $\underline{1}$st rule is finished not at the end, but at the second marker (See [9] for further details).

The MMA (with or without markers) can be transformed to a two stack algorithm, a fact which is not surprising in view of the equivalence between the MA and the Turing Machine. In the simplest version of the two stack algorithm stack A is initially empty and the input string is in stack B with the left most symbol on top of the stack followed by the second symbol, etc. A symbol is read from B to A and the rules of the grammar are compared with the top of stack A. If a match to a rule R_i is found and some addition conditions (which are the essence of the transformation of the MMA to a two stack algorithm) are met, $\underline{p}_i$ is removed from the top of stack A and $\underline{q}_i$ is put on top of stack B. If $\ell(\underline{q}_i) = 1$, $\underline{q}_i$ can be put into stack A. Similar simplification occurs with rules of the type $\underline{\alpha}_i\underline{\pi}_i\underline{\beta}_i \rightarrow \underline{\alpha}_i\underline{\rho}_i\underline{\beta}_i$

The main advantage of the two stack algorithm seems to be in the search procedure. The table of rules does not change during the parsing process and can be coded efficiently in order to decrease the search for a match. The input string changes continuously and it is not clear how to code it. Thus it is faster to compare the symbols on top of the stack with all rules than to compare one rule with the whole input string.

The markers and the two stack algorithm are general transformations to f equivalent forms. It is clear that advantage can be also taken of special properties of some grammars. Examples are the following:

If all the rules of the grammar have the same priority and the rules do not overlap (or if equivalence form the point of view of sentential form is sought and conditions of theorem Ib are satisfied) then any match to the symbols on top of stack A can be reduced (no additional conditions are needed).

When applicable, (conditions of theorem IIb) computer time can be reduced by a change in the order of the rules. This aspect is discussed in section 4.5 in relation with the Algol example of section IV. It also turns out that a single stack and a queue are sufficient for the parsing of that example.

Advantage can be taken of the special cases to reduce the memory requirements; see section 4.6 for an example.

3.6 Relation with Bounded Context Grammars.

We are concerned with the relation between (m,n) bounded context grammars as defined by Floyd [13] (see also chapter 6 in [8]) and the MMA grammars in particular those which satisfy the conditions of theorem Ib.

In our notations an (m,n) bounded context grammar is a set of rules of the form

$$\underline{t}_i\underline{u}_i\underline{v}_i \rightarrow \underline{t}_i U_i \underline{v}_i$$

where U_i is a single symbol and where additional conditions are imposed on the rules such that for any i and for any sentential form whenever the substring $\underline{t}_i\underline{u}_i\underline{v}_i$ appears in the input string it can be replaced by $\underline{t}_i U_i \underline{v}_i$ with the result being a sentential form. The conditions are given in [13] page 63. The values m and n are given by $m = \max_i \ell(\underline{t}_i)$ and $n = \max_i \ell(\underline{v}_i)$. They play no rule in the discussion as long as they are finite.

From the above it is clear that a sentential form of any bounded context grammar is also a sentential form of an MMA whose rules have the form

$$R_i \quad N_i \quad \underline{t}_i\underline{u}_i\underline{v}_i \rightarrow \underline{t}_i U_i \underline{v}_i$$

where N_i is arbitrary.

Theorem IV

Let G be a grammar whose rules satisfy all the conditions of theorem Ib and the additional condition $\ell(\rho_i) = 1$ for all i. The rules of G (ignoring the priority levels) form a bounded context grammar.

Proof:

The above statement of the theorem requires some clarification.

Note that the concepts of predicate and environment are not used in [13]. The environment concept is not used by the rules of G which satisfy Ib. The predicate can be considered as a shorthand notation for several rules, e.g.

$\underline{\alpha\pi_1} \rightarrow \underline{\alpha\rho_1} | \underline{\text{if}}\ \underline{\alpha} \neq \text{'*'}\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

is replaced by several rules - a rule for each symbol in the alphabet which is different from '*'.

The proof consists of showing that conditions (1) (2) (3) of Ib imply each of the conditions R_{11} through R_{44} on page 65 of [13]. Since the technique of the proof are the same for any R_{ij} we give a detail proof of R_{11} only.

In our notations (see section 2.2) condition R_{11} is the following:

$((\exists\ \underline{x})\ (\underline{xy} \Rightarrow_* \underline{w})$ is abbreviated $\ldots\underline{y} \Rightarrow_* \underline{w})$

Condition R_{11}:

There does not exist a string $\ldots\underline{t}\ \underline{u_1}\underline{u_2}\underline{v}\ldots$ and

two rules $\quad R \quad N \quad \underline{t}\ \underline{u}\ \underline{v} \rightarrow \underline{t}\ U\ \underline{v}\ ,\ \underline{u} = \underline{u_1}\underline{u_2};$

$\qquad\qquad R_1 \quad N_1 \quad \underline{\alpha}\ \underline{w}\ \underline{u_1}\ \beta \rightarrow \underline{\alpha}\ X\ \underline{\beta}\ ;$

such that:

1) $\ldots\underline{t}\ \underline{u_1}\underline{u_2}\underline{v}\ldots \Rightarrow_* \ldots\underline{w}\ \underline{u_1}\underline{u_2}\underline{v}\ldots$, where $\ldots\underline{t} \Rightarrow_* \underline{w}$;

2) using rule R_1 $\ldots\underline{w}\ \underline{u_1}\underline{u_2}\underline{v}\ldots \Rightarrow \ldots X\ \underline{u_2}\underline{v}\ldots \triangleq \underline{\gamma}$;

3) $\underline{\gamma} = \ldots X\ \underline{u_2}\underline{v} \ldots \Rightarrow_* S$.

The proof shows that if (1) (2) and (3) hold then from the conditions of Ib it follows that R_1 and R overlap which contradicts the conditions of theorem Ib.

Consider the different possibilities under which $\ldots t \Rightarrow_* w$:

(a) If $\ldots\underline{t} = \underline{w}$ then

$\ldots \underline{w}\ \underline{u}_1\underline{u}_2\underline{v} \ldots = \ldots \underline{t}\ \underline{u}_1\underline{u}_2\underline{v} \ldots$ with R and R_1 matching two strings which overlap.

(b) If there exists a number n, $n \geq 1$, $\ell(\underline{t}) > n$, such that $\underline{w} = \ldots \text{tail}_n(\underline{t})$ and $\underline{w} \neq \ldots \text{tail}_{n+1}(\underline{t})$, then we have the following: From (1) and (3) of Ib follows that $\text{head}_1(\text{tail}_{n+1}(\underline{t})) \notin \xi_1$, and that $\ell(\underline{t}) = n+1$. From the above and (7) of Ib follows the $\ell(\underline{w}) \geq n+1$. Therefore both R and R_1 match $\ldots \underline{w}\ \underline{u}_1\underline{u}_2\underline{v} \ldots$ and these matches overlap.

(c) If $\text{tail}_1(\underline{w}) \neq \text{tail}_1(\underline{t})$ then from (1) and (3) of Ib follows that $\text{tail}_1(\underline{t}) \notin \xi_1$ and $\ell(\underline{t}) = 1$ and $\text{tail}_1(\underline{w}) \notin \xi_1$. Since $\ell(\underline{w}) \geq 1$, it follows that both R and R_1 match $\ldots \underline{w}\ \underline{u}_1\underline{u}_2\underline{v} \ldots$ and these matches overlap.

If $\underline{t}$ is empty, then the overlap of R_1 and R follows from (2) above.

Thus R_{11} cannot hold if the rules of G satisfy the conditions of Ib.

QED

In concluding this section we remark that it seems that the relation between the Context Bounded Grammars and the MMA's requires additional investigations. Note that condition (3) of R_{11} was not used in the proof and that we did not say that to every bounded context grammar corresponds an MMA whose rules satisfy the conditions of Ib.

Section IV: Discussion of the Algol Example

4.1 General

This section describes the methodology used in constructing an MMA parser for Algol 60. The complete description of the MMA grammar rules and the environment is given in the Appendix.

We have chosen Algol 60 as the subject of the example because the language is well known and well documented and thus it enables the comparison of the MMA technique with the conventional ones. A problem oriented language, as the graph language mentioned above [7], could have been chosen. This, however, would require both the description of the semantics of the new language and a description of the syntax using BNF (plus English) to facilitate the comparison between the techniques.

In the next subsection the general methodology of the synthesis of a parser is discussed. The method involves the partitioning of the language into sublanguages,constructing a parser for each part and then putting the parts together to form one MMA parser. The subparsers for expressions and declarations are discussed in 4.3 and 4.4 respectively. The parts are put together in 4.5. The effect of the reordering of the rules is discussed in 4.6.

4.2 The Synthesis Method

Assume that we have in mind a certain language; in this example, the language is Algol 60 but it can be any problem oriented language.

The synthesis of an MMA grammar for a language involves a basic problem. Any set of rules we write down defines a parser but does this parser define the language we have in mind? The answer, of course, depends on the original description of the language. In cases that this description is mathematical, then, conceivably, one can define an algorithm that transforms the language description to MMA grammar rules. In case that the description is partly BNF and partially English, as Algol 60, or partially BNF, partially vague and partially non-existant, as in a new problem oriented language, no such algorithm can be defined. If the language is of any complexity it is impossible to determine by inspection that the MMA grammar is indeed defining the same language as the one we have in mind.

The synthesis procedure suggested here is to divide the language to several small parts or sub-languages, e.g. declarations, arithmetic expression, etc. Each part is then provided with its own MMA parser. Composition is used to form the parser for the whole language from the parsers for the sub-languages. When it is so desired the next step is to use techniques discussed in section 3.4 to form an equivalent parser which uses one MMA.

The main advantage in the above procedure is that each of the parsers for the sub-languages contains only a few rules and their 'correctness' can be easily verified by inspection.

Several additional points can be made with respect to the application of synthesis procedure to our Algol example:

a) We do not have a formula or an algorithm for partitioning the language into sub-languages. In Algol the partitioning was done intuitively more or less the way it is done in an informal exposition of the language (see list in appendix A).

b) The synthesis of parsers for the sub-languages is rather simple, this is mainly a result of the fact that most of the syntactic constructs follow a few patterns. The common patterns are:

(b.1) A list-like pattern:

<first symbol><construct><separator><construct>

<separator>...<last symbol>.

Examples: A list: (<item>, <item>, <item>)

A block without nesting: <u>begin</u> D, D, S, S, . . . <u>end</u> .
See also the rules for processing declarations (Group D in Appendix A), for statements (Group F), assignment statement (Group AS).

This pattern is handled by recognizing the first symbol, substituting a marker for it and using the marker for collecting the items from left to the right. Example: Consider an (integer) array variable, ρ is the marker.

$<\text{arith variable}>\ [\ \rightarrow\ \rho$

$$\rho \left\{ \begin{matrix} < SE > \\ \\ < E > \end{matrix} \right\} , \rightarrow \rho$$

$$\rho \left\{ \begin{matrix} < SE > \\ \\ < E > \end{matrix} \right\}] \rightarrow \quad <\text{subscripted arith. variable}>$$

(b.2) Expressions: See group E and group BR in the appendix and the comments on expressions below.

(b.3) Nested structures:

In the list-like patterns nested structures appear when each of the < constructs > can be a list structure, e.g. blocks. In this case no special device is required and the technique of (b.1) processes such constructs automatically. Some care, however, has to be take with respect to the scope of variables; this case is discussed below in 4.3.

Another form of nesting appears in expressions where an expression can appear between parenthesis. The parentheses can be viewed as introducing a local change in the priority of the operators. This case is discussed in 4.2 below.

(c) The following fact greatly simplifies the synthesis of MMA for the sub-languages: It turns out that for most sub-parsers (all except expressions and declarations) the order of the rules is unimportant since the rules satisfy the conditions of theorem Ib.

4.3 Expressions

For simplicity let us first consider expressions using the operators ↑, * and + (no brackets) with the usual priority hierarchy associated with them. An arithmetic variable is denoted by X and <SE> denotes a simple (arithmetic) expression.

It is clear that the sublanguage can be defined by the following grammar:

A.1	4	X → <SE>
A.2	3	<SE>↑<SE> → <SE>
A.3	2	<SE>*<SE> → <SE>
A.4	1	<SE>+<SE> → <SE>

If parentheses are now introduced a rule of the form (<SE>) → <SE> has to be added. A simple example, (X+X)*X*X↑(X+X), shows that the addition of this rule is not enough and that rules A.2 through A.4 have to be modified in order to obtain a parse which corresponds to the usual meaning of arithmetic expressions.

B.1 5 X → <SE>

B.2 4 (<SE>) → <SE>

B.3 3 <SE>↑<SE> → <SE>

B.4 2 α<SE>*<SE>β → α<SE>β | if α ≠ '↑' and β ≠ '↑' then true else false;

B.5 1 <SE>+<SE>β → <SE>β | if α ≠ '↑' and α ≠ '*' then true else false;

Note that the α on the left of the input-pattern of B.4 takes care of situation such as (X+X)↑X*X. Such a α is not needed in B.5 for the following reason: when a string is processed by the grammar and a match to B.5 is found, no right parentheses can appear to the left of it. From the same argument follows that we can avoid the α in B.5 if we make B.1 the lowest priority rule. The result is

C.1 5 (<SE>) → <SE>

C.2 4 <SE>↑<SE> → <SE>

C.3 3 <SE>*<SE>β → <SE>β | if β ≠ '↑' then true else false;

C.4 2 <SE>+<SE>β → <SE>β | if β ≠ '↑' and β ≠ '*' then true else false;

C.5 1 X → <SE>

Expressions of the above type are an interesting special case as far as changing the order of the rules are concerned. We shall discuss the case with reference to the above simple example; the result, however, holds for the more general case of rules of groups E and BR in the appendix.

Let us change the priority of C.1 to 1 and introduce left and right markers for each priority level. Without loss of generality we can start scanning with C.5. Note the following: (1) After any replacement the string (plus markers) is ...LM <SE> RM... where LM and RM are the left and right markers, respectively, which are associated with priority level 1 (C.1 and C.5) (See [9] for exact discussion of handling markers). (2) Since no <SE> can appear before C.5 has processed the string there is no reason to check matches to C.2, C.3 and C.4 in positions which require <SE> to the right of RM. (Figure 8a). (3) From (1) and (2) follows that all matches to C.2, C.3 and C.4 are at positions shown in Figure 8b. Since the input-pattern of C.2, C.3, and C.4 are different they all can be taken as having the same priority.

The conclusion is that we can perform the reductions of the above MMA by using a stack and a queue of length 1. (Compare with Section 3.5). The stack contains all symbols to the left of RM. The queue contains the symbol to the right of RM. All matches and reductions are done on top of the stack with the value of β being the symbol in the queue. Comparisons between top of stack and the rules can be done in any order since only one rule can match the top of stack. If no match is found the symbol from the queue is transferred to the stack and a symbol from the input string is moved to the queue.

The result, of course, is not new but it is interesting to see that it can be readily obtained from the MMA using markers.

4.4 Declarations

If we ignore for a moment the declarations of arrays which may contain expressions then the structure of the declarations is quite simple. The declarations of variables may appear at the head of each block and their format is list-like which can be processed as discussed in Section 4.1a.

The nesting of blocks and the scope of variables is handled in the following way: When the beginning of a block is recognized, a symbol table is opened and made current. The newly declared variables are put into this table. The table is closed when the end of the declaration statements is reached (rule D.12). The following mechanism insures that each <identifier> is converted to the appropriate variable (<arith. variable> or <Boolean variable>) according to the correct scope. All rules of group D have the same priority and thus scanning by this group of rules is done from left to right. Processing of <identifier>'s inside a block starts after the block's declaration has been processed. When an identifier is encountered the search in the tables is done in the usual method for nested tables (the current table is searched first, its parent is searched next, etc.). A new table is opened and made current when the beginning of a nested block is recognized. When the end of this block is recognized the parent table becomes current again and any identifier which appears at the end of the block enjoys the correct scope.

Some minor complications result from the appearance of the expressions in the declaration of arrays. The expressions have to be processed but the variables which appear in them are defined in the <u>outside</u> blocks. For this reason the search in the symbol tables skips the current symbol table if this table is open (the symbol table is open and made current but it is 'connected' to the main nested symbol table only after it is closed).

Labels and label tables are handled similar to the way symbols and symbol tables are handled. The differences, however, are the following:

Since a label may be defined after it is used, the label table is being closed by the rule which tecognizes the end of the block (D.16). This implies that a current label table is searched for duplication in definitions but the parents table cannot be searched until the whole program is processed. The actual search is done by rule B.3 which is the rule which plants the root symbol into the input string.

4.5 Combining the Sub-Parsers to One Parser

This section has two objectives: The first one is to explain how the parsers of the sub-languages are put together to form the parser for the whole language which is described by figure 7. The second objective is to use the results of 3.4 to obtain one MMA parser from the composition of figure 7.

To facilitate the discussion we name the MMA's for the sub-languages in the following way: E - expressions, BR - Boolean and relational expressions, D - declarations; AS - assignment statement; F - for clause; S - statements; L - labels and B - blocks.

It is clear that the first operation which one has to do upon entering a block at the first time is to process the declarations. Indeed one approach to parsing is to process all the declaration of the program first in what is commonly called a first pass. The processing of declarations requires the parsing of expressions and thus E, BR and D are appended to form E;BR;D which corresponds to box 1 of figure 7.

The appending of BR to E follows from the priority of the operators. Declarations have to be processed from left to right in order to insure the correct scope of variables (see 4.4). To insure this, the rules of E;BR are given a higher priority than the rules of D.

The operation of E;BR;D is explained as follows: As soon as an <identifier> is reduced to <arith. variable> it is processed by the rules of E;BR and transformed, say, to <SE>. If this <SE> and symbols to its left can be further reduced by E;BR then this reduction takes place. Thus, an expression which appears in a declaration of an array is completely reduced to <SE> before the rules of D operate on any <identifier> which is to the left of this expression and which is not a part of it (see the example of figure 6).

Note that if the input string is a sentential form then after the termination of E;BR;D the input string does not contain any declarations, any expressions and any operators which appear in expressions (+, -, etc.). In addition <identifier>s have been reduced to other constructs.

Our next step is to process all assignment statement. This amounts to composing AS with E;BR;D to form AS∘(E;BR;D). We note now that any sequence of reductions by the rules of AS cannot introduce the symbols <u>begin</u>,<u>end</u>,<u>integer</u>,<u>Boolean</u>, +, -, ... etc., into the input string. Thus, the application of these rules cannot cause the appearance of a match to any rule in E;BR;D. From lemma 3 it follows that E;BR;D;AS can be used instead of AS∘(E;BR;D).

The <for clause>s can be now processed. Using a similar argument we can append the rules of F to the above result and form E;BR;D;AS;F which is denoted by M in the sequel.

If the initial string <u>s</u> is a sentential form then upon termination of M the input string may contain statements, blocks and labels (more precisely the following symbols may appear: <uncond. statement>, <u>if</u>, <u>then</u>, <u>else</u>, <SE>, <E>, <SB>, <B>, <u>B</u>', <u>end</u>' (for <u>begin</u> and <u>end</u>), <u>go to</u>,

<label>, FOR', ';', ':' end finish). Let us apply [L∘S] to the input string. Among other reductions that [L∘S] may do, it reduces to B'S; ...S; end' (or to B'S; ...; S end finish), where S stands for <statement>, any block which does not have any other block nested in it. If B is applied to the result, the nesting level is reduced at least by one. Thus, the operation we need is [B∘[L∘S]] (see figure 7).

Now, the rules of L S satisfy the requirement of theorem IIIa and thus S;L replaces [L∘S]. Using theorem IIIa again we get S;L;B for [B∘[L∘S]].

Our parser is now (S;L;B)∘M. Using lemma 3 we get M;S;L;B which is a parser implemented by one MMA.

4.6 Changes in the Order of the Rules

The rules of M;S;L;B satisfy all requirement of theorem IIb with the rules of E;BR;D the α-rules and AS;F;S;L;B the β-rules. Applying theorem IIb we get the parser in the form it appears in the appendix which is E;BR;D preceded by the rules of AS, F, S, L and B where all the latter rules having the same priority.

This arrangement has an interesting property. Parsing proceeds in the following manner:

Let M_1 = AS;F;S;L;B with all rules having the same priority. Without loss of generality we can start with the two markers of the D rules at the left of the input string. The first block is entered and declaration are processed by E;BR;D. Next, the first block statement is entered. As soon as a match to M_1 appears it is being reduced. Thus when the left D marker leaves a statement it is completely reduced to <statement>; as this marker leaves a (internal) block, the block is completely reduced to a <statement>. Thus generally speaking the parsing proceed from 'left to right' with every block left completely reduced.

The concept of 'left to right' parsing can be made precise. For this purpose let us put the markers on the left and right of the $\underline{\rho}_i(\underline{\alpha}_i \downarrow \underline{\rho}_i \downarrow \underline{\beta}_i)$ rather than on the left and right of the $\underline{\rho}_i$. In our example, the subset of Algol 60, whenever both the left and right D marker are present there is only one symbol between them (except for the first step) and these markers are present at all times. If we measure the progress of the parsing by the distance of the left marker from the right end of the string, then with each reduction this distance becomes smaller or remains constant.

The above change in the use of markers, and consideration similar to the ones used in 4.3 can be used to make all rules of this example of the same priority and use a stack and a queue to parse the input string.

Section V Summary

The theoretical part of the article uses the MMA to define the syntax of computer languages. The theorems present sufficient conditions for the equivalence (from a sentential form point of view) between two MMA's, one of which is derived from the other by changing the priority of the rules, and sufficient conditions under which feedback composition of MMA's can be obtained by appending of MMA's which satisfy the above conditions. Some general and some special techniques for the reduction of the parsing time are presented and the relation between the MMA (satisfying conditions of theorem IIa) and the (m,n) bounded context grammar of MMA are discussed.

We believe that the main value of the MMA and of the above theorems is in the synthesis of the syntax of programming languages. The Algol example given here and the graph language for which we have written an MMA parser indicate that writing of such a parser is relatively simple as compared with the use of BNF plus English. The simplicity is attributed to the following properties of the MMA and the Algol-like language:

1) The MMA uses a small number of rules and
2) A smaller number of non-terminal symbols.
3) The reduction in the number of rules and the number of non-terminal symbols is attributed to the use of the predicate and the environment.
4) To a large extent, the rules resemble the way that a programmer thinks about a language.
5) The synthesis of a grammar can be carried out by partitioning

the language into sub-languages, providing each sub-language with a parser and composing these parsers into a single parser.

6) In the examples which we have encountered, we have found (6.1) that often the order of the rules is not important; when the order is important it fits our own intuition of what has to be done first and what later. In addition (6.2) the composition of the sub-parsers involved the appending operation only.

7) The above properties, (6.1) and (6.2), are results of theorems I through III. It is fairly simple to check whether a grammar satisfies the above conditions. For small grammars this can be done by inspection.

8) The languages used in the examples contain a few standard constructs which appear again and again. In synthesizing a parser one has to provide a solution for each construct rather than to each occurrence of the construct in the grammar.

9) The MMA as it is defined provides a parser which is usually too slow to be of practical use. There are several ways in which improvement can be achieved. These ways can be general (Markers, two stack algorithm) or specific, taking advantage of some particular properties of the grammar (changing order of rules, a stack and a queue algorithm). It seems quite straight forward to construct an algorithm which accepts a description of the MMA, and checks for conditions under which transformations which reduce computer time can be carried out and if the conditions are satisfied find the exact transformation.

At this point it is interesting to compare the MMA as a language parser to the production language (chapter 7 in [8], [11]) as used as a parser, in FSL [12] for example. While the starting point is different the over all philosophy of the result is amazingly similar. In both cases a language is defined by the parser, the definition can involve and effect an environment, the parser can be constructed from parts which in the production language are put together by transfer instructions. The production language uses one stack only and if the storage in the 'action' part is bounded then the MA is a more general computation scheme. This can be changed by providing the production language with an additional stack which contains the input (stack B in 3.5) and to which symbols can be returned from the regular stack. We do not know, however, of a case where this lack of generality handicapped the usage of the production language.

References:

[1] Markov, A. A., Theory of Algorithms, Academy of Sciences of the USSR, 1954. English translation (by J. J. Schozz-Kon and staff of the Israel Program for Scientific Translation) is available from Office of Technical Services, U. S. Dept. of Commerce, Washington D. C.

[2] Mendelson, E., Introduction to Mathematical Logic, D. Van Nostrand Co., Inc., 1964.

[3] Galler, B. A. and Perlis, A. J., A View of Programming Languages, Addison-Wesley Publication Comp., Reading, Mass.

[4] Caracciolo Di Forino, A., String Processing Languages and Generalized Markov Algorithms, in 'Symbol Manipulation Languages and Techniques', Bobrow, D. G. (ed.), North-Holland, 1968.

[5] De Bakker, J. W., Formal Definition of Programming Languages, Mathematical Center, Amsterdam, 1970,(2nd ed.).

[6] Bell, J. R., The Design of a Minimal Expandable Computer Language, Doctoral Dissertation, Stanford University, 1968.

[7] Milgrom, E., Design of an Extensible Programming Language, Doctoral Dissertation, Technion - Israel Institute of Technology, 1971.

[8] Gries, D., Compiler Construction for Digital Computers, John Wiley & Son, Inc., 1971.

[9] Katzenelson, J., Markov Algorithm as a Language Parser - Linear Bounds, to be published, Journal of Systems and Computer Sciences.

[10] Naur, P. et al., Revised Report on the Algorithmic Language Algol 60, International Federation for Information Processing, 1962; also C. ACM, 6, Jan. 1963, 1 - 17.

[11] Floyd, R. W., A Descriptive Language for Symbol Manipulation, J. ACM 8, Oct. 1961, 579 - 584.

[12] Feldman, J. A., A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler, C. ACM, 9, (Jan. 1966).

[13] Floyd, R. W., Bounded Context Syntactic Anaylsis, C. ACM, 7, Feb. 1964, 62 - 67.

Appendix A: The MMA Syntax of a Subset of Algol 60

A.1 The Subset

The subset was derived from Algol 60 by omitting the following items: Procedures, functions, switches, designational expressions and numbers of type real.

The use of a subset is a result of the desire to keep the example simple. We believe that the inclusion of the above items is not complicated and does not require any additional technique.

A.2 Terminal Symbols and the Scanner

For simplicity we assume that the MMA operates on a input string which has been previously processed by a scanner. The input to the scanner is a string of letters, digitals, logical values and delimiters (see [10], section 2) minus the items omitted in A.1. The output of the scanner consists of the symbols:

<number>	(numbers or integer),
logical values	(true | false),
<identifier>	(any string of digits and letters that does not fall into the other categories. Each <identifiers> has a value which is called name and is the string of digits and letter that the scanner replaces by the particular <identifier>).
symbols representing delimiters	(+ | - | * | / | ÷ | ↑ | < | ≤ | = | ≥ | > | ≠ | ≡ | ⊃ | or | and | — | go to | if | then | else | for | do | , | : | ; | := | step | until | while | (|) | [|] | '|' | begin | end | Boolean | end.finish | integer | array).

The above symbols are the terminal symbols for our MMA grammar.

A.3 Non-terminal Symbols

The non-terminal symbols introduced are:

<statement>, <unconditional statement>, <program>, <label> <for statement>, <SE>, <E>, <SB>, <B>, <arithmetic variable> <Boolean variable>, <subscripted Boolean variable>, <subscripted arithmetic variable>,

B°, end', B', FOR', <left part>, ρ, ξ, B", B"', B^4.

The second group of symbols are actually markers in the sense of 4.2.

A.4 Groups of Rules

The format of the rules is described in section II. The names given to the rules are derived from the small MMA parsers which are later put together to form the complete grammar. Thus, the third rule in the small MMA which processes declaration is called D.3 and its priority is 4.

The Groups of Rules and their Functions are:

B	blocks
L	labels and go to statements
S	statements
F	for statements
AS	assignment statements
A	Arithmetic Expressions
BR	Boolean and Relational Expressions
D	Declarations.

A.5 <u>The Environment</u>

The environment consists of two tables: a symbol table and a label table. Both tables are nested.

Each entry in the symbol table consists of two components: The value of the first component is a name of an identifier and the value of the second one is the identifier type (integer, Boolean integer array or Boolean array in this subset). If an entry corresponds to an array then the entry has components which contain the dimension and the upper and the lower bounds of the array.

Each item in the label table has two components. The value of the first component is the name of a (label) <identifier>. The second component contains either the string 'defined' or the string 'undefined' with the obvious meaning.

Both symbol and label tables are nested tables. At any time, in each nested structure one (sub) table is current.

There are two variables which can have the value 'open' or the value 'closed'. Each variable is associated with a table and indicate whether the current table is open or closed.

An additional variable called T whose value can be a type is used by the declarations.

Variables which are used to count the dimension of arrays and number of subscripts of subscripted variables have to be provided. For simplicity, however, we do not refer to them in the program part of the rules.

Initially, at the start of the parsing, the tables are empty and each consists of one (sub) table only which is closed.

We assume the following operations which change the environment or check its value. All these operation have the usual meaning:

Open a symbol (label) table and make it current;

Close a symbol (label) table (its parent becomes the current one);

Search a (nested) symbol table for a given item;

Create an entry with a given value and put it in the current table;

Change the value of an entry in a table.

A.6 List of Rules

The following shorthand notations are used in the sequel:

$\underline{\varepsilon}$ denotes the empty string;

<uncond. statement> denotes <unconditional statement>;

<arith. exp> and <E> denotes <arithmetic expressions>;

<s. arith. exp> and <SE> denotes <simple arithmetic expression>;

<Boolean exp> and <B> denotes <Boolean expression>;

<s. Boolean exp> and <SB> denotes <simple Boolean expression>

Group B: Blocks

B.1 50 $\left\{\begin{matrix} B^1 \\ B^0 \end{matrix}\right\} \left\{\begin{matrix} \text{<statement>} \\ \underline{\varepsilon} \end{matrix}\right\} ; \to B^0$

B.2 50 $\left\{\begin{matrix} B^1 \\ B^0 \end{matrix}\right\} \left\{\begin{matrix} \text{<statement>} \\ \underline{\varepsilon} \end{matrix}\right\} \underline{\text{end}}' \to \text{<uncond. statement>}$

B.3 50 $\left\{\begin{matrix} B^1 \\ B^0 \end{matrix}\right\} \left\{\begin{matrix} \text{<statement>} \\ \underline{\varepsilon} \end{matrix}\right\} \underline{\text{end finish}} \to \text{<program>}$

|| closed current label table. If all labels in label table have been defined then true else false.

(Compare with rules D.10 to D.13 and note that a label can be defined in an outside block after the inside block has been processed). Terminate.

Group L: Labels

L.1 50 go to <label> → <uncond. statement>

L.2 50 <label> : <for statement> → <for statement>

L.3 50 <label> : <uncond. statement> → <uncond. statement>

L.4 50 <label> : <statement> → <statement>

Group S: Statements

S.1 50 <cond. statement> → <statement>

S.2 50 α<uncond. statement> → α<statement> | if (α ≠ 'then') and (α ≠ ':') then true else false;

S.3 50 α<for statement> → <statement> | if α ≠ 'then' and α ≠ ':' then true else false;

S.4 50 if <SB> then <uncond. statement>α → <cond. statement>α | if (α ≠ 'else') and (α ≠ '; else') then true else false;

S.5 50 if <SB> then <for statement> → <cond. statement>

S.6 50 if <SB> then <uncond. statement> $\left\{ \begin{matrix} \text{else} \\ ;\ \text{else} \end{matrix} \right\}$ <statement> → <cond. statement>

S.7 50 FOR' <F> do <statement> → <for statement>

Group F: For Clause

Denote the following constructs by < F > :

$$\langle F \rangle \triangleq \left\{ \begin{array}{l} \langle\text{s. arith exp.}\rangle \\ \langle\text{s. arith exp.}\rangle\ \underline{\text{step}}\ \langle\text{s. arith exp.}\rangle\ \underline{\text{until}}\ \langle\text{s. arith exp.}\rangle \\ \langle\text{s. arith exp.}\rangle\ \text{while}\ \langle\text{SB}\rangle \\ \langle\text{s. arith exp.}\rangle\ \text{while}\ \langle\text{B}\rangle \end{array} \right\}$$

F.1 50 FOR' <F>, → FOR'

F.2 50 <u>for</u> <arith. variable> := → FOR'

Group AS: Assignment Statement

Denote the following constructs by <A> :

$$\langle A \rangle \triangleq \left\{ \begin{array}{l} \langle\text{arith. variable}\rangle \\ \langle\text{subscripted arith. variable}\rangle \\ \langle\text{Boolean variable}\rangle \\ \langle\text{subscripted Boolean variable}\rangle \end{array} \right\}$$

AS.1 50 <left part> <A>:= → <left part>

$$\text{AS.2}\quad 50\quad \langle\text{left part}\rangle \left\{ \begin{array}{l} \langle\text{B}\rangle \\ \langle\text{SB}\rangle \\ \langle\text{arith. exp.}\rangle \\ \langle\text{s. arith. exp.}\rangle \end{array} \right\}\beta \rightarrow \langle\text{uncond. statement}\rangle\beta \;\Big|\; \underline{\text{if}}\ \alpha = \text{';'}\ \text{then}\ \underline{\text{true}}\ \text{else}\ \underline{\text{false}};$$

AS.3 50 α<A> := → α<left part> | <u>if</u> $\alpha \neq$ '<u>for</u>' <u>then</u> <u>true</u> <u>else</u> <u>false</u>;

Group E and BR: Arithmetic, Boolean and Relational Expressions

The following shorthand notation are used by these groups:

<SE> $\triangleq$ <simple arith. exp.>

<E> $\triangleq$ <arith. exp.>

<SB> $\triangleq$ <simple Boolean exp.>

<B> $\triangleq$ <Boolean exp.>

E.1 40 $\left(\begin{Bmatrix}\text{<SE>}\\ \text{<E>}\end{Bmatrix}\right) \rightarrow \text{<SE>}$

E.2 40 $\text{<SE>}\uparrow\text{<SE>} \rightarrow \text{<SE>}$

E.3 36 $\text{<SE>}\begin{Bmatrix}*\\ /\end{Bmatrix}\text{<SE>}\beta \rightarrow \text{<SE>}\beta$ | if $\beta \neq$ '↑' then true else false;

E.4 34 $\alpha\begin{Bmatrix}+\\ -\end{Bmatrix}\text{<SE>}\beta \rightarrow \alpha\text{<SE>}\beta$ | if $\alpha \neq$ $'\begin{Bmatrix}(\\ :=\end{Bmatrix}'$

and $\beta \neq$ $'\begin{Bmatrix}\uparrow\\ *\\ /\end{Bmatrix}'$ then true else false;

E.5 32 $\text{<SE>}\begin{Bmatrix}+\\ -\end{Bmatrix}\text{<SE>}\beta \rightarrow \text{<SE>}\beta$ | if $\beta \neq$ $'\begin{Bmatrix}\uparrow\\ *\\ /\end{Bmatrix}'$ then true else false;

E.6 30 if <SB> then <SE> else <E> → <E>

E.7 28 <arith. variable>α → <SE>α | if (α ≠ ':=') and (α ≠ '[')

then true else false;

E.8 28 <subscript arith. variable>α → <SE>α | if α ≠ ':=' then true

else false;

E.9 28 <arith. variable>[→ ρ | if <arith. variable> has been declared

as an array then true else false;

E.10 28 $\rho\begin{Bmatrix}\text{<SE>}\\ \text{<E>}\end{Bmatrix}, \rightarrow \rho$

E.11 28 $\rho\begin{Bmatrix}\text{<SE>}\\ \text{<E>}\end{Bmatrix}] \rightarrow$ <subscripted arith. variable>

E.12 28 α<number>β → α<simple arith. exp.>β | if (current symbol table

is close) and α = 'go to' or β = ':' then false

else true;

BR.1 24 $\langle SE\rangle \left\{\begin{matrix} < \\ \le \\ = \\ > \\ \ge \\ \ne \end{matrix}\right\} \langle SE\rangle\beta \to \langle SB\rangle\beta \quad \mid \underline{\text{if}}\ \beta \ne \text{'}\left\{\begin{matrix} \uparrow \\ * \\ / \\ \underline{+} \end{matrix}\right\}\text{'}\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.2 22 $\left(\left\{\begin{matrix} \langle B\rangle \\ \langle SB\rangle \end{matrix}\right\}\right) \to \langle SB\rangle$

BR.3 20 $\underline{\text{not}}\ \langle SB\rangle \to \langle SB\rangle$

BR.4 18 $\langle SB\rangle\ \underline{\text{and}}\ \langle SB\rangle \to \langle SB\rangle$

BR.5 16 $\langle SB\rangle\ \underline{\text{or}}\ \langle SB\rangle\beta \to \langle SB\rangle\beta \quad \mid \underline{\text{if}}\ \beta \ne \text{'}\underline{\text{and}}\text{'}\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.6 14 $\langle SB\rangle \supset \langle SB\rangle\beta \to \langle SB\rangle\beta \quad \mid \underline{\text{if}}\ \beta \ne \left\{\begin{matrix} \underline{\text{and}} \\ \underline{\text{or}} \end{matrix}\right\}\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.7 12 $\langle SB\rangle \equiv \langle SB\rangle\beta \to \langle SB\rangle\beta \quad \mid \underline{\text{if}}\ \beta \ne \left\{\begin{matrix} \underline{\text{and}} \\ \underline{\text{or}} \\ \supset \end{matrix}\right\}\ \underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.8 10 $\underline{\text{if}}\ \langle SB\rangle\ \underline{\text{then}}\ \langle SB\rangle\ \underline{\text{else}}\ \langle B\rangle \to \langle B\rangle$

BR.9 8 $\langle\text{Boolean variable}\rangle\alpha \to \langle SB\rangle\alpha \quad \mid \underline{\text{if}}\ \alpha \ne \text{':='}\ \underline{\text{and}}\ \alpha \ne \text{'['}$
$\underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.10 8 $\langle\text{subscripted Boolean variable}\rangle\alpha \to \langle SB\rangle\alpha \quad \mid \underline{\text{if}}\ \alpha \ne \text{':='}\ \underline{\text{then}}$
$\underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.11 8 $\langle\text{Boolean variables}\rangle[\ \to \xi \quad \mid \underline{\text{if}}\ \langle\text{Boolean variable}\rangle$ is declared
as an array $\underline{\text{then}}\ \underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}};$

BR.12 8 $\xi \left\{\begin{matrix} \langle SE\rangle \\ \langle E\rangle \end{matrix}\right\}, \to \xi$

BR.13 8 $\xi \left\{\begin{matrix} \langle SE\rangle \\ \langle E\rangle \end{matrix}\right\}] \to \langle\text{subscripted Boolean variable}\rangle$

BR.14 8 $\left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\}$ → <SB>

Group D: Declarations

The following shorthand notations are used:

$\langle\text{type}\rangle \triangleq \left\{\begin{matrix}\underline{\text{integer}}\\ \underline{\text{Boolean}}\end{matrix}\right\}$

D.1	4	begin → B'		Open a symbol table and a label table and make them current, (the symbol table is open but is not attached to main table);
D.2	4	B'<type>α → B''α	if α ≠ 'array' then true else false;	Set T to the value of <type> (Next list of identifiers have type <type>);
D.3	4	B''<identifier>, → B''		Put <identifier> in symbol table (check if it does not appear already in current table, etc.);
D.4	4	B'<identifier>; → B'		Put <identifier> in symbol table;
D.5	4	B'<type> array → B'''		Set T to the value array <type>;
D.6	4	B'''<identifier>, → B'''		Put <identifier> in symbol table;
D.7	4	B'''<identifier> [→ B^4		Put <identifier> into symbol table. Prepare for upper and lower bounds.
D.8	4	$B^4\left\{\begin{matrix}\langle E\rangle\\ \langle SE\rangle\end{matrix}\right\}:\left\{\begin{matrix}\langle E\rangle\\ \langle SE\rangle\end{matrix}\right\}\beta \to B^4\beta$	if $\beta = '\left\{\begin{matrix}]\\ ,\end{matrix}\right\}'$ then true else false;	Put upper and lower bounds into table.
D.9	4	$B^4, \to B^4$		Increase dimension of array by one.
D.10	4	$B^4, \to B'''$		
D.11	4	$B^4; \to B'$		

D.12	4	B'$\alpha \to$ B^0	if $\alpha \neq$ <type> then true else false;	close current symbol table, connect current symbol table to main symbol table;
D.13	4	<identifier>$\alpha \to$ <arith. variable>α	((<identifier> has been declared and its <type> is <integer>) and (symbol table is closed) and ($\alpha \neq$ ':') then true else false);	
D.14	4	<identifier>$\alpha \to$ <Boolean variable>α	if (<identifier> has been declared and its <type> is <Boolean>) and (symbol table is closed) and ($\alpha \neq$ ':') then true else false;	
D.15	4	$\alpha\left\{\begin{matrix}\text{<identifier>}\\ \text{<number>}\end{matrix}\right\}\beta \to \alpha$<label>$\beta$	if (current symbol table is closed) and ((β = ':') or (α = 'go to')) and (if $\beta \neq$ ':' then true else (if $\left\{\begin{matrix}\text{<number>}\\ \text{<identifier>}\end{matrix}\right\}$ does not appear in current label table) then true else false)) then true else false;	if (α = 'go to') and ($\left\{\begin{matrix}\text{<number>}\\ \text{<identifier>}\end{matrix}\right\}$ has not been declared in current label table) then enter it in label table as an undefined label. if (β = ':') then begin if

$\left\{\begin{matrix}\text{<number>}\\ \text{<identifier>}\end{matrix}\right\}$ is in current label table make it into a defined <label> else enter it in symbol table as a defined label. end;

D.16 4 end → end' | close current label table; make current the parent of the current label table and the parent of current symbol table;

LIST OF CAPTIONS

Figure 1: An Example of a Syntax Graph

a) Short-hand notation

b) Full notation

The Grammar: input string: ABBBCBA

Rule number	Priority	patterns
1	5	ABC $\rightarrow$ X
2	4	BBB $\rightarrow$ BB
3	3	AB $\rightarrow$ A
4	2	XB $\rightarrow$ A
5	1	AA $\rightarrow$ S

Figure 2: An Improper use of the Handle

a) The Syntax Graph generated by the MMA

b) The Syntax Graph generated by the Handle after Handle method.

The Grammar:

Rule number	Priority	Patterns
1	5	ABC $\rightarrow$ X
2	4	CB $\rightarrow$ B
3	3	BB $\rightarrow$ B
4	2	AB $\rightarrow$ A
5	1	AA $\rightarrow$ S

Figure 3: The Modified Tree

a) R_i is applied before R_j

b) R_j is applied before R_i

c) The corresponding modified syntax tree

$R_i \quad \underline{\pi_i}\ \underline{\beta} \rightarrow \underline{\rho_i}\ \underline{\beta}$ | $\underline{\text{if}}\ \underline{\beta} = A\ \underline{\text{or}}\ C$ then $\underline{\text{true}}$

$R_j \quad \underline{\beta}\ \underline{\pi_j} \rightarrow \underline{\beta}\ \underline{\rho_j}$ | $\underline{\text{if}}\ \underline{\beta} = A\ \underline{\text{or}}\ C$ then $\underline{\text{true}}$

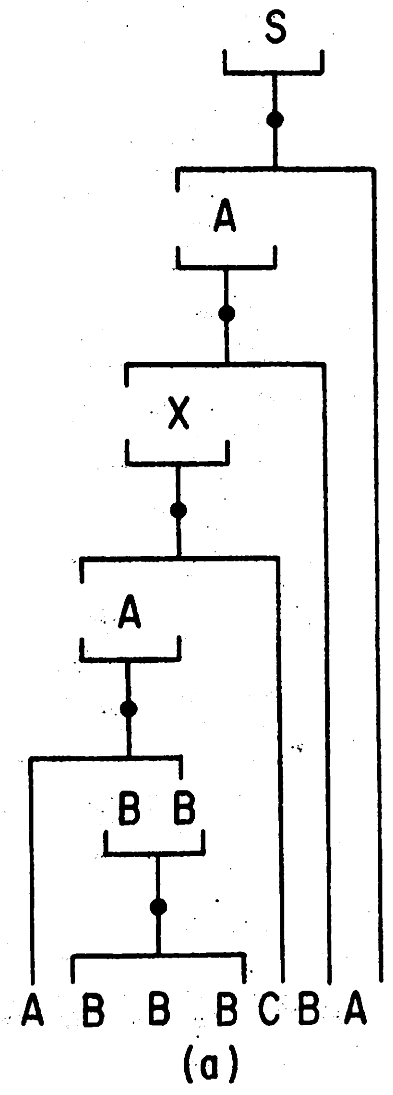
Figure 4: The Feedback Composition of MA_1 and MA_2

Figure 5: Markers

Figure 6: An example of the parsing of the declaration of an array: Reductions are done in the order indicated by the numbers to the left of the nodes. To the right of each node appears the name of the rule used in the reduction. <a.v.> is an abbreviation for <arithmetic variable>.

Figure 7: The Algol Parsers as a Composition of MMA's.

Figure 8: Possible (b) and Impossible Positions of Matches in Parsing Expressions by Grammar C.



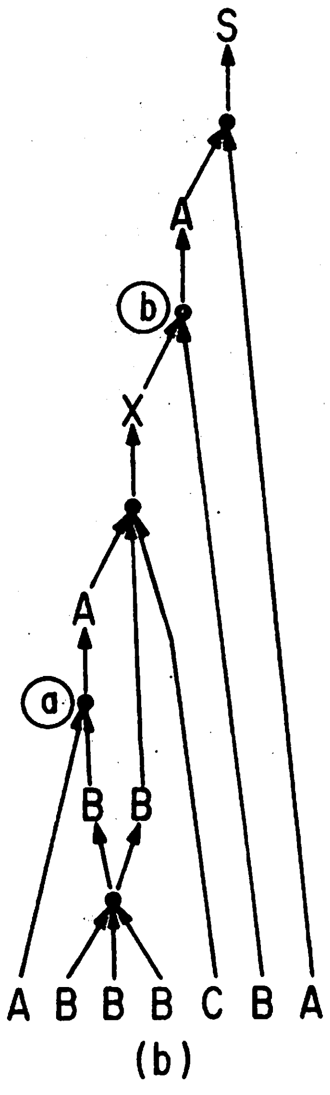


Figure 1.

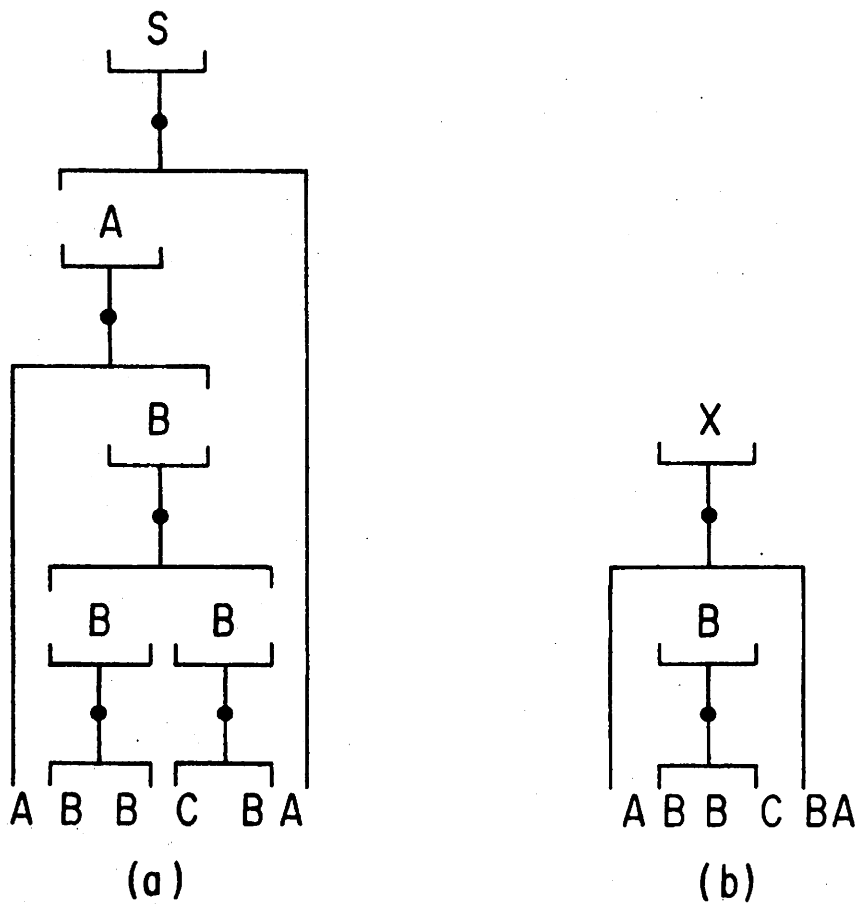


Figure 2.

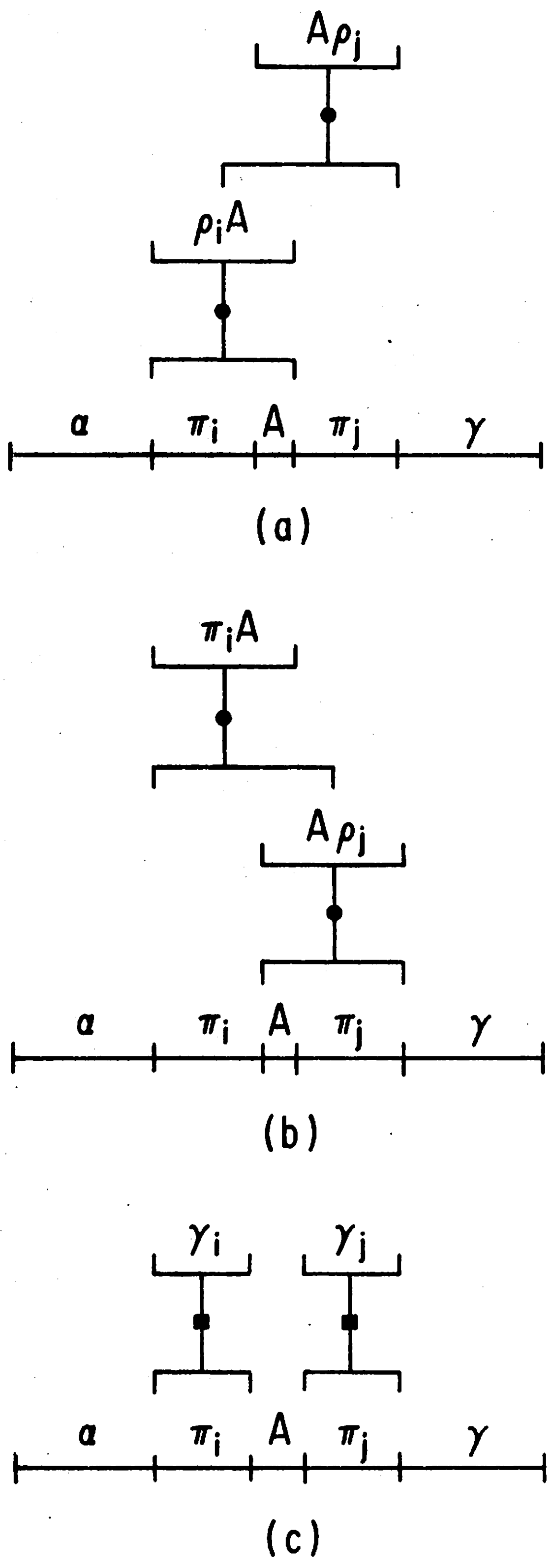

$A\rho_j$
$\rho_i A$
α
π_i
A
π_j
γ
(a)
$\pi_i A$
$A\rho_j$
α
π_i
A
π_j
γ
(b)
γ_i
γ_j
α
π_i
A
π_j
γ
(c)

Figure 3.

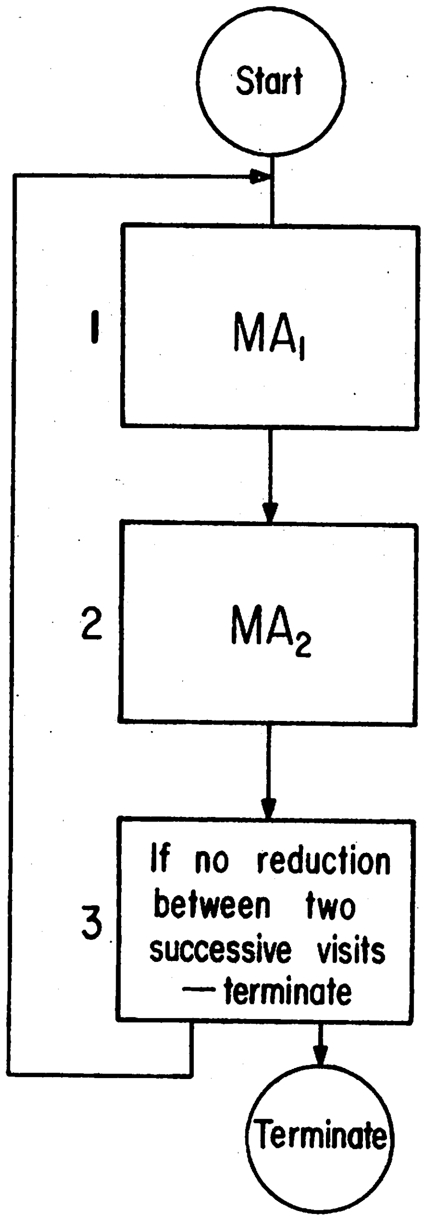
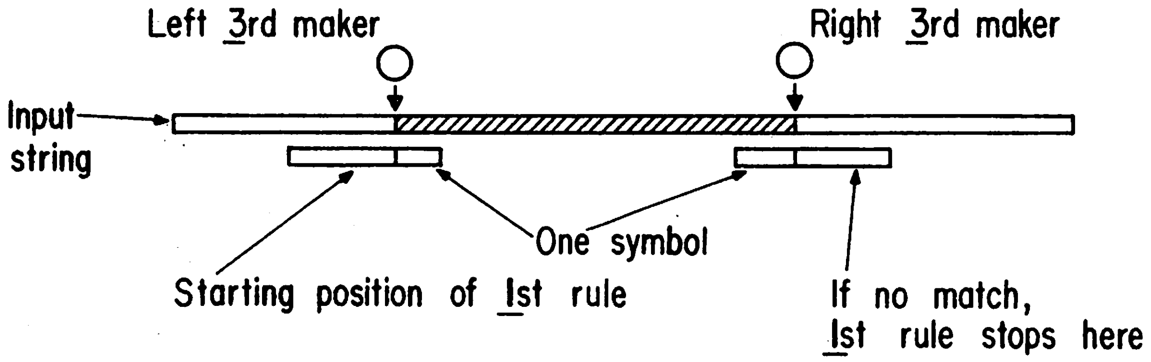

Start
1
MA_1
2
MA_2
3
If no reduction between two successive visits — terminate
Terminate

Figure 4.



Left 3rd maker
Right 3rd maker
Input string
One symbol
Starting position of 1st rule
If no match, 1st rule stops here

Figure 5.

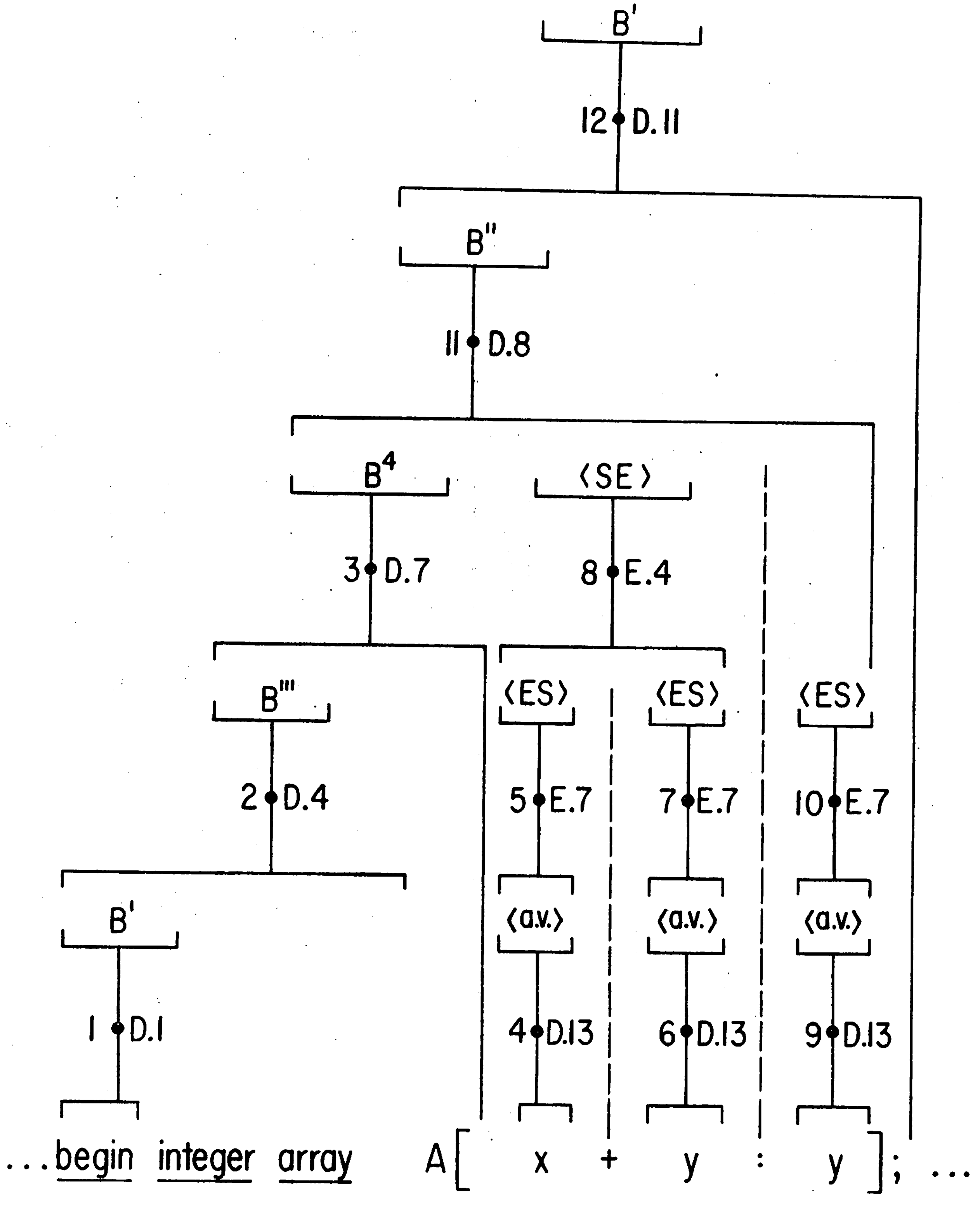

B'
12 D.11
B''
11 D.8
B4
⟨SE⟩
3 D.7
8 E.4
B'''
⟨ES⟩
⟨ES⟩
⟨ES⟩
2 D.4
5 E.7
7 E.7
10 E.7
B'
⟨a.v.⟩
⟨a.v.⟩
⟨a.v.⟩
1 D.1
4 D.13
6 D.13
9 D.13
...begin integer array A[x + y : y]; ...

Figure 6.

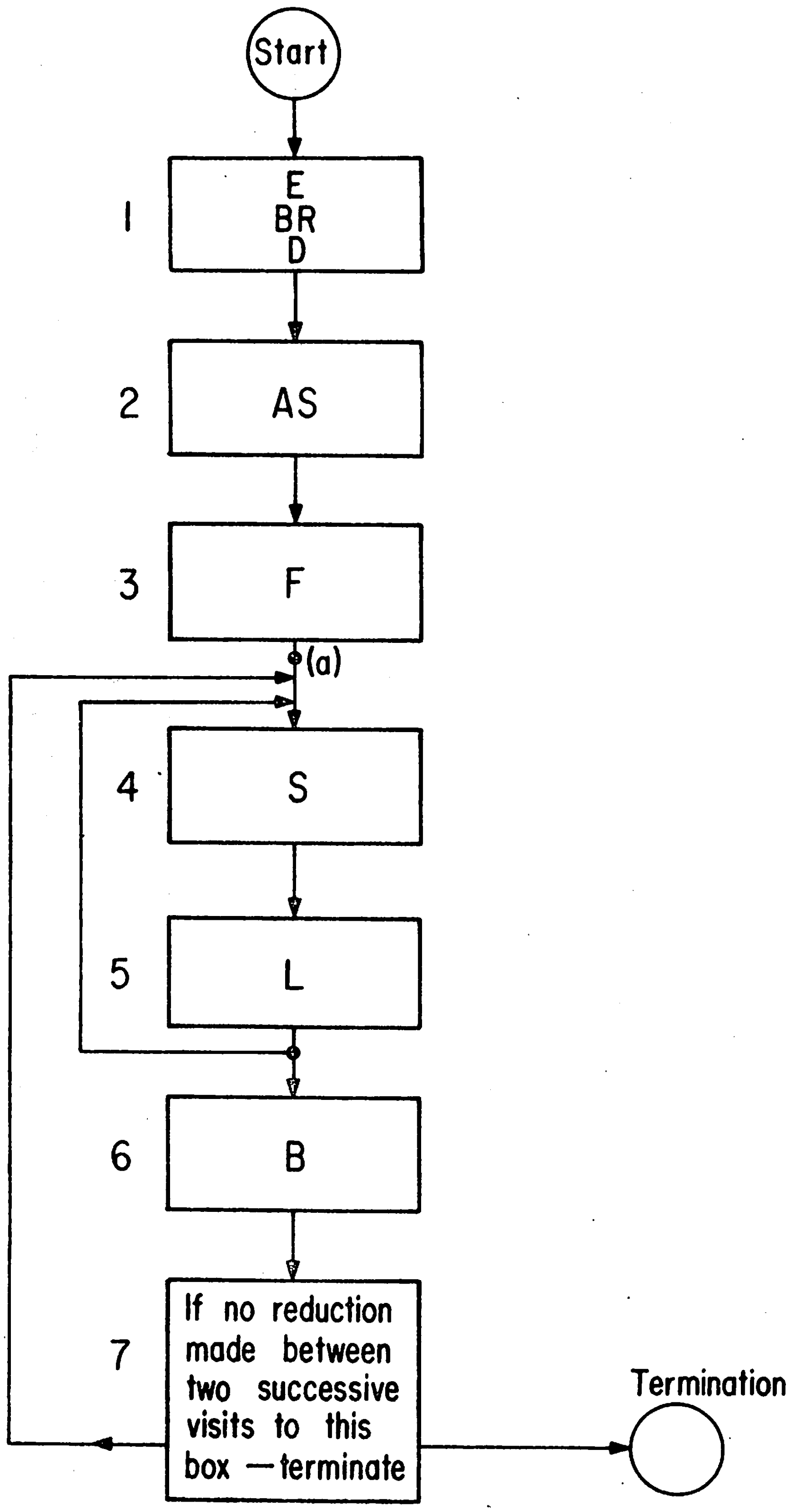

Start
1
E
BR
D
2
AS
3
F
(a)
4
S
5
L
6
B
7
If no reduction made between two successive visits to this box — terminate
Termination

Figure 7.

```
                LM       RM
   . . . . . .  ↓  <SE>  ↓  . . . . .
                         ¦
                   <SE>  ¦*  <SE>
                         ¦
                   <SE>  ¦↑  <SE>
                         ¦
                         ¦
                         ¦
```

(a)

```
                    LM    RM
       . . . . . .↓<SE>↓. . . . .

              <SE>*<SE>

              <SE>↑<SE>
```

(b)

Figure 8.