S O M E A S P E C T S O F E R R O R C O R R E C T I O N O F P R O G R A M M I N G L A N G U A G E S

by

Militon FRENTIU

We can only see a
short distance ahead,
but we can see plenty
there that needs to
be done.

Turing

A thesis submitted for the degree
of Doctor of Philosophy at the
BRUNEL UNIVERSITY
1976

ACKNOWLEDGEMENTS

I am very indebted to Professor Dr. Igor Aleksander for his guidance, encouragement and many helpful discussions during the period of this project. These facts were very important in completing this thesis.

Also, I would like to thank Professor Douglas Lewin for his help and for many suggestions that guided me in this important area of research.

Thanks are also due to all my colleagues, in special Clylton Fernandes, for their help and "feedback" in learning English.

I wish to thank the Ministry of Learning and Education of Romania for financial support.

Also, I wish to thank Mrs. J. Marshall who typed this thesis.

Finally I would like to thank my wife for her understanding, patience and encouragement during the period of this research.

Militon Frentiu
Brunel University
November 1976

ABSTRACT

This thesis treats the problem of error correction in a context free language, and the design of an error correcting parser for the BASIC language.

Two important things can be said about this thesis.

First, it presents the problem of error correction in a context free language, and the existing results in the field. The concept of a context free language as a model for a programming language, and the definitions and results used later are presented or reviewed. A distance between two strings is defined and used to develop a "minimum distance error correcting parser".

Second, the thesis develops two global error correcting parsers.

The first one is the top-down global error correcting parser, obtained by transforming Unger's top-down parser into an error correcting one.

Then the idea of Graham and Rhodes, of condensing the surrounding context of the error, is extended, and a global simple precedence error correcting parser is obtained by analysing the whole context of the error, available from the input string.

These parsers, and other known methods are then used to design and partially implement an error correcting parser for BASIC.

CONTENTS

	page
Acknowledgements	1
Abstract	2
Notations	5
1. Introduction	7
1.1. Statement of the problem	7
1.2. The main parts of this thesis	10
2. Basic definitions	13
2.1. Context-free languages	13
2.2. The parsing problem	16
2.3. An efficient parsing algorithm for context-free languages	18
2.4. A global top-down parser	28
2.5. Parsing a simple precedence grammar	32
2.6. Acceptors	34
3. The problem of error correction and a survey of previous work	38
3.1. The distance between two strings	38
3.2. The problem of error correction in a programming language	42
3.3. A survey of the previous work in error recovery and error correction techniques	45
4. Some error correcting parsers	51
4.1. Error correction by a transducer	52
4.2. Error correction in a regular language	57
4.3. A simple precedence error correcting parser	59

4.4. A global top-down error correcting parser 91

4.5. Summary 102

5. The design of an error correcting parser for BASIC-like languages 103

5.1. The general and particular features of BASIC 104

5.2. Structural design of an error-correcting parser for BASIC 107

6. Conclusion and future work 118

6.1. What are the results of this thesis ? 118

6.2. What remains to be done? 121

6.3. Finally , what is the real nature of the problem 123

7. Bibliography 125

8. Appendices

8.1. Appendix 1 - The syntax of BASIC 130

8.2. Appendix 2 - Program for building a simple precedence matrix 135

8.3. Appendix 3 - The top-down error correcting parser 145

8.4. Appendix 4 - The simple precedence error correcting parser 153

NOTATIONS

$A = \{a_1, a_2, \ldots, a_n\}$	the set A has elements a_1, a_2, ... ,a_n.
$A = \{a \mid p(a)\}$	A is the set of all elements a which have the property p, i.e. p(a) is 'True'.
$a \in A$	a belongs to A
$A \subset B$	all elements of A belong to B
$A \cap B$	the set of all common elements of the sets A and B
$A \cup B$	the union of the sets A and B
$(\exists a)p(a)$	there exists a such that p(a) is 'True'.
$A \times B$	the set of all pairs (a,b), where $a \in A$, and $b \in B$
ϕ	the empty set
$f: A \rightarrow B$	f is a map from A to B
$\Rightarrow$	implies. Often this symbol is used in this thesis to denote a relation between strings, defined in section 2.1.
L	usually denotes a language
$G = (N,T,P,S)$	notation defined in section 2.1. G stands for grammar, N stands for nonterminal alphabet, T stands for terminal alphabet, P is the set of all productions of the grammar G, and S denotes the sentence metasymbol.
V	V denotes the set of all symbols, i.e. V = NUT
$\pi : A \rightarrow \alpha$	the production π is $A \rightarrow \alpha$
π_i	is the i-th production of P
LHS	left hand side (of the production π is A)
RHS	right hand side (of the production π is α)
T^+	denotes the set of all strings over the set T, but nonempty.
ε	is the empty string (or the null element for concatenation)

T^*	is the set of all strings over the alphabet T, i.e. $T^* = T^+ \cup \{\epsilon\}$
a,b,c,...	usually denote characters from T
x,y,z,..	usually denote characters from V
u,v,w	usually denote strings of T^*
$r, \alpha, \beta, \gamma, \eta$.	denote strings of V^*
::=	has the same meaning as $\longrightarrow$ (symbol used by Bachus et al.)
$\xrightarrow{+}$	is the transitive closure of $\longrightarrow$, defined in section 2.1.
$\min_{p(a)} f(a)$	the minimum of f(a), for those values of a that have property p.
2^T =P(T)	is the set of all subsets of the set T
$M=(K,T,\Gamma,\delta,s_0,Z_0,F)$	denotes a pushdown automaton, defined on page 35
$M_t=(K,T,\Gamma,\Delta,\delta_t,s_0,Z_0,F)$	denotes a transducer associated with the pushdown automaton M, defined on page 52
Γ	is the stack alphabet of a pushdown automaton
Δ	is the output alphabet of a transducer

I INTRODUCTION

1.1. STATEMENT OF THE PROBLEM

With the increasing use of computers the writing of computer programs is becoming a time consuming and therefore expensive activity. Very often a program is compiled five times before the first correct version of it is obtained. This is particularly true in universities, where many students are faced with their first program. It has been observed [D1, M2, L9] that a large percentage of errors encountered are very simple ones, like one insertion, one deletion, or one substitution. Thus it becomes possible to design schemes which correct errors or provide good diagnostics.

Often the detected errors can be "corrected" in many ways, since one cannot decide exactly what the user intended to say. In this case it is important to provide good diagnostics, or sometimes to indicate all possible ways of correcting the error. The compiler must show which substring(p.13)(ideally symbol) is in error, why there is an error, and how this error can be corrected.

The process of using a computer for solving problems consists, first of all, in finding an agreed form of communication between the computer and the user. The problem must be 'sent' and 'understood' by computer. The need for coding information and its transmission to the computer is obvious and has arisen as a result of the existence of computers themselves. The appearance of high-level programming languages has solved the problem of coding for communication in a two-way man-machine channel.

Being developed for communication in a two-way channel, programming languages were not designed for error correction with a given error-correcting capability, like error-correcting block codes [P2, F5] for data transmission. Designers of languages did not consider the occurrence

*) Since some concepts, which are defined lately are used in this chapter, the page where this is done is indicated in the paranthesis.

of errors in their design . However they allow error detection, in the sense that a message would be sent back to the programmer when this detection takes place. This two-way communication channel is more evident in an interactive system.

Receiving a list of errors the user will correct them and send the program back to the computer. Therefore it is important that as many errors as possible be detected in one pass. But often, after detecting an error the compiler fails to detect some other errors, or detects 'imaginary' errors. Take for example a FORTRAN program in which the first DIMENSION statement is erroneous. This causes many compilers to 'think' that all statements containing arrays are incorrect. It seems self-evident that a good compiler should always correct an erroneous DIMENSION statement.

We say that a program is a correct version of an input string (p.18) if this program is, in some way, the most similar to the input string. It is difficult to say what should be understood by "the most similar". Also very often the solution is not unique and it is impossible to choose any one alternative by any criteria other than arbitrary ones.

This impossibility of general error correction and the need for the detection of all or almost all errors leads us to the concept of error recovery. By error recovery one understands the removal of the effect of a detected error and a decision on how to continue to analyse the input string (p.43).

Although a programming language was designed without error-correction in mind, it still contains enough redundancy to permit the detection of almost all probable errors. At the same time the correction of many of them is possible.

All codes are constructed from basic symbols which form their alphabets. For programming languages the alphabet consists of letters, digits, and other special characters. However we can look at a programming language in a different way, that is, as a stratified language.

Some words are built from the basic alphabet symbols. Statements are then formed with these lexical words. The program is an ordered sequence of statements.

At any one point in error correction one can only be concerned with a specific level of the programming language. At the level of a compiler the language is the set of all correct programs. At the level of interpreter the language is the set of all correct statements. At the level of a scanner the language is the set of all lexical words!

There are four types of errors which we can find in a program:

- lexical errors (ex. GOTU instead of GOTO)
- syntax errors (ex. V2(5.5), not V2(5,5))
- semantic errors (ex. DIM V2(0)) for a positive integer)
- logical errors(ex. NEXT-statement on a wrong line).

A programmer who does not know the intent of the user can correct the first three types of error. However many semantic errors can be corrected through his experience rather than using the syntax of the language itself. The information needed to correct lexical and syntax errors is contained almost entirely in the syntax of the statements, and not of the program itself. The errors which can be corrected from the syntax of the program are only a few, and some of them very rarely happen.

Some methods to correct errors will be described or developed in this thesis. Then the design of an error correcting parser will be presented, and an implementation to correct errors at the level of interpreter will be described.

1.2. THE MAIN PARTS OF THE THESIS

This thesis treats the problem of error correction in a programming language. Different error-correcting parsers are discussed and used to design an error correcting parser for BASIC.

Existing work in this area will be discussed and all notions used in this thesis will be briefly defined. As seen this first chapter states the problem of error correction, and its importance to programming.

The second chapter discusses the background field of this thesis. The notion of a context-free language, introduced by Chomsky (1956) [C2], and Backus et al (for the purpose of studying a programming language) (1960) [N1, N2], is defined here. The problem of sentence recognition and parsing are also presented. Then some important parsers are described. Only a few among many existing parsing schemes for different subclasses, or for the entire class of context-free languages have been included. The choice was restricted to all those parsers which are used or referred to in the latter part of the thesis. Chapter Two ends with the definition of finite state machines and pushdown automata, as recognizers for regular languages, or context-free languages, respectively. Also the notion of a transducer is defined.

The third chapter introduces the problem of error correction in a more formal way. First one defines (section 3.1) the notion of error, the edit operation as an action to correct the error, and the distance between two strings. Then, based on this distance, the correction of an arbitrary string is defined. Although normally the correction of an arbitrary string w is defined, it is underlined in many places in this thesis, that w in practice is not an arbitrary string, but an erroneous version of an initially correct sentence. After all we are studying the problem of communication in the man-machine channel, and we are

interested in decoding algorithms for meaningful information.

Section 3.3. is a survey of the literature in the field of errpr correction for different classes of languages.

The main results of this thesis are presented in the fourth chapter. They open the possibility for the design of a global top-down error correcting parser. This design will be done for the BASIC language in the firth chapter.

The first section of chapter four presents,briefly, error correction by a transducer. This is a limited correction,of only anticipated errors.

Section 4.2. presents error correction in a regular language(p.34). Although a programming language is not usually regular, many parts of it can be described by regular grammar.

The next two sections contain the most important results of the thesis. A global simple precedence parser is given in section 4.3, and a global top-down error correcting parser for the entire class of context free languages is presented in section 4.4. The error correcting parsers are called global since they analyse the context of the whole input string in order to decide how to correct the error. The global top-down error correcting parser is the result of transforming the global top-down parser introduced by Unger [U2], into an error correcting parser.

For both error correcting parsers mentioned above the process of error correction does not hamper the analysis of correct sentences. Both parsers can be viewed as having two modes (as defined by Levy [L], or in [G2,L6]): a "standard mode" used for the syntax analysis of correct sentences, and an "error correcting mode". The standard mode proceeds at the same speed as the original parser for analysing any input string until the detection of at least one error.

The global error correcting parser for a simple precedence language extends the idea of condensing the surrounding context of the error

to acquire more information needed for correction (see Graham and Rhodes [G2]). Here the context of the whole string is analysed and condensed by a "scanning phase" that is equivalent to the "standard mode" of the parsing process mentioned above. It scans the input string character by character, detects simple phrases, and reduces them to the corresponding metasymbols(p13)It also detects and reports the errors, marking the detection points of the errors on the stack(p32 .If no errors are detected, this phase terminates with a complete analysing of the corresponding sentence.

If errors are detected, the error correction mode is entered after the whole input string has been scanned, condensed, and the reduced version of the imput string has been stored in the stack. It analyses the contents of the stack, which now contains all the information available from the input string, and makes a decision about how to correct the errors.

While the global top-down error correcting parser is valid for any programming language, the global simple precedence error correcting parser is not, since not all programming languages are simple precedence languages.

Sometimes a programming language can be transformed into a simple precedence language by a suitable change of its lexicon and its grammar. If this is possible, clearly the global simple precedence error-correcting parser can be used. Otherwise the global top-down error correcting parser, which is not very efficient, must be used. Nevertheless its efficiency can be improved by combining it with other, more efficient parsers. This is investigated in the fifth chapter, where an error correcting parser for BASIC is designed and discussed.

Finally the last chapter looks back at the problems treated in this thesis, making some suggestions for future work. Also the entire problem is briefly looked at from a totally different but broader point of view.

CHAPTER II

BASIC DEFINITIONS

2.1. CONTEXT FREE LANGUAGES

In order to study programming languages, a theoretical model for these languages is necessary. It is not intended to present here the entire theory of formal languages. Since the class of context-free languages introduced by Chomsky [C2] is a convenient model, generally accepted, the discussion will be restricted to this class.

An alphabet T is a finite nonempty set of elements which we call either symbols or terminals. Any finite sequence

$$a_1 a_2 \cdots a_n$$

of symbols is called a string. n is the length of the string. We denote by ε the empty string, i.e. the string of length zero. The set of all strings of nonzero length is denoted by T^+ and $T^* = T^+ \cup \{\varepsilon\}$ is the set of all strings over T. A language with the alphabet T is a subset of T^*. We need a finite procedure to specify this subset.

A context-free grammar G is a quadruple (N , T , P , S) where

- N is a finite set of metasymbols called nonterminals or variables
- T is the alphabet of the language. $V = N \cup T$ is called the vocabulary.
- S is a "distinguished" metasymbol, called sentence metasymbol [G4].
- P is a finite set of productions. A production is an ordered pair $(A , \alpha) , A \in N , \alpha \in V^*$. It means that in any string over V, the nonterminal A can be replaced by the string α. Often it is denoted by $A \longrightarrow \alpha$ and sometimes is called a rewriting rule. In bottom-up parsing we reduce the string α to the variable A, thus to any production (A , α) there corresponds a reduction rule $\alpha \longmapsto A$. We say that A is the left hand side (LHS), and α is the right hand side (RHS) of this production.

Once we have defined the grammar G, we must show how it can be used to generate the language. First we show how the productions can be used

to derive some new strings from existing ones.

For a given grammar G we define a relation $\underset{G}{\Longrightarrow}$ over V^*. We say that the string $\alpha \in V^+$ directly produces the string $\beta \in V^*$, and write $\alpha \underset{G}{\Longrightarrow} \beta$, if there are some strings $r_1, r_2, \eta \in V^*$, and a nonterminal A such that

$$\alpha = r_1 A r_2$$
$$\beta = r_1 \eta r_2 , \qquad (2.1.1)$$

and (A , η) is a production. We also say that β directly reduces to α, and write $\beta \vdash \alpha$. Note that for any production $A \longrightarrow \eta$ we have $A \underset{G}{\Longrightarrow} \eta$. If the grammar G is properly agreed we write $\Longrightarrow$ instead of $\underset{G}{\Longrightarrow}$.

We say that α produces β, and write $\alpha \overset{+}{\Longrightarrow} \beta$, or β reduces to α, and write $\beta \overset{+}{\vdash} \alpha$, if there exists an integer n, $n \geq 1$ and a sequence of direct derivations

$$\alpha = \alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow \alpha_n = \beta .$$

We say that $\alpha \overset{*}{\Longrightarrow} \beta$ if $\alpha \overset{+}{\Longrightarrow} \beta$, or if $\alpha = \beta$.

We are able now to define the language generated by the grammar G. A string produced by the distinguished metasymbol S is called a sentential form. A sentence is a sentential form consisting only of nonterminals. The language $L(G)$, generated by the grammar G, is the set of all sentences, i.e.

$$L(G) = \{ w \mid S \overset{*}{\Longrightarrow} w \text{ and } w \in T^* \} \qquad (2.1.2)$$

A phrase of a sentential form α is a substring w that can be generated by a variable. More formally, if $\alpha = \beta_1 \eta \beta_2$ is a sentential form, η is a phrase of α if there exists a metasymbol A such that $S \overset{*}{\Longrightarrow} \beta_1 A \beta_2$, and $A \overset{+}{\Longrightarrow} \eta$. A phrase is called a simple phrase if it is a right hand side of a production.

The purpose of having a grammar is to describe a language. All productions in the grammar are expected to be useful, i.e. to help in the derivation of at least one sentence. In addition, we would like to have a minimum number of productions. For example a production $A \longrightarrow A$

replaces the metasymbol A by itself and is useless in the sense that a new grammar, without this production, generates the same language.

It is known that for any context-free grammar there exists an equivalent context-free grammar which is ε-free [G1]. Two grammars are equivalent if they both generate the same language. A grammar is said to be ε-free if there is no production of the form $A \longrightarrow \varepsilon$. We will assume that all our grammars are ε-free.

Also we will require that all grammars are reduced. This is important when we analyse a sentence in our language. If a variable A does not generate a phrase, this variable does not appear in the derivation of any sentence, and it is futile to try to reduce a phrase to A. A context free grammar $G = (N, T, P, S)$ is said to be reduced if for any variable $A \in N$ the following conditions hold:

1) there exists a sentential form that contains A, i.e. there exists some strings $\alpha, \beta \in V^*$ and the derivation $S \overset{*}{\Longrightarrow} \alpha A \beta$.

2) the nonterminal A generates at least one terminal string, i.e. there is a string $w \in T^*$ such that $A \overset{+}{\longrightarrow} w$.

3) the derivation $A \overset{+}{\Longrightarrow} A$ is not possible.

Finally one needs to mention that for each nonterminal A of the grammar G there exists a grammar $G_A = (N, T, P, A)$ associated with it. The grammar G is reduced if all languages $L_A = L(G_A)$ are nonempty and the sentences of L_A are phrases in L.

2.2. THE PARSING PROBLEM

For a given language L (grammar G), an important problem is to discover if an arbitrary string $w \in T^*$ is a sentence of L. According to the previous definitions, w is a sentence of L if it is possible to form the sequence of direct derivations

$$S = \alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ \ldots \Longrightarrow \alpha_n = w. \qquad (2.2.1)$$

The process of obtaining this sequence of direct derivations is called parsing. We can better illustrate this sequence of derivations by a syntax tree. The root of this tree is the distinquished metasymbol S. To each derivation $\alpha_i \Longrightarrow \alpha_{i+1}$ there corresponds a production $A \longrightarrow \eta$, that has been used to replace the variable A in α_i by η (Compare with 2.1.1). Correspondingly, there exists a node in the syntax tree labelled A with the descendents labelled, from left to right $x_1, x_2, \ldots, x_l$, where

$$\eta = x_1 x_2 \ldots x_l .$$

Let us for example, consider the grammar

$$G = (\{sum, fac\} , \{a , + , * \} , P , sum)$$

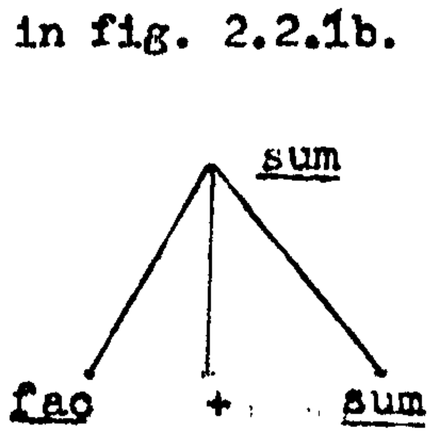
where

$$P = \{ sum \rightarrow fac , sum \rightarrow fac + sum, fac \rightarrow a , fac \rightarrow a^* fac \}.$$

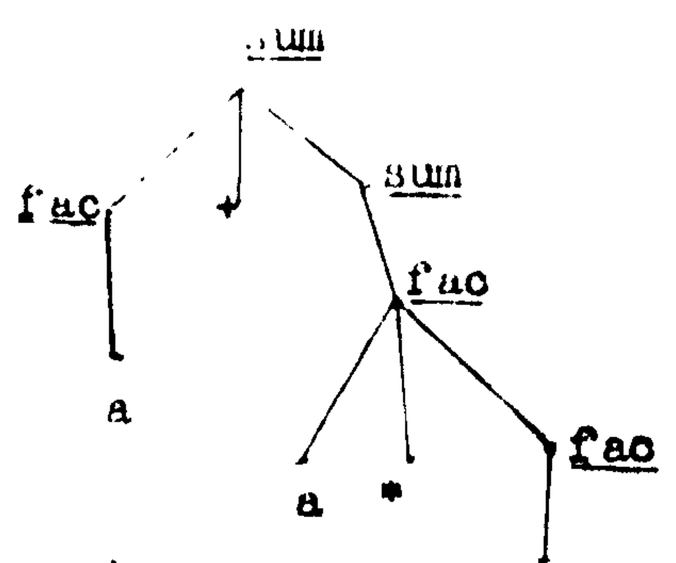
Since

$$sum \longrightarrow fac + sum \Rightarrow a + sum \Rightarrow a + fac \Rightarrow a + a^* fac \rightarrow a + a^* a , \qquad (2.2.2)$$

holds, a + a* a is a sentence and all other intermediate strings are sentential forms. The syntax tree of the sentenial form fac + sum is given in fig. 2.2.1a and the syntax tree of the sentence a + a* a is given in fig. 2.2.1b.



a)



b)

Fig.2.2.1

The sequence of derivations of the sentence a + a* a is not unique. The following sequence of derivations

$$\underline{sum} \Longrightarrow \underline{fac} + \underline{sum} \Longrightarrow \underline{fac} + \underline{fac} \Longrightarrow \underline{fac} + a^* \underline{fac} \Longrightarrow \underline{fac} + a^* a \Longrightarrow a + a^* a \qquad (2.2.3)$$

also holds.

But in our case, both derivations (2.2.2) or (2.2.3) have the same syntax tree shown in fig. 2.2.1b.

Nevertheless, there are languages which have two different syntax trees for the same sentence. Such a language is called ambiguous. In any unambiguous language there exists exactly one syntax tree for each sentence. With this we associate a canonical derivation, from many possible derivations of the sentence w . The derivation

$$S = \alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \alpha_m = w ,$$

is canonical if each direct derivation $\alpha_i \Longrightarrow \alpha_{i+1}$, is obtained by replacing the leftmost metasymbol of the sentential form α_i . That is, $\alpha_i = u A \beta$, $\alpha_{i+1} = u \eta \beta$, where u is a terminal string and $A \rightarrow \eta$ is a production.

There are different methods of parsing a sentence. If we try to build the syntax tree of the sentence w starting from the root S , going downwards towards w , we have top-down parsing. If we start with the sentence w and try to make reductions to arrive to S , i.e. going upwards in the syntax tree, we have bottom-up parsing. Almost all parsing methods are left-to-right since the symbols in the sentence w are read from left to right. Often both methods of parsing are combined to obtain a more efficient scheme.

Some parsing methods are presented in the next paragraphs since we need to use or refer to them.

2.3. AN EFFICIENT PARSING ALGORITHM FOR CONTEXT-FREE LANGUAGES

Here we intend to present briefly some algorithms for parsing some classes of context-free languages. The first, due to Earley [E1], is the most general and efficient one, for the class of all context-free languages.

We assume the productions in our grammar G are numbered from 1 to n . We add a new terminal symbol ## to mark the end of each sentence, a new distinguished metasymbol S_0 and the production number 0:

$$S_0 \longrightarrow S \#\# .$$

We often need to examine those productions with the same LHS. We say that a production with the variable A on its LHS is an A-production.

Let $w = a_1 a_2 \ldots a_n \#\#$ be the input string to be analysed. The derivation of the string w must start with $S_0 \Longrightarrow S \#\#$. Then an S-production must be chosen to continue the derivation. Normally we choose the first one. If we fail, we try to use the second S-production, then the third, and so on, until we succeed in finding a derivation. Earley's algorithm tries to eliminate backtracking, by keeping all possible syntax trees for the previously analysed substring $a_1 a_2 \ldots a_i$. It builds lists of states for each $i = 0, 1, \ldots, n$. A state in the list L_i is a 3-tuple (j, l, k), where j denotes the rank of production concerned, l is a pointer in this production $(0 \leq l \leq l_j$, where l_j is the length of the RHS of j-th production), and k is a number showing the position in the input string w , where the production number j begins to be considered. If $l = l_j$, the state (j, l, k) is called a final state. It means that all symbols in the RHS of production j have been successfully matched against a phrase in the string w . Each list L_i is built by three operations: SCANNER, PREDICTOR, and COMPLETER. Their actions are shown below.

We illustrate this method by an example, and then describe it by a concise algorithm.

Let us consider the grammar G with the following productions:

π_0 : root ⟶ list ##

π_1 : list ⟶ array

π_2 : list ⟶ array , list

π_3 : array ⟶ var (intg , intg)

π_4 : array ⟶ var (intg)

π_5 : var ⟶ A

π_6 : var ⟶ A1

π_7 : intg ⟶ 1

π_8 : intg ⟶ 1 intg ,

and the sets:

$N = \{$ root , list , array , var , intg $\}$

$T = \{$ A , 1 , (, , ,) , ## $\}$.

Let us try to parse the string w = A (1 1) ## .

All syntax trees must begin with the root, i.e. with the tree from fig. 2.3.1a. Thus the state (o , o , o) must be in the list L_o since the production concerned is the production π_0 , and we have not matched anything yet. More intuitively, this state is denoted by a pointer in the production π_0 , and the number i , derived from the list L_i (i.e. i is a pointer in the input string denoted by a full stop):

1) root ⟶ . list ## : o , i.e. (o , o , o).

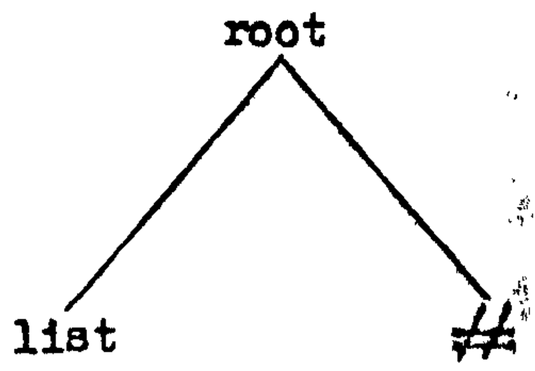
We can "predict" now that a list-production must be used, and thus the states

2) list ⟶ . array : o , i.e. (1, o , o)

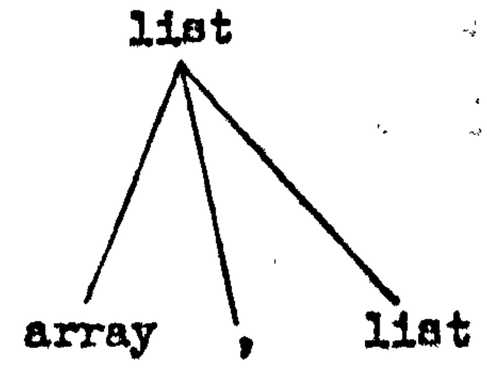
3) list ⟶ . array , list : o , i.e. (2 , o , o)

represented by fig. 2.3.1b, must be added to list L_o . These two steps combined, give the possible syntax trees represented in fig. 2.3.1c.

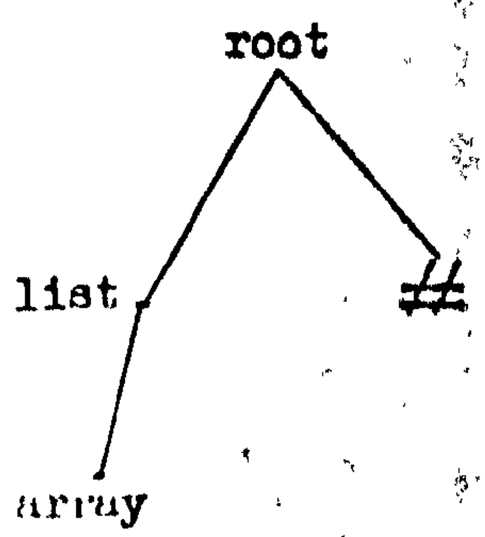
We can predict now (see fig. 2.3.1c) that an array-production must be used, and the states

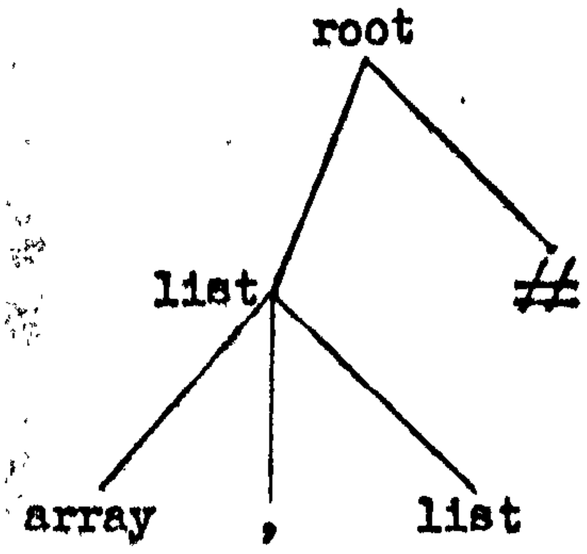


(a)

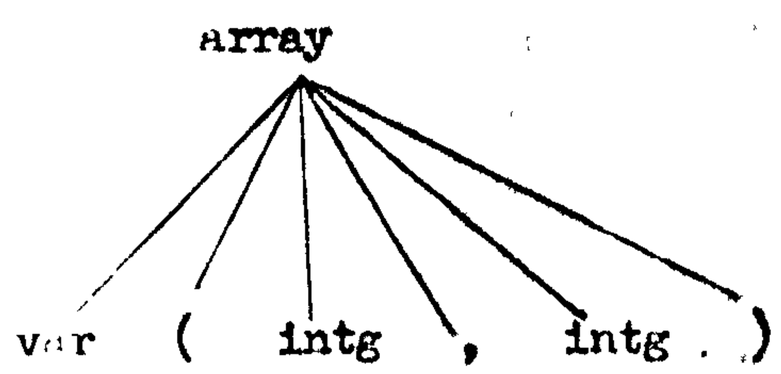


(b)

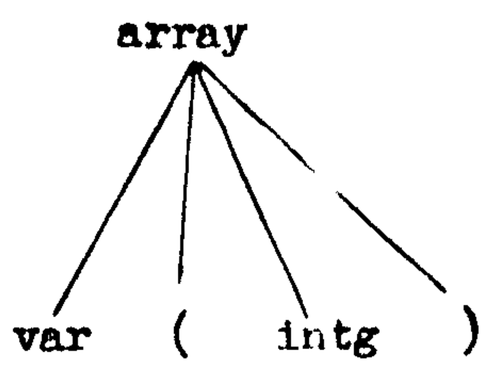


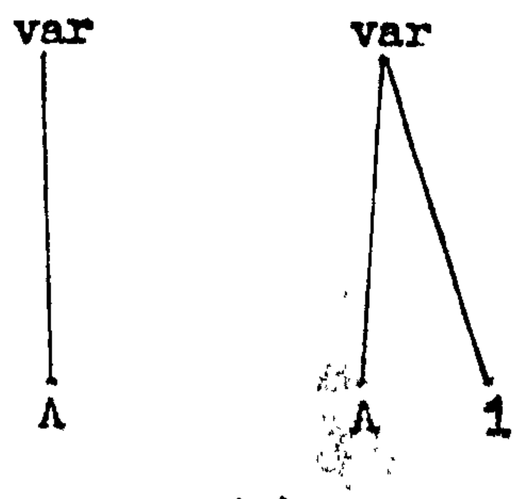


(c)



(d)





(e)

Fig . 2.3.1

4) array ⟶ . var (intg , intg) : o , i.e. (3 , o , o) ,

5) array ⟶ . var (intg) : o , i.e. (4, o , o) ,

must be added to L_o . Finally the states

6) var ⟶ . A : o , i.e. (5 , o , o) ,

7) var ⟶ . A1 : o , i.e. (6 , o , o) ,

must be added (in the same "predictive" way) to L_o.

We scan now the first symbol A of the string w and we look for states with A immediately after the pointer. If we find one, we must move the pointer one position to the right since we have matched this symbol with the symbol immediately after the pointer in the right hand side of that production. Since we have scanned the first symbol in the string, the list L_1 is formed. In our case, thestates

8) var ⟶ A . : o , i.e. (5 , 1 , o)

9) var ⟶ A.1 : o , i.e. (6 , 1 , o)

must be added to L_1.

Since 8) is a final state, i.e. the whole RHS of the production π_5 has been matched against a phrase of the string w , all states in L_o with var immediately after the pointer, must be added to L_1 with the pointer moved one position to the right, to indicate that var has been completely used. Thus we "complete" the list L_1 with the states

10) array ⟶ var . (intg , intg) : o , i.e. (3 , 1 , o)

11) array ⟶ var . (intg) : o , i.e. (4 , 1 , o).

There are no other final states in L_1 , therefore the list L_1 is complete.

There is no variable, immediately after the pointer, in any state of L_1, thus the prediction phase does not add other states to L_1.

We scan now the next symbol "(" and try to match it with the symbol after the pointer in the states of L_1 . The match is successful for the states 10) and 11) and "the scanner" adds the states

12) array ⟶ var (. intg , intg) : o, i.e. (3 , 2 , o)

13) array ⟶ var (. intg) : o, i.e. (4 , 2 , o)

to L_2 . These are not final states, thus the completer cannot act.

But the variable intg occurs immediately after the pointer, and we "predict" that all intg-productions can be used at this stage. The states

14) intg ⟶ . 1 : 2, i.e. (7 , o , 2)

15) intg ⟶ . 1 intg : 2, i.e. (8 , o , 2)

must be added to L_2 . There are no other states that can be added to L_2.

The next symbol to be scanned is a_3 = "1" . The "scanner" adds the states

16) intg ⟶ 1 . : 2 , i.e (7 , 1 , 2) .

17) intg ⟶ 1 . intg : 2 , i.e. (8 , 1 , 2)

to L_3 . The state 16) is a final one. The production π_7 used in state 16) has been initiated in list L_2 . Therefore all states from L_2 with intg immediately after the pointer, must be added to L_3 with the pointer moved one position to the right. Thus the states

18) array ⟶ var (intg . , intg) : o , i.e. (3 , 3 , o)

19) array ⟶ var (intg .) : o , i.e. (4 , 3 , o) ,

must be added to L_3. State 17) has the metasymbol intg after the pointer, thus the "predictor" must add the states

20) intg ⟶ . 1 : 3 , i.e. (7 , o , 3)

21) intg ⟶ . 1 intg : 3 , i.e. (8 , o , 3)

to L_3 . There are no other states that can be added to L_3.

The next symbol to be scanned is a_4 = "1". The "scanner" adds the states

22) intg ⟶ 1 . : 3 , i.e. (7 , 1 , 3)

23) intg ⟶ 1 . intg : 3 , i.e. (8 , 1 , 3)

to L_4. State 22) is a final one, the "completer" must add the state

24) intg ⟶ 1 intg . : 2 , i.e. (8 , 2 , 2),

to L_4 . This again is a final state, carried from the list L_2, thus all states in L_2 with intg immediately after the pointer must be added to L_4 with the pointer moved one position to the right. They are:

25) arrary ⟶ var (intg . , intg) : o , i.e. (3 , 3 , o)

26) array ⟶ var (intg .) : o , i.e. (4 , 3 , o).

Since the state 23) contains the variable intg immediately after the pointer, the prediotor adds to L_4 the following states

27) intg ⟶ . 1 : 4 , i.e. (7 , o , 4)

28) intg ⟶ . 1 intg : 4 , i.e. (8 , o , 4) .

No other states can be added to L_4.

We scan now the symbol a_5 = ')' and the "scanner" adds the state

29) array ⟶ var (intg) . : o , i.e. (4 , 4 , o)

to L_5 . This is a final state "carried" from L_o , thus the completer must add the states

30) list ⟶ array . : o , i.e. (1 , 1 , o)

31) list ⟶ array . list : o , i.e. (2 , 1 , o)

to L_5 ; Since 30) is a final state the "completer" adds the following state

32) root ⟶ list . ## : o , i.e. (o , 1 , o)

to L_5 . There are no other states that can be added to L_5.

Finally the last symbol a_6 = '##' is scanned. The state

33) root ⟶ list ## . : o , i.e. (o , 2 , o)

must be added to L_6 . The presence of this state in L_6 shows that the string w is a correct sentence, since the root-production has been completely matched against it.

We must now build the syntax tree for w . Note that a final state shows that a simple phrase can be generated by the corresponding production. We must search backward through all the information stored in these states, and keep only the final states encountered. The synta. tree can be seen in Fig.2.3.4.

To do this, we need some link pointers between states. In the process of forming the list of states, a new state s was added by "scanner", "predictor" or "completer", starting from a previously existing state q (and q_1 - there are two states in the completing steps). Thus, the state s must have a pointer to the state q, if s was obtained by the "scanner" from q, or two pointers if s was obtained by the completer, the first one to the final state q, the second one to the other state. The list of all states, for our example, together with these link pointers, can be seen in fig. 2.3.2.

We must start with state (0 , 2, 0)(in our example, state 33), store all states to which there is a pointer, and retain only the final states to form the syntax tree of the sentence.

This process is set out in fig. 2.3.3.

The parsing algorithm for the input string

$$w = a_1 a_2 \dots a_m \#\#$$

in the language generated by a context free grammar G with the productions $\Pi_j : A_j \longrightarrow x_1^{(j)} x_2^{(j)} \dots x_{l_j}^{(j)}$, $j = 1, 2, \dots, n$,

and

Π_0 : root $\longrightarrow$ S## , is :

A) The recognizer:

a) At the beginning all lists L_i are empty. Add the state (0 , 0 ; 0) to L_0.

b) For i = 0 to n and for each state s = (j , l , k) in L_i, execute the following steps, until no other states can be added to L_i:

1) Predictor: If s is nonfinal and $x_{l+1}^{(j)}$ is a nonterminal, then for each $x_{l+1}^{(j)}$ -production Π_t, add the state (t , 0 , i) to L_i.

The input string: w = A (1 1) #

State		Intuitive description of the state		Pointers due to
list L_0				
1)	(0 , 0 , 0)	root	. list #	: 0
2)	(1 , 0 , 0)	list	. array	: 0
3)	(2 , 0 , 0)	list	. array , list	: 0
4)	(3 , 0 , 0)	array	. var (Intg , intg)	: 0
5)	(4 , 0 , 0)	array	. var (intg)	: 0
6)	(5 , 0 , 0)	var	. A	: 0
7)	(6 , 0 , 0)	var	. A 1	: 0
list L_1 : a_1 = A				
8)	(5 , 1 , 0)	var	A .	: 0 scanner 6)
9)	(6 , 1 , 0)	var	A . 1	: 0 scanner 7)
10)	(3 , 1 , 0)	array	var . (intg , intg)	: 0 completer 8) 4)
11)	(4 , 1 , 0)	array	var . (intg)	: 0 completer 8) 5)
list L_2 : a_2 = (				
12)	(3 , 2 , 0)	array	var (. intg , intg)	: 0 scanner 10)
13)	(4 , 2 , 0)	array	var (. intg)	: 0 scanner 11)
14)	(7 , 0 , 2)	intg	. 1	: 2
15)	(8 , 0 , 2)	intg	. 1 intg	: 2
list L_3 : a_3 = 1				
16)	(7 , 1 , 2)	intg	1 .	: 2 scanner 14)
17)	(8 , 1 , 2)	intg	1 . intg	: 2 scanner 15)
18)	(3 , 3 , 0)	array	var (intg . , intg)	: 2 completer 16) 12)
19)	(4 , 3 , 0)	array	var (intg .)	: 2 completer 16) 13)
20)	(7 , 0 , 3)	intg	. 1	: 3
21)	(8 , 0 , 3)	intg	. 1 intg	: 3
list L_4 : a_4 = 1				
22)	(7 , 1 , 3)	intg	1 .	: 3 scanner 20)
23)	(8 , 1 , 3)	intg	1 . intg	: 3 scanner 21)
24)	(8 , 2 , 2)	intg	1 intg .	: 2 completer 22) 17)
25)	(3 , 3 , 0)	array	var (intg . , intg)	: 0 completer 24) 18)
26)	(4 , 3 , 0)	array	var (intg .)	: 0 completer 24) 13)
27)	(7 , 0 , 4)	intg	. 1	: 4
28)	(8 , 0 , 4)	intg	. 1 intg	: 4
list L_5 : a_5 =)				
29)	(4 , 4 , 0)	array	var (intg) .	: 0 scanner 26)
30)	(1 , 1 , 0)	list	array .	: 0 completer 29) 2)
31)	(2 , 1 , 0)	list	array . , list	: 0 completer 29) 3)
32)	(0 . 1 . 0)	root	list . #	: 0 completer 30) 1)
list L_6 : a_6 = #				
33)	(0 , 2 , 0)	root	list # .	: 0 scanner 32)

Fig. 2.3.2. - 25

Number	The current examined state intuitive representation			This state was formed by	It has link pointers to	The next states to be examined
(33)	root	list ≠ .	: 0	scanner	32)	32)
32)	root	list . ≠	: 0	completer	30) 1)	30) 1)
(30)	list	array .	: 0	completer	29) 2)	29) 2) 1)
(29)	array	var (intg) .	: 0	scanner	26)	26) 2) 1)
26)	array	var (intg .)	: 0	completer	24) 13)	24) 13) 2) 1)
(24)	intg	1 intg .	: 2	completer	22) 17)	22) 17) 2) 1)
(22)	intg	1 .	: 3	scanner	20)	20) 17) 13) 2) 1)
20)	intg	. 1	: 3	predictor	-	17) 13) 2) 1)
17)	intg	1 . intg	: 2	scanner	15)	15) 13) 2) 1)
15)	intg	. 1 intg	: 2	predictor	-	13) 2) 1)
13)	array	var (. intg)	: 0	scanner	11)	11) 2) 1)
11)	array	var . (intg)	: 0	completer	8) 5)	8) 5) 2) 1)
(8)	var	A .	: 0	scanner	6)	6) 5) 2) 1)
6)	var	. A	: 0	predictor	-	5) 2) 1)
5)	array	. var (intg)	: 0	predictor	-	2) 1)
2)	list	. array	: 0	predictor	-	1)
1)	root	. list ≠	: 0	-	-	-

Fig. 2.5.3

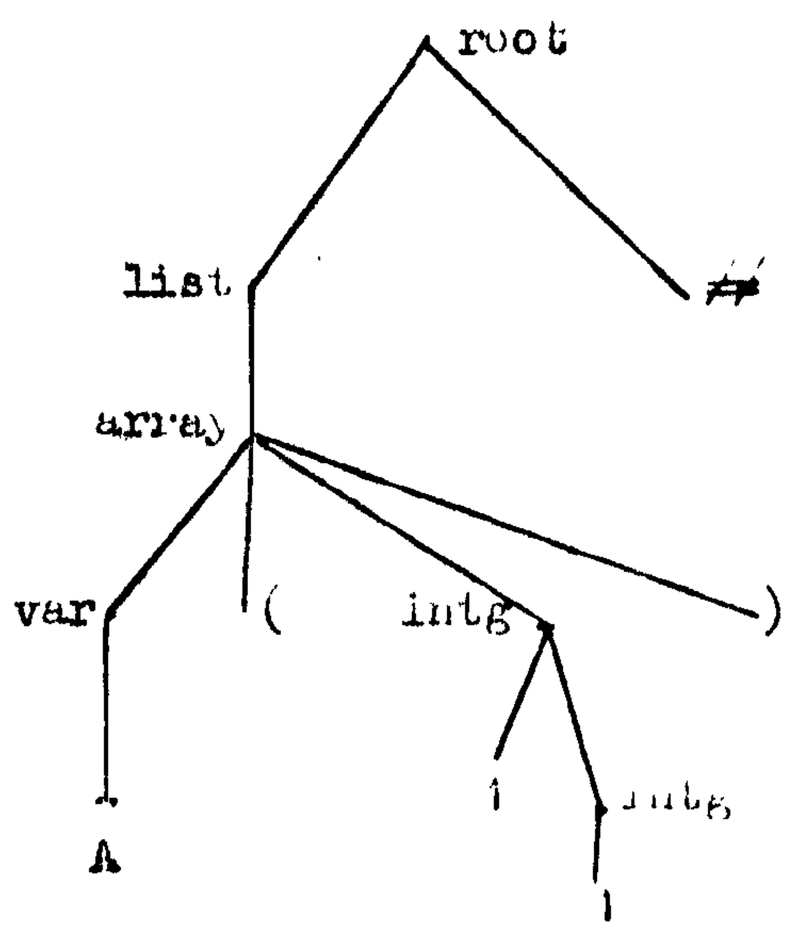


Fig. 2.5.4

2) Completer: If s is a final state and (j_1, k_1, t) is in L_k and $A_j = x^{(j)}_{k_1+1}$, then add the state (j_1, k_1+1, t) to L_i. Add two link pointers, the first to s, and the second to (j_1, k_1, t).

3) Scanner: If s is nonfinal and $x^{(j)}_{l+1}$ is terminal, then if $x^{(j)}_{l+1} = a_{i+1}$, add the state $(j, l+1, k)$ to L_{i+1}. Add a link pointer to the state s.

4) If L_{i+1} is empty, then reject the string.

c) If $L_{n+1} = \{(0,2,0)\}$, then the string is a sentence in the language.

B) For constructing the syntax tree of the sentence w, begin with the state $(0, 2, 0) \in L_{n+1}$, check in order all states connected by link pointers, and retain only the final states.

2.4. A GLOBAL TOP-DOWN PARSER

In a top-down parsing algorithm we must choose an S-production with which to start. Our target is the input string $w = a_1\ a_2 \ldots a_n\ \#$. In the previous algorithm one input character 'a_1' is read at a time. Knowing a_1, some S-productions can be eliminated. If, in some way, we read the whole string w, it may be possible to decide which of the S-productions must be chosen.

Unger [U2] has suggested such an algorithm. Suppose that the input string is $w = a\ b\ b\ c\ d\ d\ e\ f$ and the grammar has the productions

$$\pi_1 : S \longrightarrow a\ S_1\ S_2\ d\ S_3\ f$$

$$\pi_2 : S \longrightarrow a\ d\ S_2\ g$$

$$\pi_3 : S_1 \longrightarrow b\ S_2$$

$$\pi_4 : S_2 \longrightarrow b$$

$$\pi_5 : S_2 \longrightarrow c\ d$$

$$\pi_6 : S_3 \longrightarrow e$$

$$\pi_7 : S_3 \longrightarrow a\ S_1\ .$$

The S-production $\pi_1 : S \longrightarrow a\ S_1\ S_2\ d\ S_3\ f$ is normally the first choice. The the first character 'a' of this S-production matches the first input symbol, 'f' matches the last one, $S_1\ S_2$ must be tentatively matched against $b\ b\ c$ or $b\ b\ c\ d$ (since we do not know which one of those two input characters 'd', corresponds to 'd' in the right hand side of the production π_1), and S_3 must match $d\ e$, or e, respectively.

We define an A-phrase as a phrase that can be derived from the nonterminal A, i.e. α is an A-phrase if $A \overset{+}{\Longrightarrow} \alpha$.

We need to match $S_1\ S_2$ against $b\ b\ c$. For doing this, the string $b\ b\ c$ must be decomposed into two substrings α_1 and α_2 such that α_1 is an S_1-phrase and α_2 is an S_2-phrase. There may be many ways of doing this.

Since, even if we succeed in matching $S_1\ S_2$ against $b\ b\ c$ there is the possibility of failing to match the rest of the string against the rest of the right hand side of the production π_1, we need a

strategy to check all ways of obtaining a derivation of the string.

We assume that the productions of our grammar are ordered, and all A-productions are arranged consecutively for any metasymbol A. Certainly this can be done for any context free grammar.

At any stage of the parsing process the important problem which we face is that of deciding if a given terminal string is an A-phrase. We need a procedure which answers this question and gives the derivation

$$A \overset{+}{\Longrightarrow} \alpha \ . \qquad (2.4.1)$$

We are going to describe the procedure PHRASE(A, α, R). This procedure recognises if α is an A-phrase. R is a logical variable which is 'True' if α is an A-phrase, and 'False' if α is not an A-phrase.

Let

$$\Pi(A,i): \qquad A \longrightarrow u_1^{(i)} A_1^{(i)} u_2^{(i)} A_2^{(i)} \ldots u_{l_i}^{(i)} A_{l_i}^{(i)} u_{l_i+1}^{(i)} ,$$

be the i-th A-production, i=1,2, ... ,n_A, where $u_j^{(i)}$ are terminal strings, possibly empty, and $A_j^{(i)}$ are metasymbols.

We need to choose one of these productions to start the derivation (2.4.1). For doing this we check all A-productions in order, and for each one it must be verified if the right hand side of the production $\Pi(A,i)$ can match the string α. If the strings $u_j^{(i)}$, j=1,2, ... , l_i+1, do not find the identical correspondent substrings in the string α, then the production $\Pi(A,i)$ must be rejected. Otherwise the string α is decomposed into substrings,

$$\alpha = u_1^{(i)} \alpha_1^{(i)} u_2^{(i)} \alpha_2^{(i)} \ldots u_{l_i}^{(i)} \alpha_{l_i}^{(i)} u_{l_i+1}^{(i)}, \qquad (2.4.2)$$

and $\alpha_j^{(i)}$ must be analysed as $A_j^{(i)}$-phrases. There can be more than

one decomposition (2.4.2) and there must be a possibility to access all of them in an ordered way. Under /Uz/ takes first decomposition (2.4.2) first, which has $u_j^{(i)}$ of minimum length. Certainly $u_j^{(i)}$ must have the length at least one, otherwise $u_j^{(i)}$ cannot be an $A_j^{(i)}$-phrase. (Remember, all our grammars are considered ε-free). If all $A_j^{(i)}$-phrases have their lengths at least $m_j^{(i)}$, then $u_j^{(i)}$ must have its length at least $m_j^{(i)}$, and all decompositions not satisfying this condition can be rejected.

Since $u_j^{(i)}$ must be an $A_j^{(i)}$-phrase for all $j=1, 2, \ldots, J_i$, we need to call Parse($A_j^{(i)}$, $u_j^{(i)}$, R), i.e. the subroutine PARSE is used recursively. A flowchart for this subroutine can be found in fig. 2.4.1.

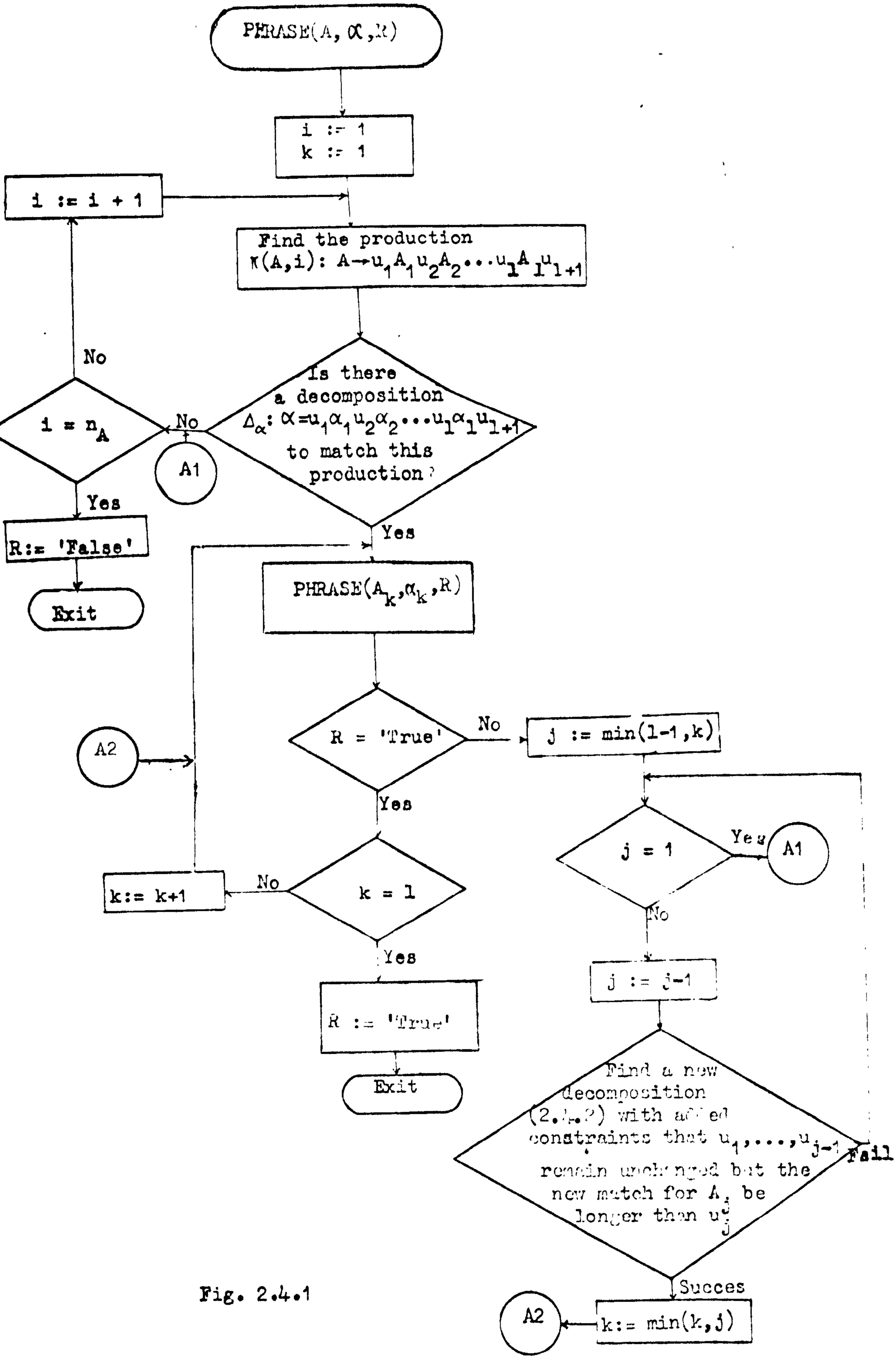

PHRASE(A, α, R)
i := 1
k := 1
i := i + 1
Find the production
R(A,i): A→u1A1u2A2...ulAlul+1
No
i = nA
No
A1
Is there
a decomposition
Δα: α=u1α1u2α2...ulαlul+1
to match this
production?
Yes
Yes
R:= 'False'
Exit
PHRASE(Ak,αk,R)
A2
R = 'True'
No
j := min(l-1,k)
k:= k+1
No
k = 1
j = 1
Yes
A1
No
Yes
j := j-1
R := 'True'
Exit
Find a new
decomposition
(2.4.2) with added
constraints that u1,...,uj-1
remain unchanged but the
new match for Aj be
longer than uj
Fail
Succes
A2
k:= min(k,j)

Fig. 2.4.1

2.5. PARSING . SIMPLE PRECEDENCE GRAMMAR

For any grammar $G = (N , T , P , S)$ with the vocabulary $V = N \cup T$, the simple precedence relations $\lessdot$, $\doteq$, $\gtrdot$ are defined for all pairs $(x , y) \in V \times V$ [⊛], in the following way:

1) $x \doteq y$ if there is a production $\pi \in P$, $\pi : A \longrightarrow \alpha x y \beta$, for some $\alpha, \beta \in V^*$. That is, the symbols x and y have the same precedence if they appear adjacent in a RHS of a production.

This means that, in a sentential form with x and y adjacent, there is the possibility that x and y can be reduced at the same time.

2) $x \lessdot y$ if there is a production $A \longrightarrow \alpha x B \beta$, and there is a derivation $B \overset{+}{\Longrightarrow} y\eta$, for some $\alpha, \beta, \eta \in V^*$. That is, if x and y are adjacent in a sentential form, it is possible that y and some consecutive characters on the right can be reduced first to a variable B.

3) $x \gtrdot y$ if there is a production $A \longrightarrow \alpha B C \beta$, and there are the derivations $B \overset{+}{\Longrightarrow} \gamma x$
$C \overset{*}{\Longrightarrow} y \eta$,

for some strings $\alpha, \beta, \gamma, \eta \in V^*$. In other words, there exists a sentential form $s = \alpha_1 \alpha \gamma x y \eta \beta \beta_1$, such that it is possible to reduce first γx to B , then $y\eta$ to C, to obtain the sentential form $\alpha_1 \alpha B C \beta \beta_1$, i.e. x has precedence over y , since it can be replaced in the sentential form s before y.

The grammar G is a simple precedence grammar if

a) for all $(x , y) \in V \times V$, at most one precedence relation holds, i. e, it is known precisely which one of the symbols x and y must be reduced first, if they appear consecutively in a sentential form.

b) there are no two productions in the grammar G with identical right hand sides, i.e. any string can be reduced in at most one way.

c) the grammar is ε-free.

⊛ $A \times B$ denotes the cartesian product of the sets A and B.

It follows that a derivation $A \overset{+}{\Longrightarrow} A$ is impossible.

We choose a new terminal symbol $\#\#$ to mark the beginning and the end of the string, and for all $x \in V$ we define $\#\# \lessdot x$, and $x \gtrdot \#\#$. The parsing algorithm for a given string

$$w = \#\# a_1 a_2 \cdots a_n \#\#,$$

uses a parsing stack. Initially this parsing stack contains the symbol $\#\#$, that is, the first input symbol. Let x denote the topmost stack symbol. The parsing algorithm is:

1) Read the next input symbol a_k. If a_k is $\#\#$ go to step 7.
2) If the relation between x and a_k is $\lessdot$ or $\doteq$ then go to step 5.
3) If the relation between x and a_k is $\gtrdot$, go to step 6.
4) There is no relation between x and a_k. Output an error message and stop.
5) Put the symbol a_k on the stack ($x := a_k$), and go to step 1.
6) A simple phrase has been stacked. Its end is the topmost character, and all its characters are related by $\doteq$. Find the production for which its right hand side matches this simple phrase. Replace this simple phrase in the stack, by the LHS of the corresponding production. Go to step 1.
7) If the stack contains $\#\# S$, then the string w has been recognised as a sentence. Otherwise w is not a sentence.

The sequence of productions used for reducing the stack, gives the canonical derivation of w.

A flowchart for this algorithm can be found in section 4.3., fig. 4.3.3.

2.6. ACCEPTORS

A context-free language is characterized by the fact that its sentences are those strings acceptod by a pushdown automaton /A3,G1/. Its subclass of regular languages is the same as the class of languages accepted by the family of finite state machines /A3,G1/.

A regular language is a context-free language with all productions of the form $A \longrightarrow a\,B$ or $A \longrightarrow a$, where a is a terminal symbol and A and B are variables.

A finite-state machine (or finite automaton) is a system $M = (K , T , \delta , so , F)$ where

- K is a finite nonempty set of states
- T is a finite input alphabet
- $So \in K$ is the initial state
- $F \subset K$ is a set of final states and
- δ is a transition function defined as follows:

a) $\delta : K \times T \longrightarrow K$, and the finite-state machine is called a deterministic finite-state machine.

b) $\delta : K \times T \longrightarrow 2^K$, and the finite-state machine is called a nondeterministic finite state machine.

The transition function δ can be extended for $K \times T^*$, as follows:

$$\delta(q , \varepsilon) = q$$

$$\delta(q , w a) = \delta(\delta(q , w) , a) ,$$

for all states $q \in K$, all input symbols $a \in T$, and for any string $w \in T^*$.

The set of strings accepted by a deterministic finite state machine M is

$$T (M) = \{ w \in T^* \mid \delta(so , w) \in F \} .$$

The set of strings accepted by a nondeterministic finite-state machine M is

$$T (M) = \{ w \in T^* \mid \delta(so , w) \cap F \neq \phi \} .$$

It can be proved [A3, G1] that the families of these two sets are identical, i.e. each set accepted by a deterministic finite state machine is accepted by a nondeterministic finite state machine and conversely.

An important result [A3, G1] is that a language is regular if and only if it is accepted by a finite state machine.

A pushdown automaton (p.d.a.) is a system $M=(K,T,\Gamma,\delta,s_0,Z_0,F)$, where

- K, T, F and s_0 have the same meaning as above
- Γ is a finite alphabet called the stack alphabet
- $Z_0 \in \Gamma$ is the initial stack symbol
- the transition function δ is defined as

$$\delta : K \times T_\varepsilon \times \Gamma \longrightarrow 2^{K \times \Gamma^*}$$

where $T_\varepsilon = T \cup \{\varepsilon\}$ and 2^D is the family of all subsets of the set D.

Intuitively, a p.d.a. can be described as in fig. 2.6.1.

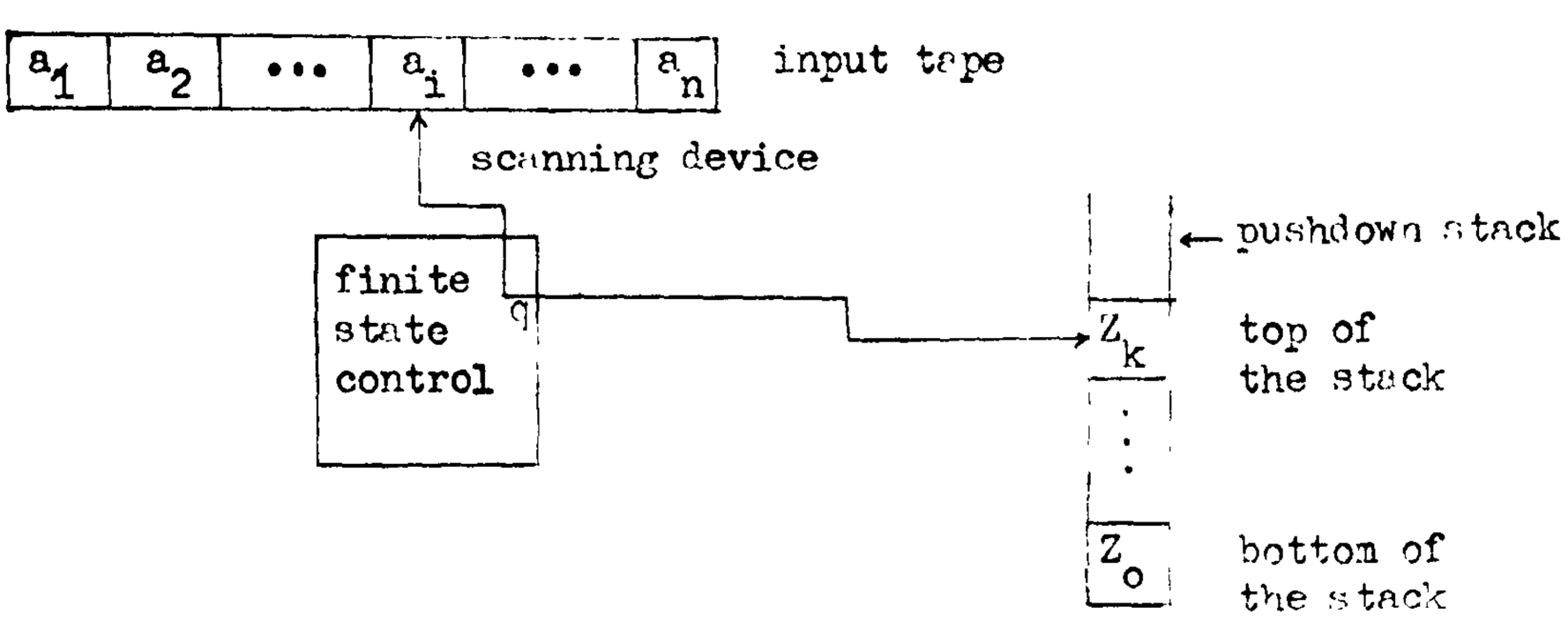


Fig. 2.6.1.

At each moment the state of the p.d.a. automaton is precisely described by the state q of the finite state control, the position of the scanning device on the input tape and the stack $\gamma = Z_k Z_{k-1} \dots Z_0$. These three elements form the configuration $(q , a_i \beta , \gamma)$, of the p d a at that moment.

We say that $(q , a\beta , Z\gamma) \vdash (p , \beta , \gamma_1\gamma)$ iff $(p , \gamma_1) \in \delta(q , a , Z)$. The relation $\vdash^*$ is the symmetric transitive closure of $\vdash$.

The language accepted by the p.d.a. M by the final state is

$$\mathcal{T}(M) = \{ w \mid (s_0, w, Z_0) \vdash^{*} (q, \varepsilon, \gamma), \text{ for some } q \in F \}.$$

The language accepted by the p.d.a. M by emptied stack is

$$\mathcal{E}(M) = \{ w \mid (s_0, w, Z_0) \vdash^{*} (q, \varepsilon, \varepsilon) \}.$$

An important result is that a language is context-free if and only if it is accepted by a pushdown automaton by final state (or by emptied stack).

Because these automata accept the strings in their corresponding languages, they are called acceptors (or recognizers). They can be transformed into transducers, which permit one to obtain a very efficient parsing algorithm.

A pushdown transducer is a pushdown automaton which has an output facility. Formally defined a pushdown transducer is a system $(K, T, \Gamma, \Delta, \delta, s_0, Z_0, F)$, where

- K, T, Γ, s_0, Z_0, and F have the same meaning as in the definition of the pushdown automata
- Δ is a non-empty finite set of output symbols
- the transition function

$$\delta: K \times T_\varepsilon \times \Gamma \longrightarrow P(K \times \Gamma^* \times \Delta^*)$$

has the property that $\delta(q, a, z)$ is a set containing a finite number of elements for any $(q, a, z) \in K \times T_\varepsilon \times \Gamma$.

For a given p.d.a. $M = (K, T, \Gamma, \delta, s_0, z_0, F)$ we say that a pushdown transducer $M_t = (K, T, \Gamma, \Delta, \delta_t, s_0, z_0, F)$ is associated with M if for each triple $(q, a, z) \in K \times T_\varepsilon \times \Gamma$, we have

$$(p, r) \in \delta(q, a, z) \Longleftrightarrow (\exists y \in \Delta^*) \{ (p, r, y) \in \delta_t(q, a, z) \},$$

(see [01]). This means that if it is possible for the p.d.a. to change the configuration (q, a, z) into (p, ε, r) the same change is possible for the pushdown transducer if we are not interested

in the output.

Thus, the important definitions regarding context-free languages and their acceptors have been presented. We will meet them all in the next sections of this thesis.

CHAPTER III

THE PROBLEM OF ERROR CORRECTION AND A SURVEY OF PREVIOUS WORK

3.1. THE DISTANCE BETWEEN TWO STRINGS

Let α and β be two strings over the same alphabet V , i.e.

$$\alpha = a_1 a_2 \dots a_n ,$$

$$\beta = b_1 b_2 \dots b_m ,$$

with $a_i \in V$, $i = 1 , 2 , \dots , n$ and $b_j \in V$, $j = 1 , 2 , \dots , m$.

Since we are going to define and study the problem of error correction in a programming language, we need to measure the similarity between two strings. For this purpose we shall define a distance between them.

In the process of communication between man and machine errors appear as a result of misspelling of words, deleting or inserting of characters. Some edit operations are needed to correct these errors. Formally an edit operation is a pair (a , b) of characters not both null , i.e. $(a , b) \neq (\varepsilon , \varepsilon)$ [W 2]. We say that the string β is obtained from the string α by performing the edit operation $e = (a , b)$ if there are two strings $r_1 , r_2 \in V_t^*$ such that

$$\alpha = r_1 a r_2 ,$$

and

$$\beta = r_1 b r_2 .$$

We write $\beta = e (\alpha)$.

We call the edit operation $e = (a , b)$ a change operation if $a \neq \varepsilon$, $b \neq \varepsilon$, and $a \neq b$. If $a = b$ we prefer to call the edit operation (b , b) an (unchanging) empty operation since its effect is null when it is performed (compare with Wagner [W2]). The edit operation (ε , b) is called an insert operation and the edit operation (a , ε) is called a delete operation.

From statistical experience or from other considerations, with each edit operation $e = (a , b)$ we associate a weight (cost) u (e).

Let S be a sequence of edit operations e_1 , e_2 , ... e_l that transform the string α into the string β , i.e. $\beta = e_l(e_{l-1}(\dots e_1(\alpha)) \dots)$.

We define the weight (cost) of the sequence S by

$$u(S) = \sum_{i=1}^{l} u(e_i). \qquad (3.1.1.)$$

There always exists a sequence S of edit operations, which transform a string α into the string β . A possible one is the following string of delete operations and insert operations

$$(a_1 , \varepsilon) , (a_2 , \varepsilon) , \dots (a_n , \varepsilon) , (\varepsilon , b_1) , (\varepsilon , b_2) , \dots , (\varepsilon , b_m).$$

Let $S (\alpha , \beta)$ be the set of all sequences of edit operations that transform the string α into the string β . There is a weight associated with each sequence $S \in S(\alpha , \beta)$. We define the distance between the strings α and β as the minimum weight of such a sequence, i.e.

$$d(\alpha, \beta) = \min_{S \in S (\alpha , \beta)} u(S). \qquad (3.1.2.)$$

Let So be a sequence of edit operations such that

$$d (\alpha , \beta) = u (So).$$

This sequence So shows us which characters in the string α have been deleted, which characters have been inserted into the string β , and which chracters (a_i , b_j) correspond one to another by change or empty operations. We define a trace T between the string α and β corresponding to the sequence So. Intuitively the trace is the correspondence between the characters $a_i - b_j$ defined by the sequence So of edit operations, i.e. if (a_i , b_j) is in So then $(i , j) \in T$. Thus a trace $T (\alpha , \beta)$ between the strings α and β , is a set of pairs of integers (i,j) with the following properties:

1) $1 \leq i \leq n$, $1 \leq j \leq m$,

2) For any two pairs $(i_1 , j_1) , (i_2 , j_2) \in T$

   a) $i_1 = i_2$ if and only if $j_1 = j_2$.

   b) if $i_1 < i_2$ then $j_1 < j_2$.

The weight (cost) of the trace is the weight of the corresponding sequence of edit operations, i.e.

$$u (T) = u (So) . \qquad (3.1.3.)$$

Often we need to compute the distance $d (\alpha , \beta)$ between the strings α and β. An algorithm for doing this can be found in Wagner [W2].

Let d_{ij} denote the distance between the strings

$$a_1 a_2 \cdots a_i ,$$

and

$$b_1 b_2 \cdots b_j ,$$

for $o \leq i \leq n$, $o \leq j \leq m$. Here the string $c_1 c_2 \cdots c_k$, for $k = 0$ is considered to be the empty string ε. Thus $d_{i,o}$ is the weight (cost) of deleting all characters $a_1, a_2, \ldots, a_i$, i.e.

$$d_{io} = \sum_{k=1}^{i} u ((a_k , \varepsilon)) ,$$

and $d_{o,j}$ is the cost of inserting all characters $b_1 , b_2 , \cdots , b_j$, i.e.

$$d_{o,j} = \sum_{k=1}^{j} u ((\varepsilon , b_j)) ,$$

and $d_{o,o} = 0$.

Then $d_{i+1 , j+1}$ can be computed according to the formula:

$$d_{i+1 , j+1} = \min \left\{ \begin{array}{l} d_{i+1 , j} + u ((\varepsilon , b_{j+1})) , \\ d_{i, j+1} + u ((a_{i+1} , \varepsilon)) , \\ d_{ij} + u ((a_{i+1} , b_{j+1})) \end{array} \right\}, \qquad (3.1.4)$$

(see [W2]). In other words, the formula (3.1.4) says that the string

$$S (\beta , 1 , j+1) = b_1 b_2 \cdots b_{j+1}$$

can be obtained from the string

$$S (\alpha , 1 , i+1) = a_1 a_2 \cdots a_{i+1}$$

in one of the following three ways:

a) the string $S(\beta, 1, j) = b_1 b_2 \dots b_j$ is obtained from the string $S(\alpha, 1, i+1)$ and the symbol b_{j+1} is inserted. In this case the distance would be

$$d_{i+1,j} + u((\varepsilon, b_{j+1})).$$

b) the string $S(\beta, 1, j+1)$ is obtained from the string $S(\alpha, 1, i) \equiv a_1 a_2 \dots a_i$ and the symbol a_{i+1} is deleted. In this case the distance would be $d_{i,j+1} + u((a_{i+1}, \varepsilon))$.

c) the string $S(\beta, 1, j)$ is obtained from the string $S(\alpha, 1, j)$, and the last character b_{j+1} is obtained from the last character a_{i+1}. In this case the distance would be

$$d_{i+1,j+1} + u((a_{i+1}, b_{j+1})).$$

3.2. THE PROBLEM OF ERROR CORRECTION IN A PROGRAMMING LANGUAGE

We are concerned in this thesis with the transmission of information in a man-machine channel. The assumption is that the information is properly coded, i.e. we assume that the program is initially correct. But, between the programmer's mind and the computer memory, some errors occur. Such errors can happen when the programmer himself writes his program on paper, or when, further, the program is punched on cards, or sent in any other way to the computer. Thus, although the program received by the computer is erroneous, it is an erroneous version of a correct pr

It would be useful to study the statistical behaviour of such a channel. Some work in this direction can be found in [L1] where Litecky and Davis have studied the errors made by a group of students learning Cobol, and the error diagnosis given by the Cobol-compiler. Also Damerau [D1] and Morgan [M2] have found that over 80 per cent of errors are simple ones. Damerau has noted "An inspection of the 3 items rejected because of spelling errors showed that over 80 per cent fell into one of four classes of single error -- one letter was wrong, or one letter was missing, or an extra letter had been inserted, or two adjacent characters had been transposed," and Morgan said: "This report was confirmed by the author after an analysis of several hundred student programs written in the FORTRAN and PL/1 languages at Cornell."

As has already been said, a context free language is a good model for a programming language. We can find a context free language L for almost all programming languages, such that a correct program is a sentence of L. But we do not say that all sentences of L are correct programs, because it may be that some semantic rules are not respected by a sentence in L.

The problem of error correction can be stated in the following way. A correct sentence $\alpha \in L$ has been sent through the channel and a string β has been received. If β coincides with α, no errors have occurred.

If β differs from α, but is itself a correct sentence in L, no errors can be detected. If β is not a sentence of L then errors are detected. To correct them means to transform the string β into the most probable sentence $\alpha \in L$. Our approach will be to find that sentence $\alpha \in L$ that minimises the distance $d(\beta, \alpha)$.

Although the approach to error correction is to transform any program into a correct one, we must be aware of the fact that the received string is not a random string, but an erroneous version of a correct one. It follows that it would be useful to define the distance $d(\beta,\alpha)$ as in (3.1.2), with the weight function u representing the statistical behaviour of the channel, i.e. the frequency of the corresponding edit operations. Based on this distance it seems that the nearest program is the most probable one.

In the second chapter we have described some parsing methods for a context free language. All these parsers fail to analyse an incorrect sentence, but detect the existence of at least one error. In other words all existing parsers have an error detection capability. Still, there is a great difference between the error detection capability of the various parsers. Early compilers were designed to refuse to analyse a wrong program, while outputting an error message on encountering the first error. This was a very inefficient approach. Today it is expected that a compiler should detect as many errors as possible, or, even, to correct some of them. To do this we need to remove the effect of the error encountered and to decide how to continue to parse the input string. Often this means that one must find a variant to correct those errors. This capability of removing the effect of an error and the decision on how to continue the parsing of the string, without skipping any portion of the input string, is called error recovery. If a parser has this capability we say that it is an error recovery parser.

We say that a parser is an error correcting parser, if it transforms any input string $w \in T^*$, into the most similar sentence u, and parses this sentence. In other words, the parser analyses the input string w, and outputs the syntax tree of the sentence u, that satisfies the equality

$$d(u, w) = \min_{v \in L} d(v, w),$$

where $d(u, w)$ is the distance between the strings u and w, defined in section 3.1. Certainly if we have the syntax tree of u, we know perfectly well the sentence u.

In the process of parsing, the input characters are scanned, usually, from left to right. When an error is detected the parser has scanned some input characters, and the next input character is a_i. We say that i is the detection point of the error. This does not mean that the character a_{i-1} (or a_i) is erroneous. It only tells us that the string

$$a_1 a_2 \cdots a_{i-1} a_i$$

is not a prefix of any sentence. For the same string, and the same error, this point can be different for different parsers, and can be far away from the real point of the error, as will be shown later.

3.3. A SURVEY OF THE PREVIOUS WORK IN ERROR RECOVERY AND ERROR CORRECTION TECHNIQUES

There are three areas in which work has been done in connection with error-correction in a language.

- error correction of spelling errors in a dictionary of English words, in an information retrieval system, or in the lexical analysis phase of a compiler.
- practical error recovery and error correction in a compiler, and
- a theoretical approach to error correction in a context-free language.

3.3.1. Error correction of misspelled words

In an information retrieval system, or in the lexical part of a programming language, a dictionary of English words (or names) is used. The correction of spelling errors is very important in both fields. Also, a solution possible in one field can be unsatisfactory in another field.

Work on spelling correction was reported many years ago. Blair (1960) [B2] has proposed a method of compression-decompression for error correction in a set of English words. Davidson (1962) [D2] gave an algorithm to retrieve misspelled airline passenger names.

The first work on spelling correction in the context of compilers can be found in Freeman's (1964) article [F4]. A review of spelling correction has been given by Morgan (1970) [M2]. Morgan gives techniques for spelling correction in compilers and operating systems.

Work in error correction of misspelled words in an information retrieval system has been done at Brunel by the "Associative Parallel Processor" group (see Dyke and Lea, (1975), [D5], and Donnely [D4]).

Since trying to match the input word against the words of a dictionary is usually done by a sequential search clearly it is more efficient to use an associative parallel processor to do this work in parallel.

Riesman and Hanson (1974) [R1], and Ullman (1976) [U1] use the n-gram technique for error correction in a large dictionary of English words. An n-gram is a string of length n. To each n-gram $g = a_1 a_2 \ldots a_n$, a weight $u(g)$ is associated. $u(g)$ is one if there is a word in the dictionary that contains g as a substring, and zero otherwise. These n-grams can be positional. For example, if n is three, to all trigrams $a_1 a_2 a_3$ a weight u_{124} is defined as one if there is a word $w = b_1 b_2 \ldots b_m$ in the dictionary such that $b_1 = a_1$, $b_2 = a_2$ and $b_4 = a_3$. For all positions i_1, i_2, i_3, $i_1 \leq m$, $i_2 \leq m$, $i_3 \leq m$, in the word w, the weights u_{i_1, i_2, i_3} are defined similarly. The set of all n-grams together with their weights (positional or nonpositional), is called "the syntax" of the dictionary. Since there are 26^n n-grams, the method is practical for n small and in a relatively large dictionary.

A more theoretical approach to the problem of misspelling is taken by Wagner et al [L3, W1, W2]. Since it is an efficient method of spelling correction in the case of a small dictionary, we have presented it in section 4.1., adapted to our purposes.

3.3.2. PRACTICAL ERROR CORRECTION AND RECOVERY BY COMPILERS

The work in error correction and recovery by compilers begins with Irons (1963) [I2] and Freeman (1964) [F4]. Irons makes local insertions, deletions, or substitutions in the input string, until a correct sentence is produced. When more than one correction is possible, Irons keeps all possible derivations for the scanned input string. He has a prepared table of successors and alternates for each

symbol. When an error is detected some possible local corrections are taken from this table. However garbled the input string, it is manipulated until a sentence is obtained.

Freeman has designed a compiler for CORC - the Cornell computing language, which corrects all errors in the input string, and runs the corrected program. Much attention is given to spelling correction. But his approach to error correction is concerned only with CORC language.

Levy [L4] attempts to give a theoretical solution for error correction in deterministic context-free languages. However Levy himself states in the abstract of his thesis that "the formal model is not practical" and he suggests some heuristics to improve its efficiency. This can be done, for example, by renouncing the correction of some errors. His error correcting algorithm has a "backward move" and a "forward move", hich try to locate the minimum left and right context where the errors have occurred.

LaFrance [L1, L2] based his error correction method on Floyd's productions. These productions are derived from the productions of the grammar, but incorporate some context in them. He uses some tables of symbols which can follow a given symbol, and generates all strings of two or three symbols which are correct at the point of error.

Gries [G3], and Conway and Wilcox [C4] store decisions of recovery based on the implementer's knowledge of errors. The technique fails if unanticipated errors occur.

The error correction in a simple precedence language is studied by Wirth and Weber [W3 , W4], Gries [G4], Leinius [L6], and Graham and Rhodes [G2].

Wirth and Gries have suggested local corrections which, when no relation exists between the topmost stack symbol andthe incoming input character, transform this situation locally, by deleting the incoming

input symbol, or by inserting some characters. This method seems to work for many errors but it is not a general method for error correction.

A good method for error correction in a simple precedence language is given by Leinius [L6]. He considers that automatic error correction must be made at phrase level. Leinius tries to find the smallest substring on the stack, that must be reduced to a metasymbol. Although the amount of context information taken into account to correct the error is limited, his method works well for the majority of practical errors.

The approach taken by Graham and Rhodes [G2], lies between error correction and error recovery. The parsers analysed are bottom-up, with no back-up. When an error is encountered, they try to obtain more information about the error, by analysing the context.

These are the results from which this thesis starts in the class of simple precedence languages. While Leinius uses only the information acquired from the scanned input string, Graham and Rhodes try to analyse the surrounding context in which the error occurs, before making any correction. They continue to condense the string beyond the detection point of the error. For doing this there is a "backward move", that tries to condense the string stack (i.e. the portion preceding the detection point of the error). Then, a "forward move" attempts to condense the string beyond the detection point of the error. This move terminates if more errors occur. In this thesis this idea is extended into a "global move", that scans the entire input string, and makes all possible reductions. Then a correction phase analyses the stack, which contains now all the information available from the input string, and corrects the errors.

Graham and Rhodes state, about Leinius's work: "He recognises that if the parsing stack, followed by the current input symbol, contains any sequence of symbols a b α c d , where a, b , c , d , are single symbols, α is a sequence of symbols, a $\lessdot$ b, c $\gtrdot$ d, and b α c contains an error, then b α c can be replaced by any "locally correct" symbol, as a recovery action."

This is not necessarily so, as can be seen in the following example, taken from the language of DIM-statements in BASIC.

Parsing the input string

$$\# \; A \; (\; 10 \;) \; , \; B \; (\; 10 \; (\; 9 \;) \; , \; C \; (\; 10 \;) \; \#$$

after scanning the second comma, the stack could be

$$\# \; ar \; , \lessdot v \; (\; 10 \; , \; (\; i \;) \gtrdot ,$$

and the metasymbol i is locally correct, since $, \doteq i \doteq ,$, but this reduction introduces a phrase error. All depends on what is understood by "locally correct". The problem is that b α c is not necessarily a wrong version of a phrase. Nevertheless it must contain at least one simple phrase.

In the absence of a global analysis, Graham and Rhodes use an error correction cost function, which helps to choose "the best correction", when more than one is possible.

Also, their article [G2] constitutes a very good survey of error correction and recovery.

3.3.3. MINIMUM DISTANCE ERROR CORRECTING PARSERS

Some authors have tried to solve the general problem of error correction in a context-free language. A distance between two strings is defined as the minimum number of edit operations (substitutions, insertions, or deletions), necessary to change one string into the other. Peterson [P1], Aho and Peterson [A1], and Thomason et al [T1, T2, T3], find an extended grammar that generates not only

the valid sentences, but all the other "correctable" input strings.

Aho and Peterson generate the extended grammar by adding new error productions to the original grammar. Then a modified Earley's algorithm is used to parse the input string in the new language, and to count the number of error productions used in the syntax tree. Thomason defines three operators (corresponding to change operation, delete operation, and insert operation) that act over the productions of the original grammar, to obtain the productions of the extended grammar. Although they give a nice theoretical solution to error correction, it is not so practical to implement it. The number of productions in the extended grammar is much bigger than in the original one, and the time and the memory space required for parsing are considerable.

Lyon [L10] tries to overcome this impediment and develops an error correcting parser based on Earley's recognizer, with a modified SCAN and COMPLETER (see section 2.3.) to recognise errors. Also, at each step in the input string, the list L(i) is inicialliized by adding for each production π_k the state (k,1,i,0). Thus the number of states in each list L(i) is very large, certainly bigger than the number of productions in the grammar. If we remember that there are as many lists L(i) as input characters, the total number of atates is very large, and checking all these states slows the speed of the parsing process. (The last component of a state represents the number of errors). Nevertheless, since Earley's parsing algorithm is an efficient one, this method may be implemented.

CHAPTER IV

SOME ERROR CORRECTING PARSERS .

This chapter presents four different methods of error correction in some classes of context free languages.

The first section presents a transducer as an error correction mechanism. This transducer recognises the sentences of the language and those strings which are considered to be correctable. Such a method gives only a limited capability to correct anticipated errors.

Second section presents an error correcting algorithm for a regular language, due to Wagner [W1].

The main results of this chapter, and of the thesis, can be found in third and fourth sections.

In the third section is presented a global error correcting parser for a simple precedence language. This parser extends the idea developed by Graham and Rhodes [G2], of condensing the surrounding context of the error. We transform it into a global error correcting parser, by scanning the whole input string, and condensing it before the correction is made.

Section 4.4. presents a global top-down error correcting parser, for the entire class of context free languages. It starts from the global top-down parser described by Unger [U2], and transforms it into an error correcting parser. The efficiency of this parser can be improved by combining it with more efficient error correcting parsers, for particular subclasses of languages, such as regular languages , and simple precedence languages. For this reason, the error correcting algorithm for a regular language due to Wagner [W1], was presented in the second section.

All these methods are used in chapter five to design a global error correcting parser for BASIC.

4.1. ERROR CORRECTION BY A TRANSDUCER

We have seen in the second chapter that for each regular (context free) language there exists a finite state (pushdown) acceptor. Also the notion of a transducer has been defined there.

We say that the transducer

$$M_t = (K , T , \Gamma , \Delta , \delta_t , z_o , s_o , F)$$

is associated with the acceptor $M = (K , T , \Gamma , \delta , z_o , s_o , F)$ of the language L if the transition from one state to another for a given input $\underline{a}$ is the same for both machines. The difference between them is the extra output capability of the transducer. Formally this means that if $(q , r, o) \in \delta_t (s , a , z)$ then $(q , r) \in \delta(s , a , z)$.

But the extra output facility of the transducer can be used, for example, to parse the sentences in language L accepted by the acceptor M. (Ollongren [01]). We will use this facility for error correction.

Let $M_e = (K_e , T , \Gamma , \delta_e , z_o , s_o , F)$ be an extension of the acceptor M, in the sense that $K \subset K_e$ and $\delta_{e/\ K \times T \times \Gamma} = \delta$. In other words, M_e is obtained from M by adding new states or new transitions. Certainly the language L is embedded in the language L_e accepted by M_e. We can look upon sentence $w_e \in L_e$ as an incorrect version of a sentence $w \in L$. If the associated transducer M_t with the extended acceptor M_e outputs, for each string $w_e \in L_e$, a sentence $w \in L$ it can be viewed as an error-correcting mechanism.

As an example let us consider the regular language accepted by the finite state machine with the transition diagram given in fig. 4.1.2. The states ◎ are final states. It can be seen that the language accepted by this finite state machine is the set of lists of real numbers separated by , (DATA-lists in BASIC).

We can define a transducer that accepts any string and outputs a corresponding list of real numbers. The assumption we make is that all other characters, except those of the set {, , d , + , - , . , E }, are

incidentally inserted or have changed a comma. We will define this transducer starting from the extended automaton with the transition table given in fig. 4.1.2. There the ununderlined transitions correspond to the transitions of the finite state machine given in fig. 4.1.1.

To each transition there corresponds an output (which can be empty). The output function o: $K \times T \longrightarrow \Delta$, is given in fig. 4.1.3. In fact our transducer performs a function t that associates to each input string $\alpha \in T^*$ a sentence β in our language, i.e.

$$t : T^* \longrightarrow L .$$

The set of 'outputs' is $\Delta = \{ o_1 , o_2 , o_3 , \ldots o_7 \}$.

These outputs are composed from some more elementary actions, that must be performed to obtain the sequence $\beta = t (\alpha) = b_1 \; b_2 \ldots b_m$. These elementary actions are:

a_1 : print error message

a_2 : j : = j + 1

a_3 : b_j : = a_i

a_4 : b_j : = ' , '

a_5 : b_j : = a_{i-1}

a_6 : b_j : = o

a_7 : j : = j - 1 ,

and the outputs are sequences of ordered actions:

$o_1 = (a_2 , a_3)$

$o_2 = (a_1 , a_3)$

$o_3 = (a_1 , a_2 , a_3 , a_2 , a_5)$

$o_4 = (a_1 , a_6)$

$o_5 = (a_2 , a_4)$

$o_6 = (a_1 , a_7)$

$o_7 = (a_1)$.

By this transducer the sequence

981 / , , 675 , - - 98 , 986 N . 45~~//~~

will be transformed into the sentence:

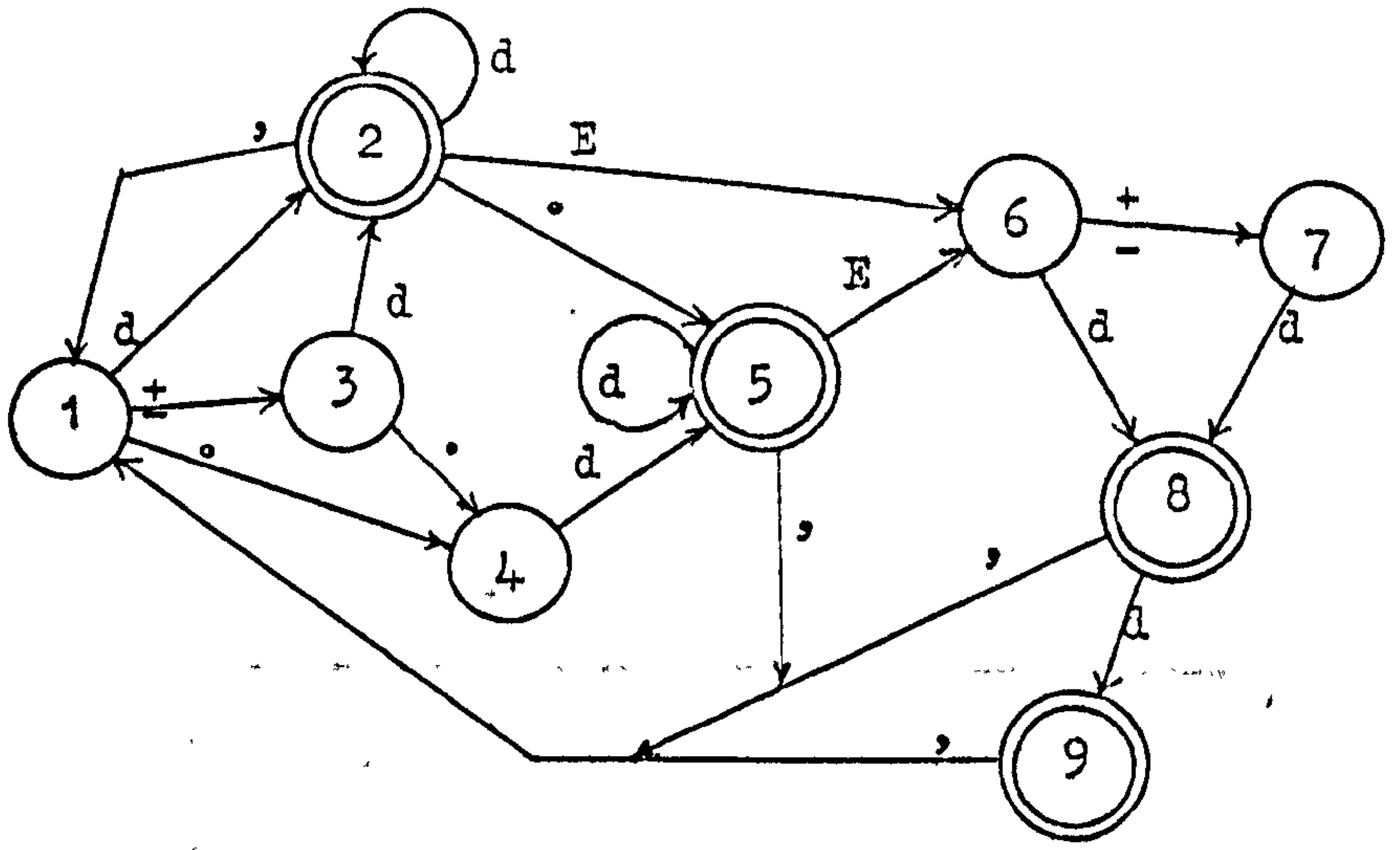


Fig. 4.1.1.

State \ Input	d	±	.	,	E	?
1	2	3	4	1	1	1
2	2	1	5	1	6	1
3	2	3	4	3	1	1
4	5	4	4	1	4	1
5	5	3	1	1	6	1
6	8	7	7	1	7	7
7	8	8	8	1	8	7
8	9	1	1	1	1	8
9	1	1	1	1	1	1

Fig. 4.1.2.

d	±	∘	,	E	?
1	o_1	o_1	o_1	o_7	o_7	o_7
2	o_1	o_5	o_1	o_5	o_1	o_5
3	o_1	o_2	o_1	o_7	o_6	o_6
4	o_1	o_3	o_7	o_6	o_7	o_5
5	o_1	o_8	o_5	o_5	o_1	o_5
6	o_1	o_1	o_7	o_5	o_7	o_7
7	o_1	o_7	o_7	o_4	o_7	o_7
8	o_1	o_5	o_5	o_5	o_5	o_5
9	o_5	o_5	o_5	o_5	o_5	o_5

Fig. 4.1.3.

981 , 675 , - 98 , 986 , . 45 =// .

The idea of using a transducer for error correction is not new. For any deterministic context free language there exists a deterministic push-down automaton that accepts this language, and parses it. But when one or more errors are encountered, the transducer enters the 'dead' state. In early compilers the right portion of the input string was scanned and neglected, until an unambiguous delimiter (as the end of a statement, a comma in a list, or an arithmetic operator +, -,*) was encountered. This implies remaining in the 'dead' state until a delimiter is met, that permits one to decide to which state to jump.

Although the idea of using a transducer can be useful in some situations it has some shortcomings. First there is the fact that only the left context is used to correct these errors and the errors are corrected always in the same way. Second it can be used only for a small language, where an exhaustivedecision about each error can be made. For a larger language, only the predicted errors will be corrected. What is worse here is the fact that the surrounding (right) context of the error is not used for error correction. It uses only the information available from the left context. This information reflects itself in the current state of the automaton.

4.2. ERROR CORRECTION IN A REGULAR LANGUAGE

A general solution to error correction in a regular language has been given by Wagner [W1].

Let L be a regular language and let $M = (K , T , \delta , s_0 , F)$ be the finite state machine that accepts L. For an arbitrary input string

$$w = a_1 a_2 \dots a_n ,$$

in T^*, the problem is to find the nearest sentence in the language.

Let $F (j , q)$ be the minimum number of edit operations needed to change the substring $a_1 a_2 a_3 \dots a_j$ into a string u_j that brings the finite state machine from the initial state s_0 into the state q. By convention

$$F (o , s) = \begin{cases} o , & \text{for } s = s_0 \\ \infty , & \text{for } s \neq s_0 \end{cases} .$$

Let $u = b_1 b_2 \dots b_m$ be the nearest sentence in L, i.e. $d(u , w)$ is minimum. Since any sentence in the language brings the finite state acceptor from the initial state s_0 into a final state q, we have

$$d (u , w) = \min_{q \in F} F (n , q) . \qquad (4.2.1)$$

Therefore the minimum number of corrections*), $er = d (u , w)$, needed to change the string w into a sentence in L, can be computed if $F(n,q)$ are known for all states $q \in F$.

These numbers can be computed by an iterative process. For each pair of states (s_1 , s_2) and for each input symbol $\underline{a}$, let $v (s_1 , s_2 , a)$ be the minimum number of edit operations needed to change the symbol $\underline{a}$ into a string $v(s_1 , s_2 , a)$, which brings the finite state machine from the state s_1 into the state s_2. The following formula holds [W1]:

---

*For simplicity it is assumed here that each edit operation e has the weight $u (e) = 1$.

$$F(j+1, s) = \min_{s_1 \in K} F(j, s_1) + V(s_1, s, a_{j+1}) \qquad 4.2.2$$

Thus, executing this algorithm, only the minimum number of errors needed to change the input string into a correct sentence, is known. To find the correct sentence we must go backward to see how $d(u, w)$ has been computed, i.e. the sequence of states $(q_1, q_2, \ldots, q_n)$ is found as follows:

a) there exists a final state q_n such that $d(u, w) = F(n, q_n)$

b) for any $j < n$, there exists a state q_j such that

$$F(j+1, q_{j+1}) = F(j, q_j) + V(q_j, q_{j+1}, a_{j+1}).$$

The correct sentence u can be found from the fact that it is the shortest string that brings the finite state machine through the states

$$q_1, q_2, \ldots, q_n.$$

At first glance, this seems to be an excellent solution to error correction in a regular language. And it is, for an acceptor with a small number of states and input vocabulary. But when the number of states grows up and the alphabet consists of 26 letters, 10 numerals and some more special characters it seems to become impractical. For a finite state machine with n states and m inputs there must be stored $n^2 m$ numbers $V(s_1, s_2, a)$. A finite state machine with 100 states and 52 input symbols would require a storage of 520K words just for the data. Nevertheless we find this algorithm very useful when it is combined with other methods of error correction, as it will be done later.

4.3. A SIMPLE PRECEDENCE ERROR CORRECTING PARSER

A parser for a simple precedence language has been presented in 2.5. That parser can detect an error in two different ways.

First, when no relation holds between the top stack symbol and the incoming input symbol. Then one of these two symbols is in error, or some symbols have been deleted.

Second, when a potential simple phrase α is detected, but α is not a right hand side of any production.

What strategy must be followed for correcting these errors? Two possible ones are given by Graham and Rhodes [G2] and Leinius [L6]. They consist mainly in trying to make corrections immediately the error has been detected, or to condense first the left and right context of the error, before making a decision. In this thesis this concept is taken a little further by trying to analyse the context of the entire string, to make decisions about the nature of the errors. But, before going into details, some points can be made about the correction of these errors.

In the first case, the top stack symbol s, and the incoming input symbol a_k are not related. Two situations must be taken into account:

- first, some symbols between a_{k-1} and a_k have been deleted. Thus one or more symbols must be inserted. To insert a character $\underline{a}$ between s and a_k, there must be a relation between s and a, and between $\underline{a}$ and a_k.

- second, one of these two symbols (a_{k-1} , a_k) is erroneous. Since s is already in the stack, it can be a metasymbol - the result of a reduction of a phrase containing more than one symbol. Thus the assumption that s is in error must be the last one. There are two cases that must be considered:

α) there is no relation between a_k and a_{k+1}, which combined with the fact that no relation exists between s and a_k leads to the conclusion that a_k has been inserted.

β) a_k and a_{k+1} are related. This can be a suggestion that a_k is correct. In this case it is probable that one or more symbols between s and a_k are missing, or that s is incorrect.

These facts are true but they do not help us to take a decision on how to correct the error. We will bear these in mind when giving a solution to error correction.

When a potential simple phrase α is detected, but α is not a right hand side for any production, it is said [G2, L5] that a phrase error has occurred. Such an error is seen, for example, in analysing the string $\beta_1 \alpha c_1 \; c \; c_2 \; r \beta_2$ in the simple precedence grammar G, that contains the productions

$$\pi_i : \; A_i \longrightarrow \alpha \, c_1 \, c \, \beta$$
$$\pi_j : \; A_j \longrightarrow \eta \, c \, c_2 \, r \, ,$$

but there is no production of the form:

$$\pi : \; A \longrightarrow \alpha \, c_1 \, c \, c_2 \, r \, .$$

In this grammar the substring $\alpha c_1 \; c \; c_2 \; r$ is considered a simple phrase. But it is not, since $\alpha c_1 \; c \; c_2 \; r$ is not a right hand side of any production.

As an example, let us consider the grammar G, which has the productions

$$\pi_1 : \quad \underline{\text{list}} \longrightarrow \underline{\text{ar}} \, [\, , \underline{\text{ar}} \,]$$
$$\pi_2 : \quad \underline{\text{ar}} \longrightarrow \text{v} \, (\, \text{i} \, , \, \text{i} \,)$$
$$\pi_3 : \quad \underline{\text{ar}} \longrightarrow \text{v} \, (\, \text{i} \,)$$
$$\pi_4 : \quad \text{i} \longrightarrow \text{d} \, [\text{d} \,]$$
$$\pi_5 : \quad \text{v} \longrightarrow \text{l d}$$
$$\pi_6 : \quad \text{v} \longrightarrow \text{l} \quad . \qquad (4.3.1)$$

Here l stands for letter (A , B , C , ... , Y , Z), d stands for digit (0 , 1 , 2 , ... , 8 , 9), and the brackets [ ] are used to replace a recursion, i.e. [u] means that the string u can be repeated any number of times. The language generated by G is the set of all lists in a DIM statement of the BASIC language. The string

w = ## A (94 , B (10 , 2) , V (10) ## will be reduced to

≠≠ v (i , ar , ar ≠≠ as can be seen in fig. 4.3.2. Here v (i , ar , ar is detected as a potential simple phrase, but it is not a phrase in our language.

This example reveals a very important fact. A simple precedence parser produces the canonical derivation of the analysed string. This is not true if the string contains an error. In the example shown in fig. 4.3.2, the substring B (1 0 , 2) is reduced to ar before A (9 4 is corrected, to yield itself a declared array (coded by the metasymbol ar).

It also reveals that the detection point of the error can be very far away from the real error.

	list	ar	i	v	l	d	(	,	)	≠≠	?
list										=	
ar								=		>	
i								=	=	>	
v							=				
l								=	>		
d								=	>	>	>
(							=		<		
,							=	=	<	<	<
)										>	>
≠≠	=	<		<	<	<	<				
?											

Fig. 4.3.1.

The precedence matrix for the grammar defined by the productions (4.3.1).

Stack String	Incoming input symbol	Rest of the string
#	A	(94 , B (10 , 2) , V (10) #
# A	(	94, B (10, 2) , V (10) #
# V	(	94 , B (10 , 2) , V (10) #
# V (	9	4 , B (10 , 2) , V (10) #
# V (9	4	, B (10 , 2) , V (10) #
# V (94	,	B (10 , 2) , V (10) #
# V (i	,	B (10 , 2) , V (10) #
# V (i ,	B	(10 , 2) , V (10) #
# V (i , B	(	10 , 2) , V (10) #
# V (i , V	(	10 , 2) , V (10) #
# V (i , V (	1	0 , 2) , V (10) #
# V (i , V (1	0	, 2) , V (10) #
# V (i , V (10	,	2) , V (10) #
# V (i , V (i	,	2) , V (10) #
# V (i , V (i ,	2	) , V (10) #
# V (i , V (i , 2	)	, V (10) #
# V (i , V (i , i	)	, V (10) #
# V (i , V (i , i)	,	V (10) #
# V (i , ar	,	V (10) #
# V (i , ar ,	V	(10) #
# V (i , ar , V	(	10) #
# V (i , ar , V	(	10) #
# V (i , ar , V (	1	0) #
# V (i , ar , V (1	0	) #
# V (i , ar , V (10	)	#
# V (i , ar , V (i)	#	
# V (i , ar , ar	#	

Fig. 4.3.2

Let us now describe the global error correcting parser in detail. The parser has to analyse the string:

$$w = a_0\, a_1\, a_2 \quad \cdots\cdot \quad a_n\, a_{n+1}\,,$$

where $a_0 = a_{n+1} =$ '#' are endmarkers. As in 2.5 a stack is used for storing the information acquired from the previously scanned string. The parsing process has two parts. A scanning one, in which the input characters are scanned and analysed, and the received information is stored in the stack. Then a reduction and recovery one, when the information stored in the stack is used for reducing the stack, by means of the reduction rules of the grammar, or for error correction and recovery.

During the scanning phase a pointer is needed in the input string. Let us say k is the integer such that the next input character is a_k. At the beginning of the scanning phase the delimiter $a_0 =$ # is put onto the stack, the pointer k is initialised, $k = 1$, and n_s is initialised to 1. Here n_s denotes the length of the stack string. $Stack_{n_s}$ is the topmost stack symbol.

If the topmost stack symbol $s = Stack_{n_s}$ and the incoming input character $i = a_k$ are not related, then the character i is put onto the stack, but a question mark '?' is inserted between s and i. An error message is given, and the pointer k is increased by one.

If s and i are related then

- if $s \lessdot i$, the beginning of a potential simple phrase is detected, the character '$\lfloor$' is put onto the stack to mark this beginning, and i is put onto the stack. The pointer k is increased by one.

- if $s \doteq i$, then i is still part of the potential simple phrase and i is put onto the stack. The pointer k is increased by one.

- if $s \gtrdot i$, the end of the potential simple phrase is detected, and '$\rfloor$' is put onto the stack to mark this end. At this stage the stack contains a potential simple phrase enclosed within the brackets $\lfloor\,\rfloor$.

The second phase must then start, i.e. a reduction procedure must be called. If no errors have occurred, then the string enclosed within the brackets '$\lceil$' and '$\rfloor$' is a simple phrase, and it must be reduced to the corresponding metasymbol. If phrase errors have occurred, they will be detected by this reduction procedure, since the potential simple phrase enclosed within the brackets '$\lceil$' and '$\rfloor$' does not match any right hand side of a production in the grammar.

If errors, marked by the presence of '?' , have been met in this potential phrase (now we are not sure if it is a simple phrase or a phrase), then two alternatives can be followed:

- first, a recovery procedure can be called at this step,

- second, since some more information can be obtained by analysing the input string further, i is put onto the stack, and the scanning phase continues.

This scanning phase of the parsing process is summarised in the flow-chart of the fig. 4.3.3. It finishes when the last input character a_{n+1} , the endmarker '$\#$' is met. At this stage the whole information available from the input string, is contained in the stack. The stack must be verified, and if needed, a recovery procedure must be followed.

At each step, the parsing process is completely described by the content of the stack, by the incoming input character i , and by the rest of the input string, i.e.

$$\text{state} = (\ \text{Stack} \mid\ i\ \mid\ a_{k+1}\, a_{k+2} \cdots\ a_n \#\ ) \qquad (4.3.2)$$

is the description of the state of the parsing process.

During the scanning phase, the parsing process may enter the state

$$(\ \# \ \eta_1 \times \lceil \eta \rfloor \mid\ i\ \mid\ a_{k+1}\, a_{k+2} \cdots\ a_n \ \#\ ) . \qquad (4.3.3)$$

This means that a potential simple phrase, or an erroneous phrase, has been detected. The most probable situation is the existence of a correct simple phrase, since the errors are not very likely. As can be seen in the flowchart of fig. 4.3.3, the procedure REDUCE is called at this state.

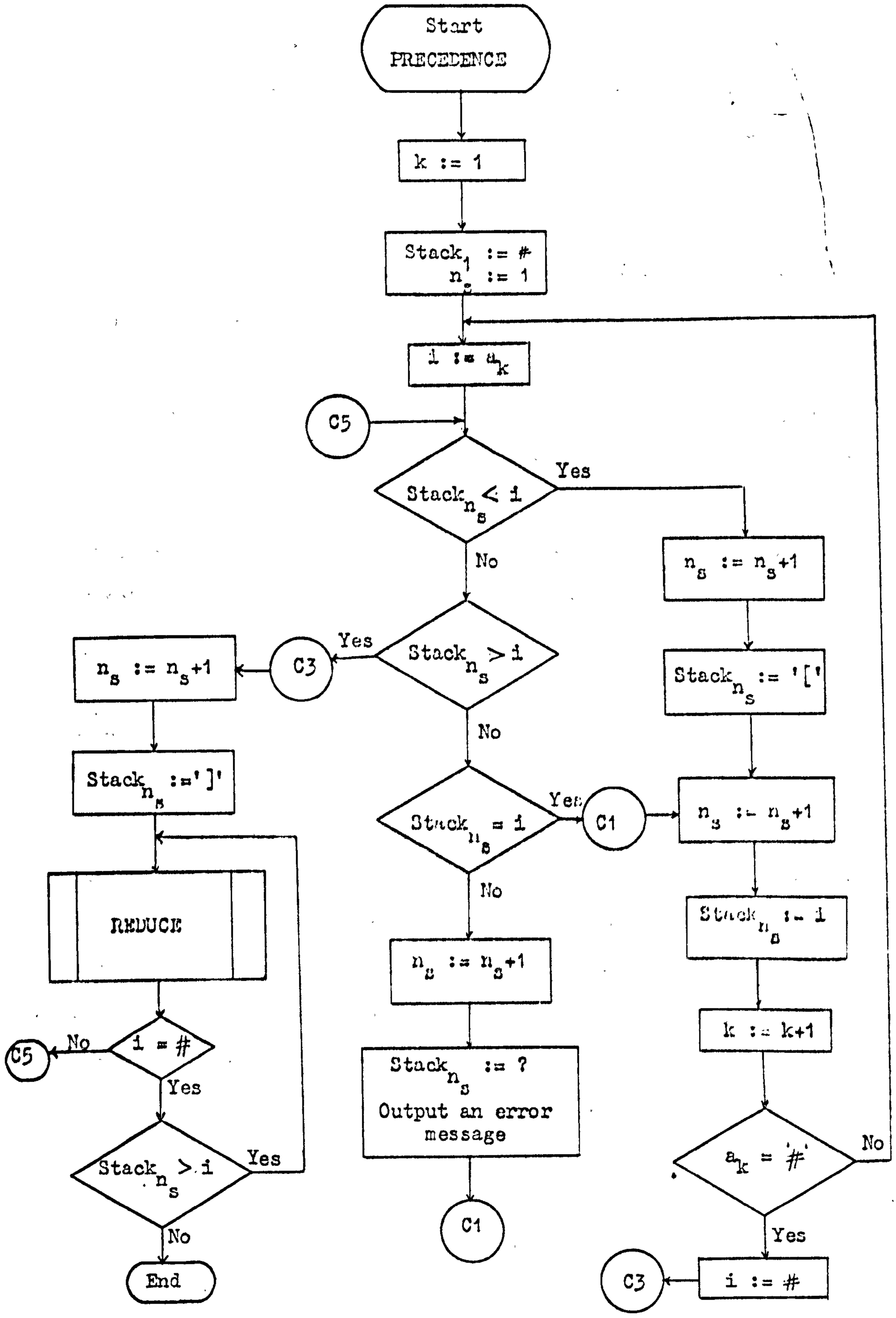

Start
PRECEDENCE
k := 1
Stack1 := #
ns := 1
i := ak
C5
Stackns < i
Yes
No
nS := nS+1
Stackns := '['
Stackns > i
Yes
C3
nS := nS+1
No
Stackns = i
Yes
C1
nS := nS+1
No
nS := nS+1
Stackns := ']'
REDUCE
Stackns := ?
Output an error message
C1
Stackns := i
k := k+1
ak = '#'
No
Yes
i = #
No
C5
Yes
Stackns > i
Yes
End
C3
i := #

Fig. 4.3.3

The procedure REDUCE analyses the potential simple phrase η . If no question marks are found in the string η , no errors have been met during the scanning phase, and the potential simple phrase is reduced. This means that the productions of the grammar are checked, a production $A \longrightarrow \eta$ is retained, and the corresponding reduction rule, $\eta \vdash A$, is used to replace the simple phrase η by the metasymbol A . If there is not a production $A \longrightarrow \eta$, then a phrase error has been detected and must be reported. A procedure PHRASE is called to correct this phrase error.

The procedure REDUCE is given in the flowchart of fig. 4.3.4.

If in the string η there are some errors marked by the presence of a '?' , then a call to the procedure RECOVER is made. The errors are corrected at this stage if no ambiguous situations arise. Otherwise the scanning phase may continue to take more information from the unscanned string.

The procedure RECOVER tries to reduce the string η . It is already known that η is erroneous, since '?' has been met in this string. In other words η has the form

$$\eta = \beta_1 ? \beta_2 ? \dots ? \beta_{mk} .$$

Here β_1 must have a common prefix with a right hand side of a production, and β_{mk} must be a suffix of a right hand side of a production. The string β_2 can be a part of a simple phrase, together with β_1 , but with an occurred error between them. Also β_2 can be a simple phrase itself with erroneous first or last characters.

It is known that the relation between two consecutive characters of β_j is $\doteq$, and the same relation exists between two consecutive symbols in the right hand side of a production. Therefore all β_j must be substrings in at least one right hand side of a production. If there is no production

$$A \longrightarrow \alpha_1 \beta_j \alpha_2 \qquad (4.3.4)$$

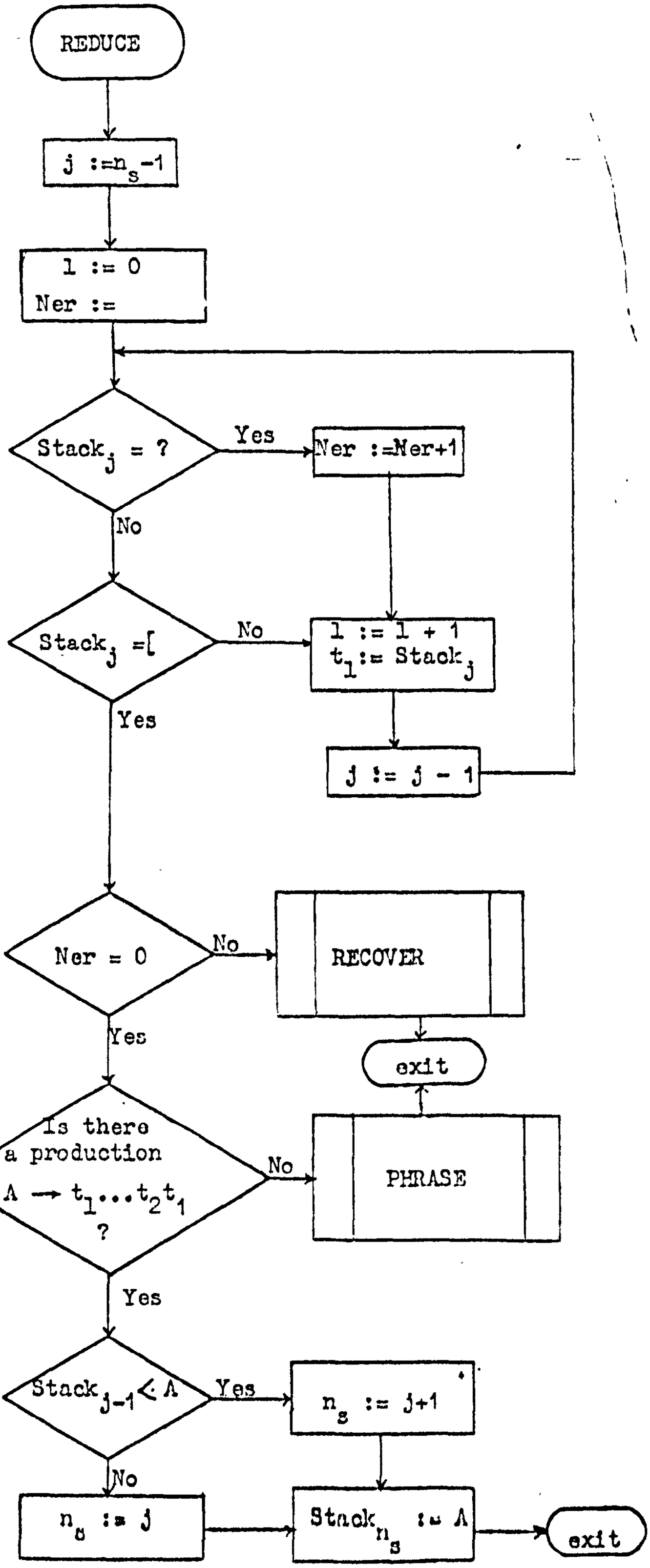

REDUCE
j :=n_s -1
l := 0
Ner :=
Stack_j = ?
Yes
Ner :=Ner+1
No
Stack_j =[
No
l := l + 1
t_l := Stack_j
Yes
j := j - 1
Ner = 0
No
RECOVER
Yes
exit
Is there
a production
A → t_1...t_2t_1
?
No
PHRASE
Yes
Stack_j-1 ⋖ A
Yes
n_s := j+1
No
n_s := j
Stack_n_s := A
exit

Fig. 4.3.4

for some strings α_1 , α_2 (possible empty strings), we still say that a phrase error has occurred (inside β_j). This error is discovered when one does not find a production (4.3.4), for the string β_j . One of the following alternatives must hold:

- either β_j is a substring of a right hand side of a production (4.3.4), incomplete one, since some characters are missing at the beginning or at the end of the simple phrase (α_1 or α_2).

- or β_j is a simple phrase, and the '?' is due to errors at its left and right side,

- or β_j is not a substring of any right hand side of a production. Here a new error, a phrase error, is discovered.

- or the whole string β_j has been inserted. This is more likely for a shorter string β_j , and less probable for a longer string β_j . If the string β_j contains only one character a , i.e. ? a ? is part of η , the probability of insertion, or of error in general, is high.

It follows that all productions (4.3.4) are candidates to be used in the reduction of β_j , and must be stored for a later decision. These decisions will be made after all substrings β_j have been analysed. Thus, for each β_j a resolution set Res (β_j) defined by

$$\text{Res}(\beta_j) = \{(A, s) \mid \text{the production } \pi_s \text{ is } A \longrightarrow \alpha_1 \beta_j \alpha_2\} \qquad (4.3.5)$$

is computed and stored.

If Res (β_j) contains only one pair (metasymbol, production), this production is the only one that can be used to reduce the string β_j. If $|\text{Res}(\beta_j)| > 1$, (here $|P|$ denotes the number of elements in the set P), then Res (β_{j-1}), Res (β_{j+1}) , ... , can help to decide which reduction rule to use.

The start of the procedure RECOVER - the decomposition of η , is described in fig. 4.3.5. It is continued by PREFIX (fig. 4.3.6), SUFFIX (fig. 4.3.7) and SUBSTRING (fig. 4.3.8), which compute the resolution sets Res (β_1) , Res (β_{mk}), and Res (β_j) respectively.

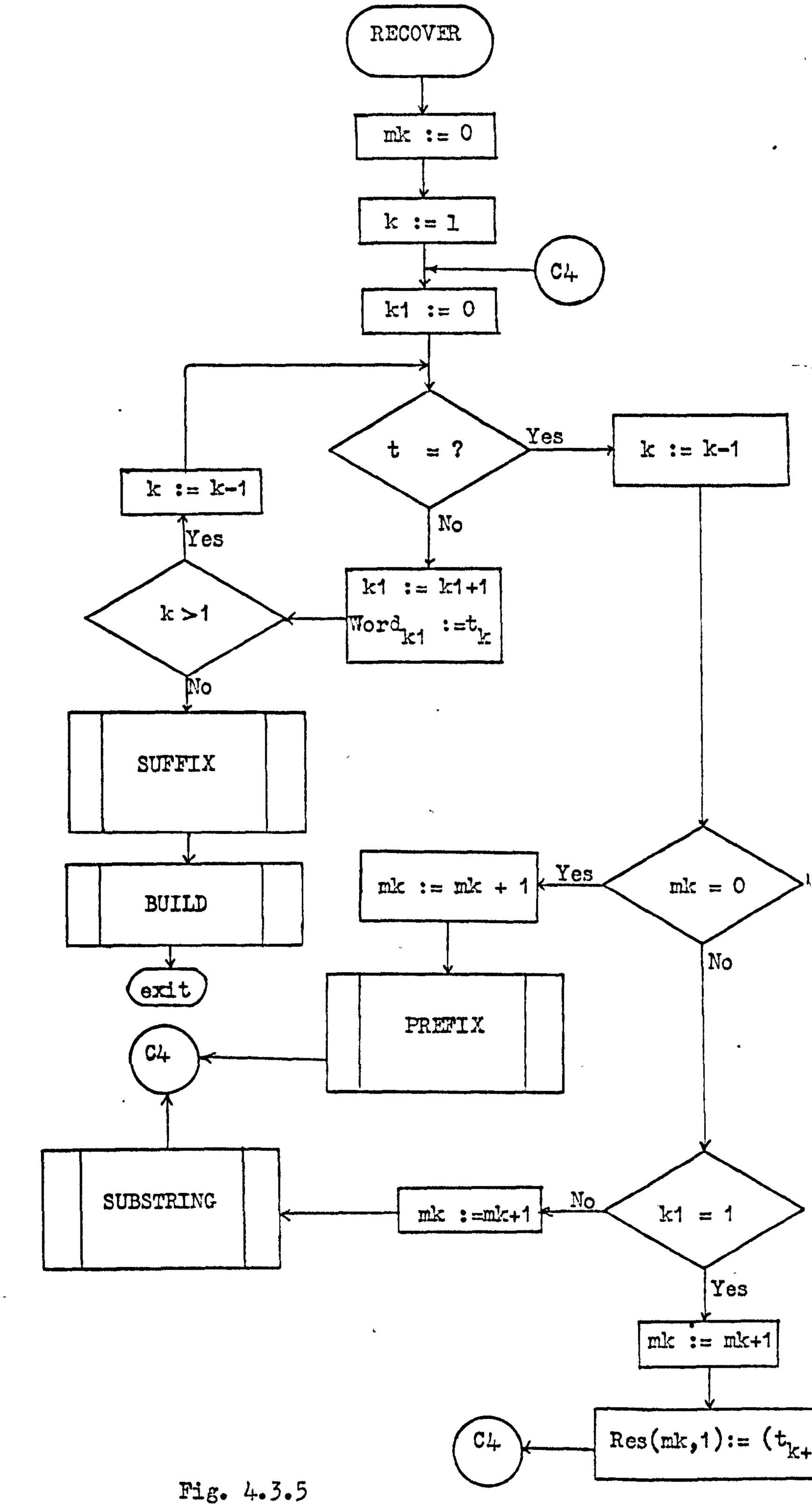

RECOVER
mk := 0
k := 1
C4
k1 := 0
t = ?
Yes
k := k-1
No
k := k-1
Yes
k1 := k1+1
Word_k1 := t_k
k > 1
No
SUFFIX
BUILD
exit
mk := mk + 1
Yes
mk = 0
No
PREFIX
C4
SUBSTRING
mk := mk+1
No
k1 = 1
Yes
mk := mk+1
C4
Res(mk,1):= (t_k+1,0)

Fig. 4.3.5

The productions of the grammar are;

$\Pi_1, \Pi_2, \ldots, \Pi_{n_p}$.

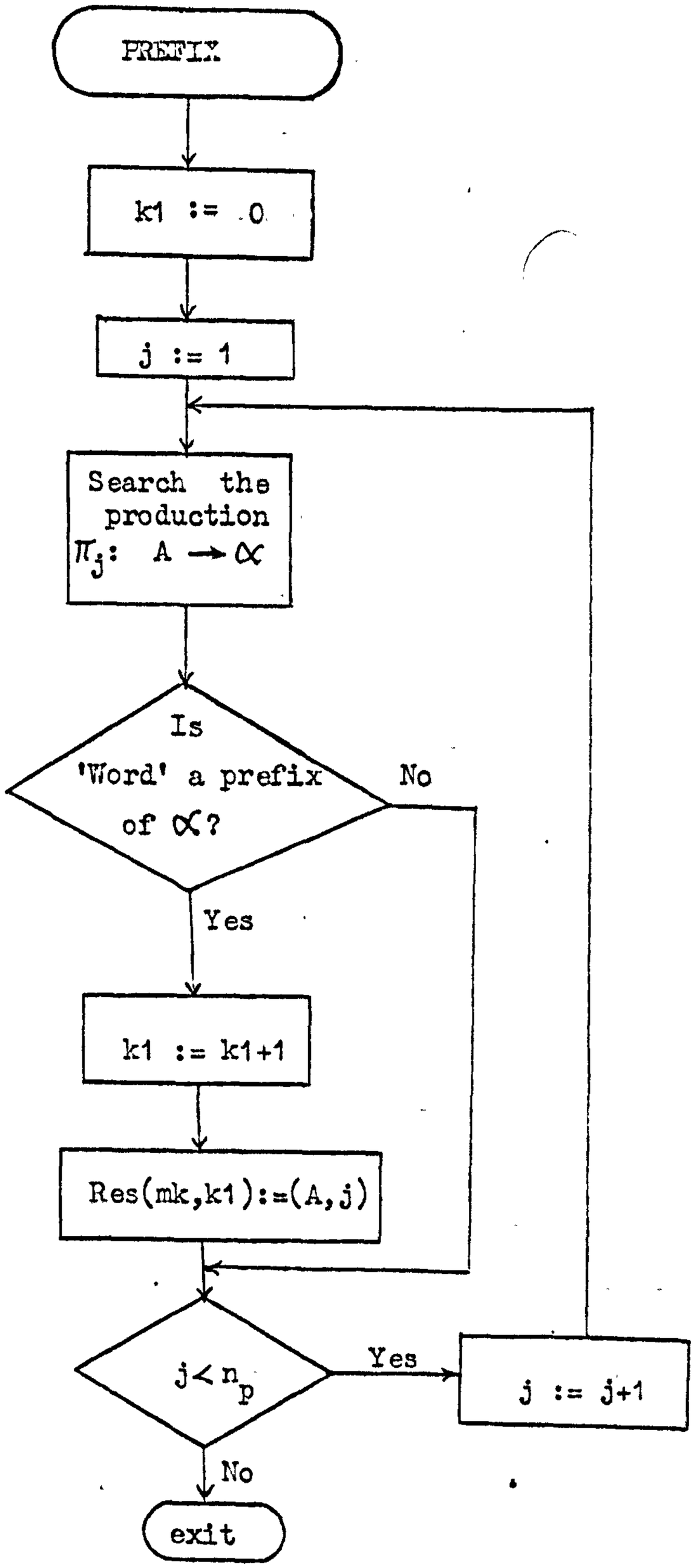


Fig. 4.3.6

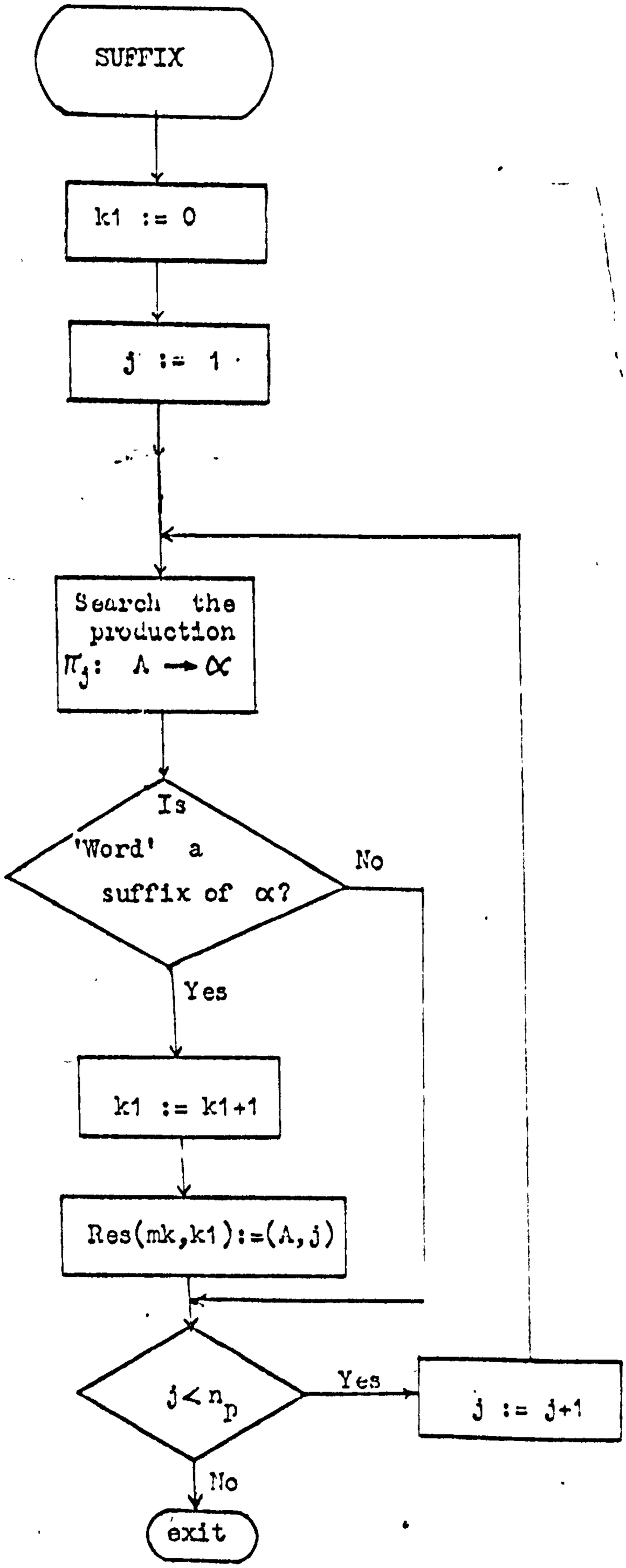

SUFFIX
k1 := 0
j := 1
Search the production π_j: $A \rightarrow \alpha$
Is 'Word' a suffix of α?
No
Yes
k1 := k1+1
Res(mk,k1):=(A, j)
$j < n_p$
Yes
j := j+1
No
exit

Fig. 4.3.7

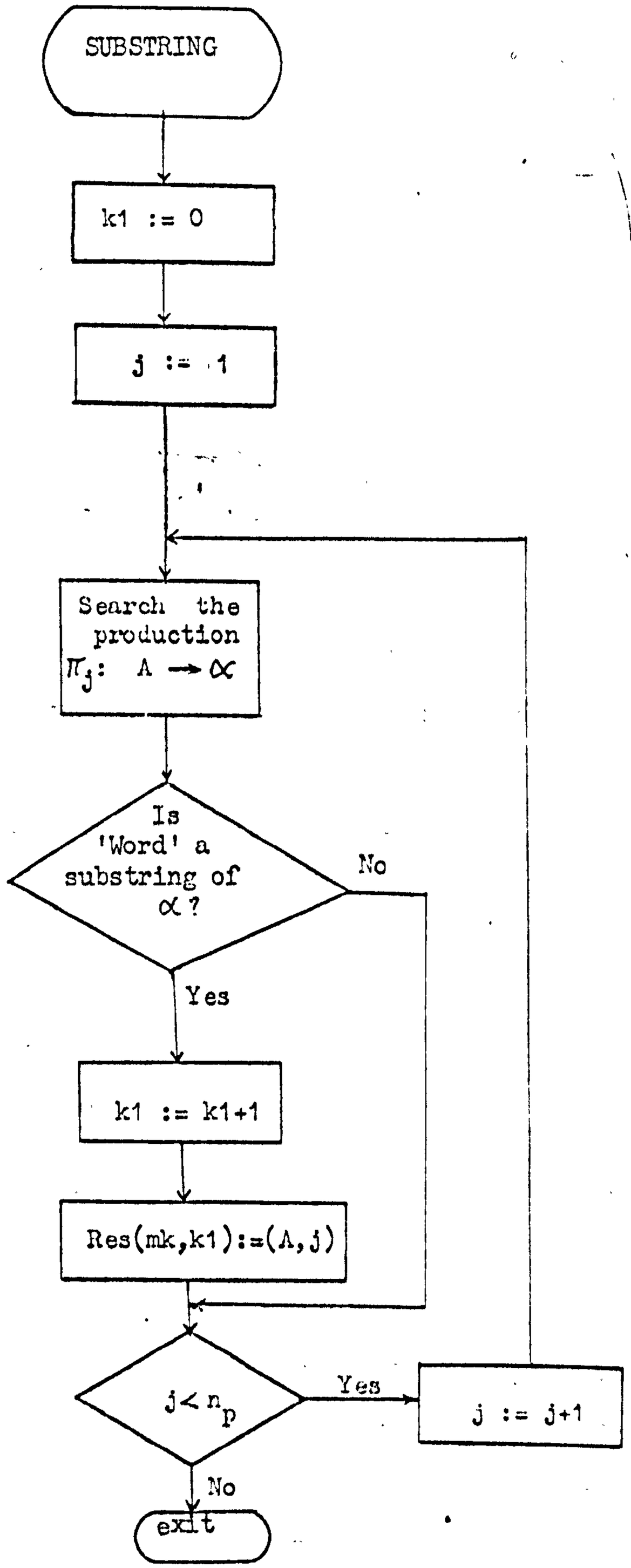

SUBSTRING
k1 := 0
j := 1
Search the production π_j: A → α
Is 'Word' a substring of α?
No
Yes
k1 := k1+1
Res(mk,k1):=(A,j)
j< n_p
Yes
j := j+1
No
exit

Fig. 4.3.8

Before proceeding further, let us analyse some examples. Consider again the grammar defined by the productions (4.3.1), and let us try to correct the input strings

w_1 = ## A) 5 (9 , 6) , B (10) ## ,

w_2 = ## A) 9 , 6) , B (10) ## ,

w_3 = ## A (9 , 6) B (10) ## ,

and

w_4 = ## A (9 , 6 B (10) ## .

For anyone familiar with BASIC, it is clear that an inserted ') ' is the least error in the string w_1 , a change of ' (' to ') ' is the error in w_2 , a missing ' , ' is the error in w_3 , and missing ' , ' and ') ' are the errors in w_4 . The RECOVER procedure is called at a stage of the parsing process described in fig. 4.3.9 . This stage is described by a state s , that has three components: the content of the stack, the incoming input character, and the rest of the string not yet scanned.

State	Stack string	,	a_k	,	rest of the string
a1)	w_1 : ## ⌊ 1 ?) ? d ⌋		(		9 , 6) , B (10) ##
a2)	w_2 : ## ⌊ 1 ?) ? d ⌋		,		6 , B (10) ##
b)	w_3 : ## ⌊ v (i , i) ? ⌋		(		10) ##
c)	w_4 : ## ⌊ v (i , ⌊ d ? ⌋		(		10) ##

Fig. 4.3.9

The contents of the stack is the same in both cases a1) and a2). The character) is enclosed within the question marks '?' . It is possible that) is inserted, and the fact that 1 is related to d (1 $\doteq$ d , see fig. 4.3.1), strengthens this assumption. In this hypothesis the reduction 1d ⊢— v , derived from the production π_5 , can be made. Also the fact that ') ' can be the result of a change operation (a , ')') must be taken into account. The only possible character is '(' , since 1 $\gtrdot$ '(' $\lessdot$ d . In this case 1 must be reduced to v since 1 $\gtrdot$ '('.

How can these results be obtained using the resolution sets? In this case $\eta = \beta_1 \,?\, \beta_2 \,?\, \beta_3 = 1' \,?\,) \,?\, d$ and

$$\text{Res}(\beta_1) = \{(v, 5), (v, 6)\}$$

$$\text{Res}(\beta_2) = \{(\,')\,', 0)\}, \text{ since } ')' \text{ is enclosed within } '?',$$

$$\text{Res}(\beta_3) = \{(i, 4), (v, 5)\}.$$

Since β_2 is very unreliable the first and third resolution sets are used. They contain a common production $\pi_5 : v \rightarrow 1d$, and $1d = \beta_1 \beta_3$ corresponds to an insertion of β_2. Also Res (β_1) contains the production $\pi_6 : v \rightarrow 1$, and Res (β_3) contains the production $\pi_4 : i \longrightarrow d$, and the reductions $1 \vdash\!\!\!- v$, and $d \vdash\!\!\!- i$ can be made. This case corresponds to the case of a change operation. Since $v \,?\, i$ appears in the production $\pi_3 : \underline{ar} \longrightarrow v\,(\,i\,)$, this change operation can be ('(' , β_2). Thereby two reductions are possible for the stack, and the stack could be

$$\# \lfloor v$$

$$\# \lfloor v\,(\,i \quad .$$

To choose between them we need some more (right) context. In both cases, a1) and a2), the incoming input characters a_k is enough, since only one of the possible topmost stack symbols, v or i, are related to a_k. In the case a1) the relation $v \doteq ($ holds, but i is not related to '(', thus the first reduction must be chosen. In the second case a2), v is not related to $a_k = ')'$, but the relation $i \doteq \lrcorner$ holds, thus the second reduction must be chosen. After this correction the parsing process enters the state

	Stack	a_k	Rest of string
a1)	$\# \lfloor v\,($	9	, 6) , B (10) #
a2)	$\# \lfloor v\,(\,i\,,$	6	) ; B (10) #

In the case b) the strings $\beta_1 = v\,(\,i\,,\,i\,)$ and $\beta_2 = 1$, are separated by '?'. To these there corresponds the following resolution sets:

$$\text{Res}\,(\beta_1) = \{(\underline{ar}, 2)\},$$
$$\text{Res}\,(\beta_2) = \{(v, 6)\}.$$

In both cases only one reduction is possible. Furthermore, no changes are needed in the strings β_1 and β_2, to transform them into simple phrases, since

$\beta_1 = v(i, i) \vdash \underline{ar}$, using the production π_2,

and

$\beta_2 = i \vdash v$, using the production π_6.

Thereby these reduction rules must be used to reduce the stack to $\# \lceil \underline{ar} ? v$. The question mark '?' between β_1 and β_2 is due to some missing chracters. How does one find them? There are two alternatives. First, we try to correct this error at this stage by inserting a character 'a' related both to $\underline{ar}$ and v. It happens that only one character, ',' , satisfies these conditions. But there can be more than one such character, and we must use the second alternative in this case.

Second, since v is related with the incoming input character '(' , the scanning phase may continue, and more right context become available. Here is the continuation of the parsing process.

State: (Stack string	a_k	Rest of the input string
$\# \lceil \underline{ar} ? v ($	1	$0) \#$
$\# \lceil \underline{ar} ? v (\lceil 1$	0	$) \#$
$\# \lceil \underline{ar} ? v (\lceil 10 \rfloor$	)	$\#$
$\# \lceil \underline{ar} ? v (1) \rfloor$	# .	

At this stage the procedure RECOVER must be called again. To the substrings $\beta_1 = \underline{ar}$ and $\beta_2 = v(i)$, there corresponds the resolution sets

$$\text{Res}\,(\beta_1) = \{(\underline{list}, 1)\},$$
$$\text{Res}\,(\beta_2) = \{(\underline{ar}, 3)\}.$$

Each resolution set contains only one production. While in the second set the whole right hand side of the production $\pi_3 : \underline{ar} \longrightarrow v(i)$

is matched by β_2 , it is not the same for the string $\beta_1 = \underline{ar}$.

Here the production

$$\pi_1 : \underline{list} \longrightarrow \underline{ar}\,[\,,\ ar\,]$$

contains an infinite number of reduction rules, and two of them may be used here:

$$\beta_1 = \underline{ar} \vdash\!\!- \quad \underline{list} \ ,$$

and

$$\underline{ar}\ ,\ \underline{ar} \vdash\!\!- \quad \underline{list} \ .$$

If the reduction $\underline{ar} \vdash\!\!- \underline{list}$ is chosen, the stack would be $\#\,[\,\underline{list}\ ?\ \underline{ar}\,]\,\#$ and the stack cannot be reduced any more. In the second case, we need a second $\underline{ar}$ to complete the reduction rule $\underline{ar}\ ,\ \underline{ar} \vdash\!\!- \underline{list}$, which fits exactly in this case, since $\beta_2 \vdash\!\!- \underline{ar}$. Clearly the '?' must be replaced by , .

Formally, the first resolution set contains the production

$$\pi : A \longrightarrow \beta_1\ u\ B\ v \ .$$

The string β_1 is not long enough to match the right hand side of this production, but there is the possibility that β_2 can be reduced to B and the terminal string u has been deleted.

In the fourth case β_1 = 'd' , and β_2 = '1' , are composed of only one character, but none of them are enclosed within '?' marks. The resolution sets are:

$$\beta_1 = d \implies \mathrm{Res}\,(\beta_1) = \{(\,i\ ,\ 4\,)\ ,\ (\,v\ ,\ 5\,)\} \ ,$$
$$\beta_2 = 1 \implies \mathrm{Res}\,(\beta_2) = \{(\,v\ ,\ 6\,)\}.$$

The second set contains only one reduction rule $\beta_2 = 1 \vdash\!\!- v$. Since v is related to the incoming input character '(' , this rule may be used. To reduce the string β_1 there are two choices. The first is to reduce d to i , and Matr (',' ,i) $\neq$ 0 , i.e. the result of the reduction is related to the left context. The result 'v' of the second reduction rule $i\,d \vdash\!\!- v$ is not related to the left character ',' , and the string β_1 = d does not match the entire right hand side of the production

$$\pi_5 : v \text{ —— } i\,d$$

Since only one reduction remains in both cases, they must be used and the state of the parsing process will be

$$\# \lceil v\,(\,i\,,\,i\,?\,v \quad | \quad (\quad | \quad 10) \#$$

The parsing process continues until it enters the s tate

$$\# \lceil v\,(\,i\,,\,i\,?\,v\,(\,i\,)\,\rfloor \quad | \quad \# \quad | \quad \varepsilon\,.$$

At this stage the procedure RECOVER is entered again, the following resolution sets are found

$$\beta_1 = v\,(\,i\,,\,i \Longrightarrow \text{Res}\,(\beta_1) = \{(\,\underline{ar}\,,\,2\,)\},$$

$$\beta_2 = v\,(\,i\,) \Longrightarrow \text{Res}\,(\beta_2) = \{(\,\underline{ar}\,,\,3\,)\},$$

and the stack must be reduced to $\# \lceil \underline{ar}\,?\,\underline{ar}\,\rfloor$. At this stage both resolution sets will contain the same production rule

$$\underline{list} \longrightarrow \underline{ar}\,\lceil\,,\,ar\,\rfloor\,.$$

The reduction rule $\underline{ar}\,,\,ar \vdash\!\!- \underline{list}$ should be used and the missing , has been discovered.

One must show how the reduction sets can be used to correct the errors in the string $\eta = \beta_1\,?\,\beta_2\,?\,\ldots\,?\,\beta_{mk}$, and to reduce it to a meta-symbol A . Often there is more than one reduction. In that case the left and the right context must be used to choose the correct reduction, But there are strings for which the parsing process reaches the state

$$(\,\#\lceil\eta\rfloor \quad | \quad \# \quad | \quad \varepsilon \quad)$$

and no other context is available.

Let us first consider the simple case $\eta = \beta_1\,?\,\beta_2$. Some errors are present in the string η , and they can be either missing characters between β_1 and β_2 , or some changed characters in the tail of β_1 , or in the head of β_2 . Normally β_1 must be a prefix of a right hand side of a production, and the resolution set Res (β_1) contains all productions $\pi_j : A \longrightarrow \beta_1\,u$, for some string u. If there is no such production, then a phrase error has been discovered, and it must be handled by the procedure PHRASE. The resolution set Res (β_2) contains all pairs (B , k) , with the property that the production $\pi_k : B \longrightarrow v\,\beta_2$, has β_2 as a suffix. Again, phrase errors can be detected when the set

– Res (β_2) is built.

For example, such an error is met in analysing the string

$$w_5 = \#\# A 5 (10) , 9 , , V (9) \#\# ,$$

in the language generated by the productions (4.3.1). The error is discovered when the parsing process enters the state

($\#\# \lceil$ v (i) , i , ? , ar $\rfloor$ | $\#\#$ | ε)

Here both strings β_1 = v (i) , i , and β_2 = , ar contain phrase errors.

It is assumed that these errors have already been corrected by the procedure PHRASE. Therefore, (A , j) $\in$ Res (β_1) if the production π_j has the form

1a) $\pi_j : A \longrightarrow \beta_1$

1b) $\pi_j : A \longrightarrow \beta_1 u$,

and (B , k) $\in$ Res (β_2) , if the production π_k has the form

2a) $\pi_k : B \longrightarrow \beta_2$

2b) $\pi_k : B \longrightarrow v \beta_2$,

for some strings u and v .

There are several possibilities for correcting the errors and for reducing the string β_1 ? β_2 , and they are depicted in fig. 4.3.10.

1) The strings β_1 and β_2 are parts of the same phrase. In this case Res (β_1) and Res (β_2) must contain a common production

$$\pi : A \longrightarrow \beta_1 u \beta_2$$

for some string u . The string u has been deleted and this is the cause of the question mark between β_1 and β_2. (Fig. 4.3.10a).

2) The string β_1 must be reduced first to a metasymbol A , using the reduction rule $\beta_1 u_1 \vdash\!\!\!- A$, for some string u. It is considered that u_1 has been deleted and now is inserted back into the sentence. Then A and β_2 must be reduced, using the reduction

$$\eta_1 A u_2 \beta_2 \vdash\!\!\!\!\!+ B.$$

Here η_1 must match the left context and u_2 has been deleted

and now is inserted back into the string. (Fig. 4.3.10b).

Thus, for all pairs $(A , j) \in \mathrm{Res} (\beta_1)$ must be verified if A is related to the character on the stack, preceding $\lceil \eta \rfloor$, and if there is $(B , k) \in \mathrm{Res}(\beta_2)$ such that the production π_k has the form

$$\pi_k : B \longrightarrow \tau \beta_2 \quad , \text{ and } \quad \tau \overset{+}{\Longrightarrow} \eta_1 A u_2 .$$

3) This case is similar to 2) but the order of reductiom differ. The string β_2 must be reduced first to a metasymbol B , then the string $\beta_1 u B \eta_2$ must be reduced to A . (Fig. 4.3.10c). It follows that, for all pairs $(B , k) \in \mathrm{Res} (\beta_2)$, it must be checked if there is a production $\pi_j : A \longrightarrow \beta_1 \tau$, i.e., $(A , j) \in \mathrm{Res} (\beta_1)$, and if

$$\tau \overset{+}{\Longrightarrow} u B \eta_2 \quad , \text{ for some terminal strings}$$

u and v .

4) The last possibility is that both β_1 , and β_2 must be reduced to A , and B respectively. Then, together with more left and right context, a derivation

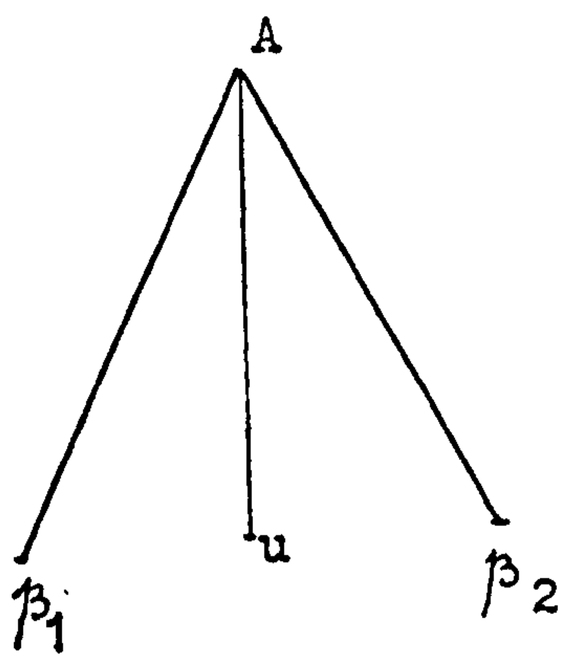
$$C \overset{+}{\Longrightarrow} \eta_1 A u B \eta_2 , \qquad (4.3.6)$$

is found. Here η_1 corresponds to the left context, η_2 corresponds to the right context and u is a terminal string that must be inserted (fig. 4.3.10d). Certainly if η_1 does not match the left context, or if η_2 does not match the right context, the derivation (4.3.6) must be rejected.

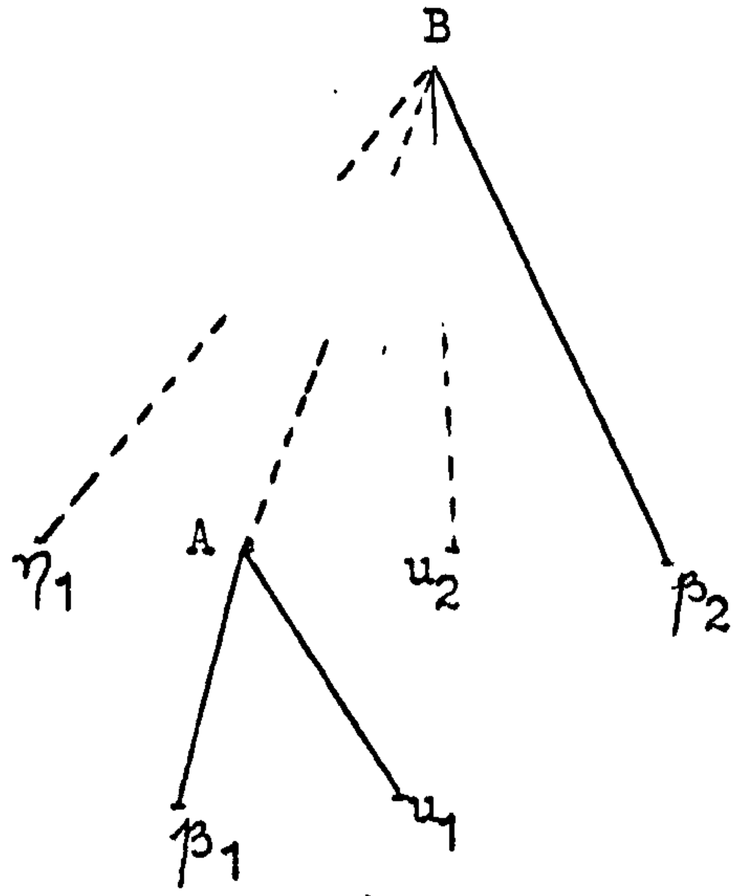
These facts are summarised in the flowchart of fig. 4.3.11.

Clearly, the smaller a reduction set is, the better. This is so, since for a longer substring β_1 , only a few productions will contain β_1 . For a string β_1 consisting of only one character, many productions will contain the string β_1 , and the resolution set $\mathrm{Res} (\beta_1)$ is larger and less reliable.

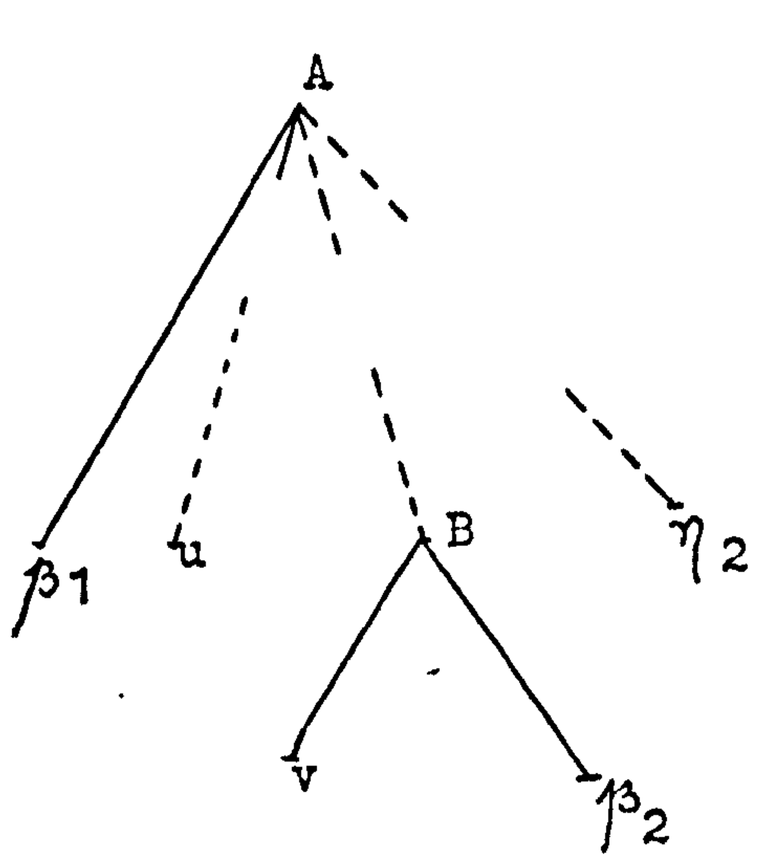
When the string η contains more than one '?' , the correction of the errors seems more complicated, but the idea is the same. Still,



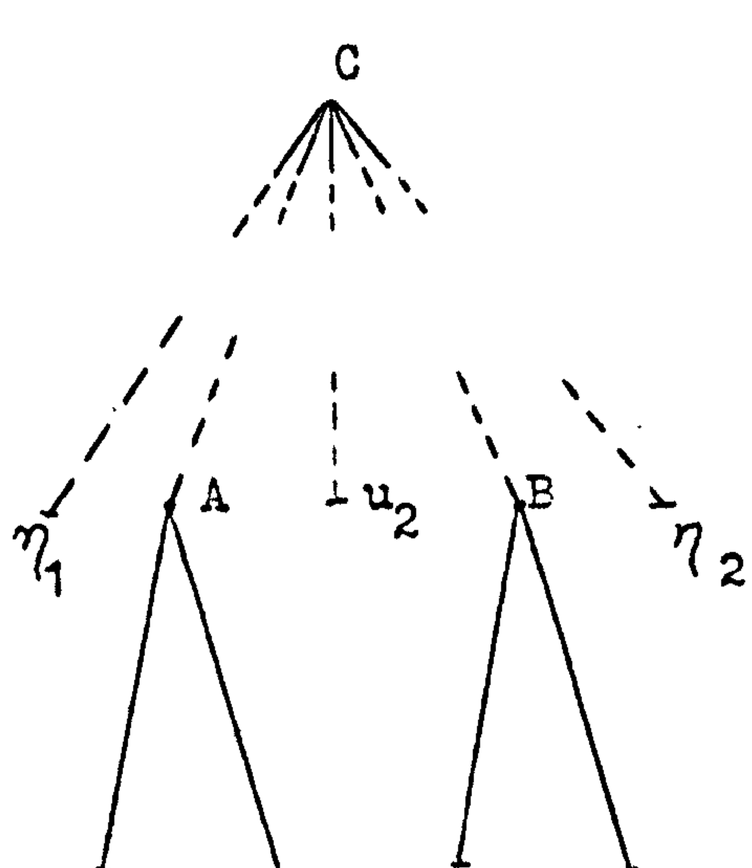
a)



b)



c)



d)

Fig. 4.3.10

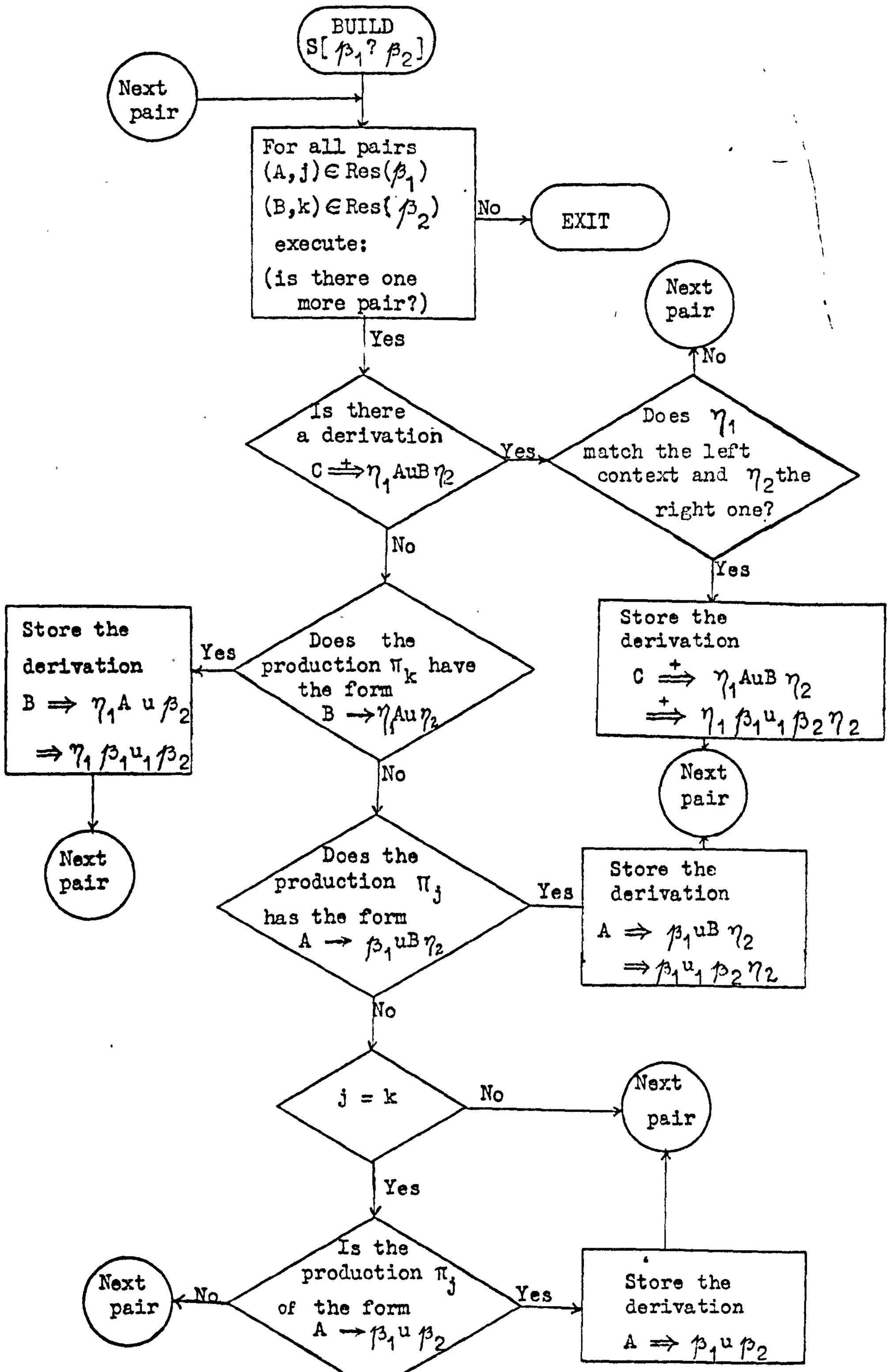

BUILD S[β1 ? β2]
Next pair
For all pairs (A,j) ∈ Res(β1) (B,k) ∈ Res(β2) execute: (is there one more pair?)
No
EXIT
Yes
Is there a derivation C ⇒+ η1 AuB η2
Yes
Does η1 match the left context and η2 the right one?
No
Yes
Store the derivation C ⇒+ η1 AuB η2 ⇒+ η1 β1 u1 β2 η2
Next pair
Does the production πk have the form B → η1 Au η2
Yes
Store the derivation B ⇒ η1 A u β2 ⇒ η1 β1 u1 β2
Next pair
No
Does the production πj has the form A → β1 uB η2
Yes
Store the derivation A ⇒ β1 uB η2 ⇒ β1 u1 β2 η2
No
j = k
No
Next pair
Yes
Is the production πj of the form A → β1 u β2
No
Next pair
Yes
Store the derivation A ⇒ β1 u β2

Fig. 4.3.11.

there is a slight difference. A string β_j , j = 2 , 3 , ..., mk-1 , is enclosed within question marks, and there is the possibility that β_j has been inserted. Now it is not known β_j is the prefix or a suffix of a phrase, as it was known before about β_1 and β_2 . Thereby, the string β_j , j = 2 , 3 , ... , mk-1 , can be reduced alone, together with β_{j-1} , together with β_{j+1} , or together with both.

The minimum context that must be taken into account when η is reduced, is the incoming input character and the immediate left symbol in the stack. Thus, if the parsing process is in the state

(# η_1 q [η] | i | rest of string) , the symbols q and i must be used to choose a possible reduction for η , or part of η . Let us say, that there is only one metasymbol C that is related to i , and they both form part of a simple phrase. This is the case when Matr (C , i) = $\doteq$, and there is no other metasymbol B such that Matr (C , i) = $\doteq$, where Matr is the precedence matrix of the grammar. Then β_{mk} must be a suffix of a C-phrase. If Res (β_{mk}) contains only the pair (C , k) , there is no doubt the production π_k must be used to reduce the string β_{mk} . In this case the reduction is made and the scanning phase may continue. This discussion is valid and for the case mk=2 .

Let us assume now that it is not possible to choose a unique derivation for the string η , and all derivations

$$C \overset{+}{\Longrightarrow} \eta_1 \beta_1 u_1 \beta_2 u_2 \cdots \beta_{mk} \eta_2 , \qquad (4.3.7)$$

must be built for some strings η_1 , η_2 , and some terminal strings u_1 , u_2 ... , u_{mk-1} . Clearly there are a finite number of such derivations, and hopefully, a small number. They can be constructed by an iterative process.

If all derivations

$$A \overset{+}{\Longrightarrow} \eta_1 \beta_1 u_1 \beta_2 u_2 \cdots u_{j-2} \beta_{j-1} \eta_2 , \qquad (4.3.8)$$

are known, they must be used to produce all derivations

$$C \overset{+}{\Longrightarrow} \eta_0 \beta_1 u_1 \beta_2 u_2 \cdots \beta_{j-1} u_{j-1} \beta_j \eta_3 . \qquad (4.3.9)$$

For $j=2$ the derivations (4.3.8) are known, and they are given by $Res(\beta_1)$. Thus for $j = 2,3, \ldots ,mk$, a similar process of adding β_j to the derivations (4.3.8), must be executed. If CONNECT is the procedure that constructs the derivations (4.3.9) from the derivations (4.3.8), using $Res(\beta_j)$, then CONNECT must be executed for $j = 2,3, \ldots , mk$.

There are four posibilities of constructing the derivation (4.3.9) from (4.3.8). First is the linkage of β_j to the derivation (4.3.8), if $\eta_2 \overset{*}{\Longrightarrow} u\,\beta_j\,\eta_3$. That is, either η_2 contains β_j as a substring, or η_2 contains a metasymbol B_1, such that $(B,k) \in Res(\beta_j)$, for some B, and k, and $B1 \overset{*}{\Longrightarrow} u_1 B\, v_1$ (Fig.4.3.12a).

Second posibility occurs when β_j is only a part of a simple phrase, and a metasymbol is needed in the left part of this phrase, i.e. there is $(B,k) \in Res(\beta_j)$, such that π_k: $B \rightarrow \gamma_1 \beta_j \gamma_2$, where γ_1 contains a metasymbol. In this case, if $\gamma_1 \overset{*}{\Longrightarrow} \gamma_3 Au$, the derivation $B \Longrightarrow \gamma_1 \beta_j \gamma_2 \overset{*}{\Longrightarrow} \gamma_3 Au\, \beta_j \gamma_2 \overset{+}{\Longrightarrow} \gamma_3 \eta_1 \beta_1 u_1 \cdots \beta_j \gamma_2$, must be added to S.(fig.4.3.12.b).

Third posibility is to find a derivation

$$C \overset{+}{\Longrightarrow} \gamma_1 A\, u\, B\, \gamma_2,$$

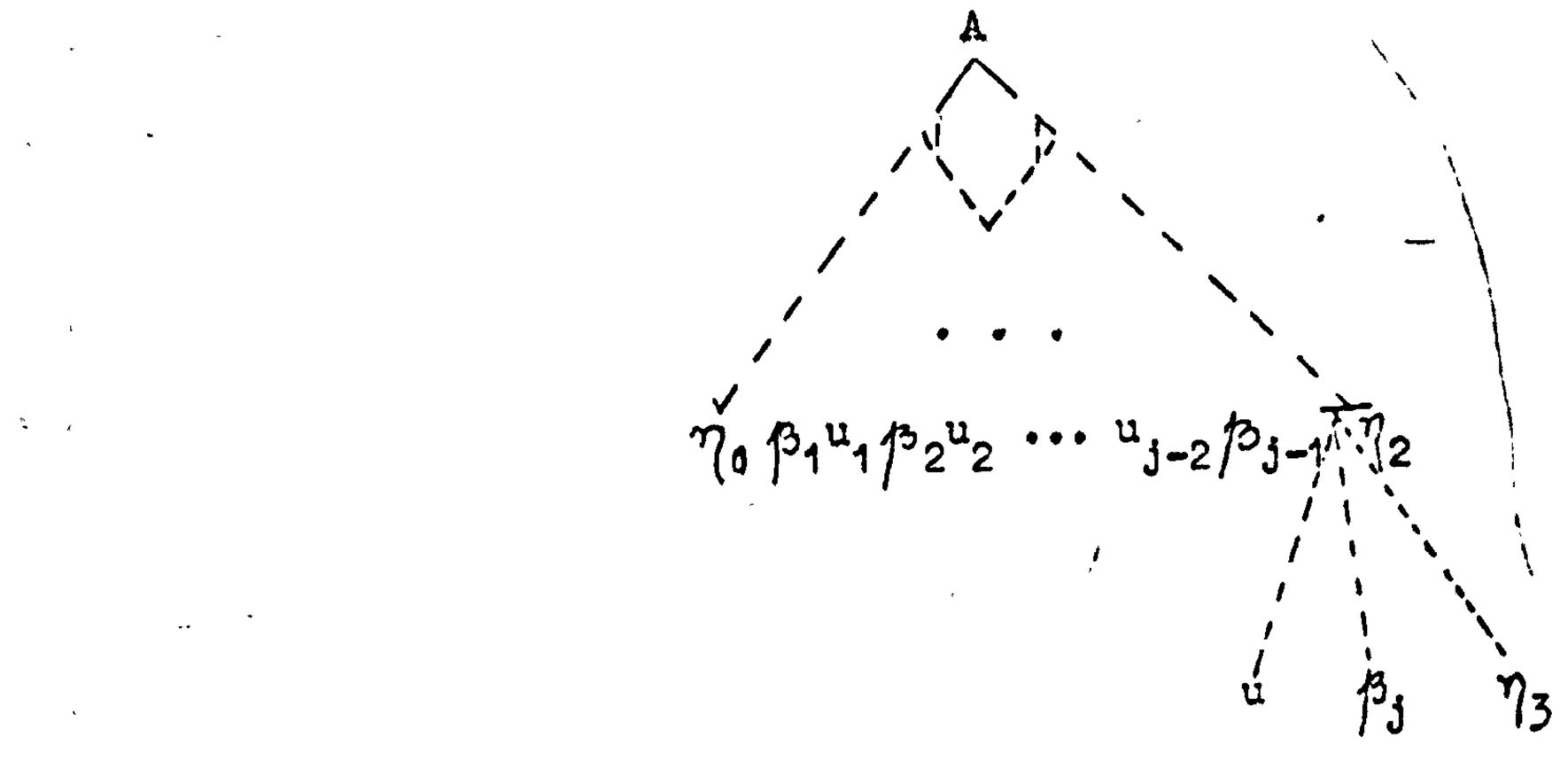
in which case, the derivation

$$C \overset{+}{\Longrightarrow} \gamma_1\, A\, u\, B\, \gamma_2 \overset{+}{\Longrightarrow}$$

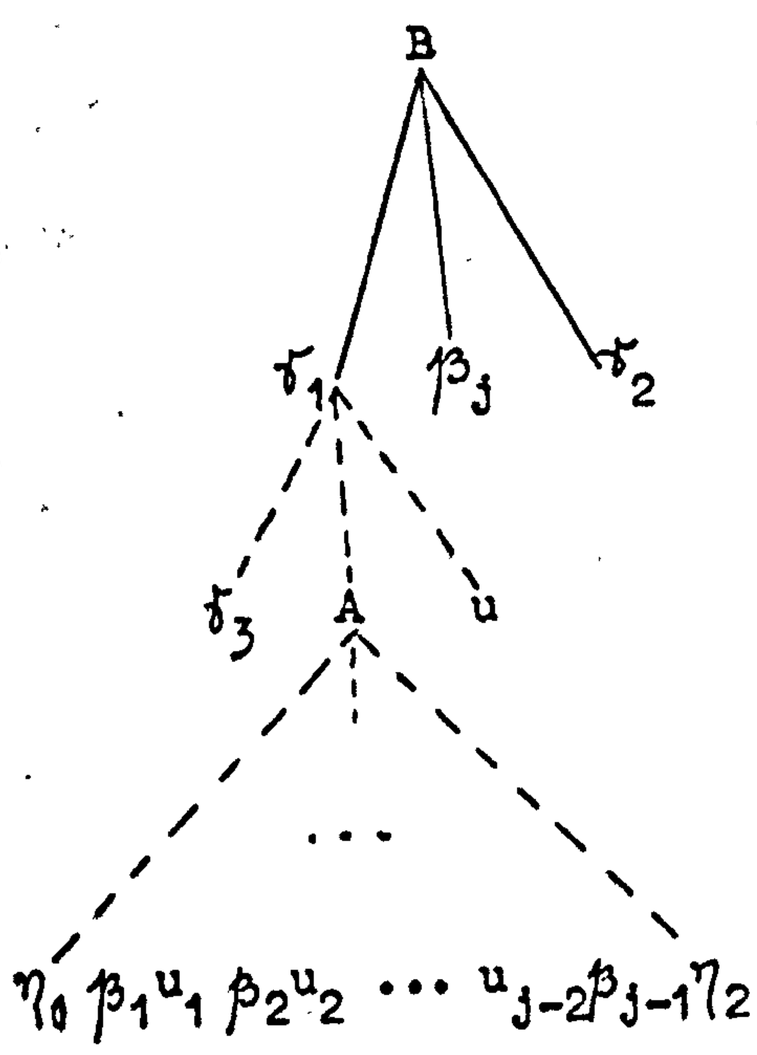
$$\overset{+}{\Longrightarrow} \gamma_1\, \eta_1 \beta_1 u_1 \cdots \beta_{j-1} \eta_2 u\, \beta_j \eta_3,$$

must be added to S (fig. 4.3.12.c).

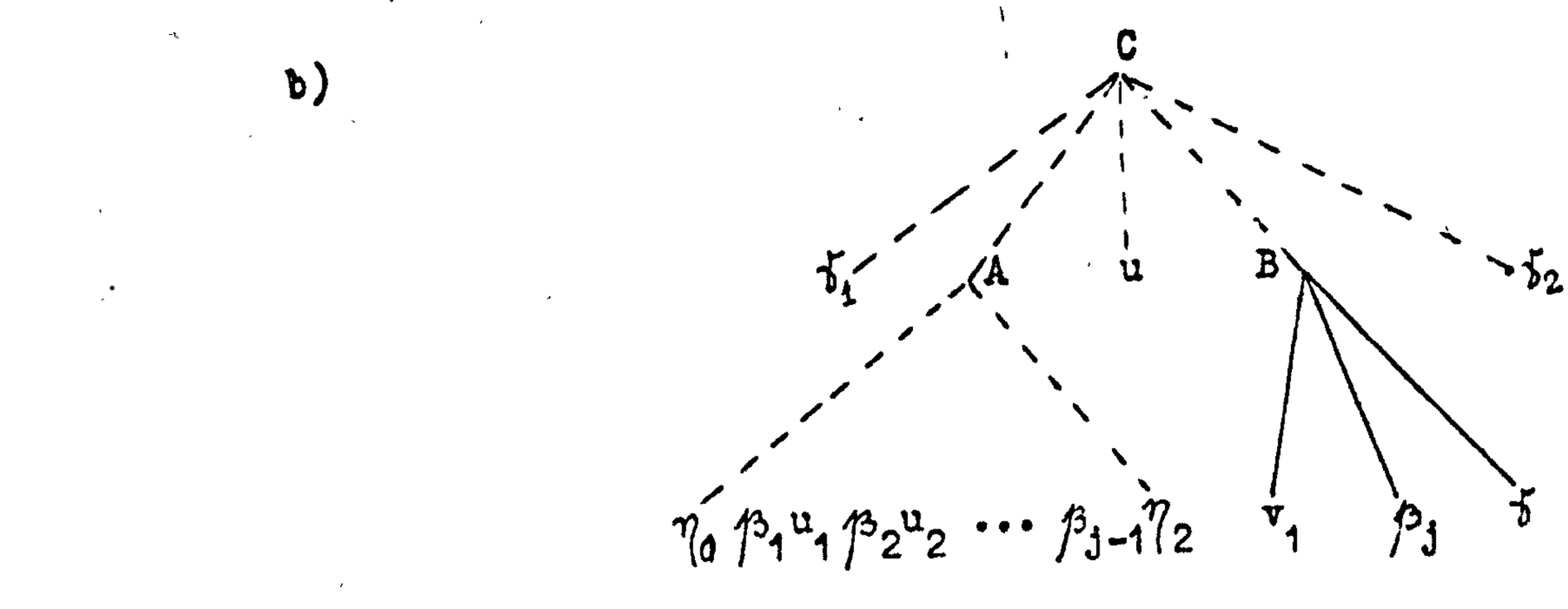
Last hypothesis that must be taken into account is to delete the string β_j, in which case all derivations (4.3.8) are carried forward. This must be the case when β_j is a short string, and does not contain metasymbols.


A
$\eta_0 \beta_1 u_1 \beta_2 u_2 \cdots u_{j-2} \beta_{j-1} \eta_2$
u
β_j
η_3

a)


B
γ_1
β_j
γ_2
γ_3
A
u
$\eta_0 \beta_1 u_1 \beta_2 u_2 \cdots u_{j-2} \beta_{j-1} \eta_2$

b)


C
γ_1
A
u
B
γ_2
$\eta_0 \beta_1 u_1 \beta_2 u_2 \cdots \beta_{j-1} \eta_2$
v_1
β_j
γ

c) .

Fig. 4.3.12.

A flowchart of the procedure CONNECT is given in fig. 4.3.14. It is called by the procedure BUILD which builds all derivations (4.3.7). The procedure BUILD can be seen in fig. 4.3.13.

From these many derivations (4.3.9), there must be chosen that one that matches the left context in the stack. It was understood that the strings u_i can be empty, or consist of small terminal strings.

Finally the continuation PHRASE of the flowchart of fig. 4.3.2 must be completed. PHRASE is the procedure that corrects one, or more phrase errors. This procedure is called when a potential simple phrase η has been discovered, but the procedure REDUCE fails to find a production of the form

$$\Pi : A \longrightarrow \eta \quad .$$

Such an error occurs when a prefix of a right hand side of a production, and a suffix of another one, merge together. One example can be found in fig. 4.3.1. Here are some more examples

a) v (i , i , i , i , i , ar

b) i d d d d

c) ar , i , i , i , i

in the same language, defined by the productions (4.3.1). Certainly some phrase errors, as in example c), although they are theoretically possible, may never happen.

Since there are a finite number of productions, and each one has a limited number of characters, there are only a finite number of different phrase errors, and they can be stored in a table. This has already been suggested [L6].

Here will be given a procedure (as opposed to a look up table as above) that corrects these errors. If η contains a phrase error, it has the form

$$\eta = \beta_1 \alpha_1 \beta_2$$

where $\beta_1 \alpha_1$ is a prefix of a production, and $\alpha_1 \beta_2$ is a suffix of a right hand side of another production.

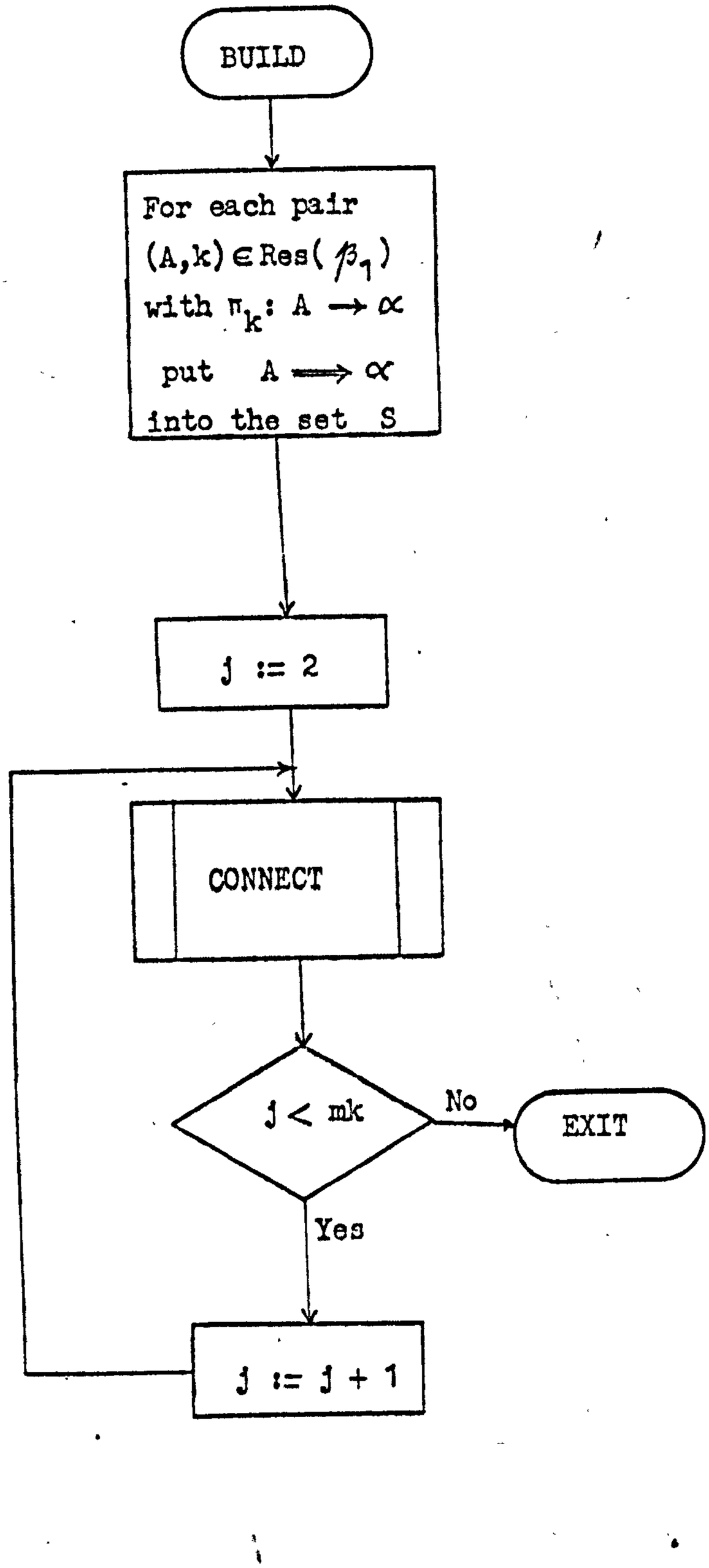

BUILD
For each pair
(A,k) ∈ Res(β1)
with πk: A → α
put A ⟶ α
into the set S
j := 2
CONNECT
j < mk
No
EXIT
Yes
j := j + 1

Fig. 4.3.13

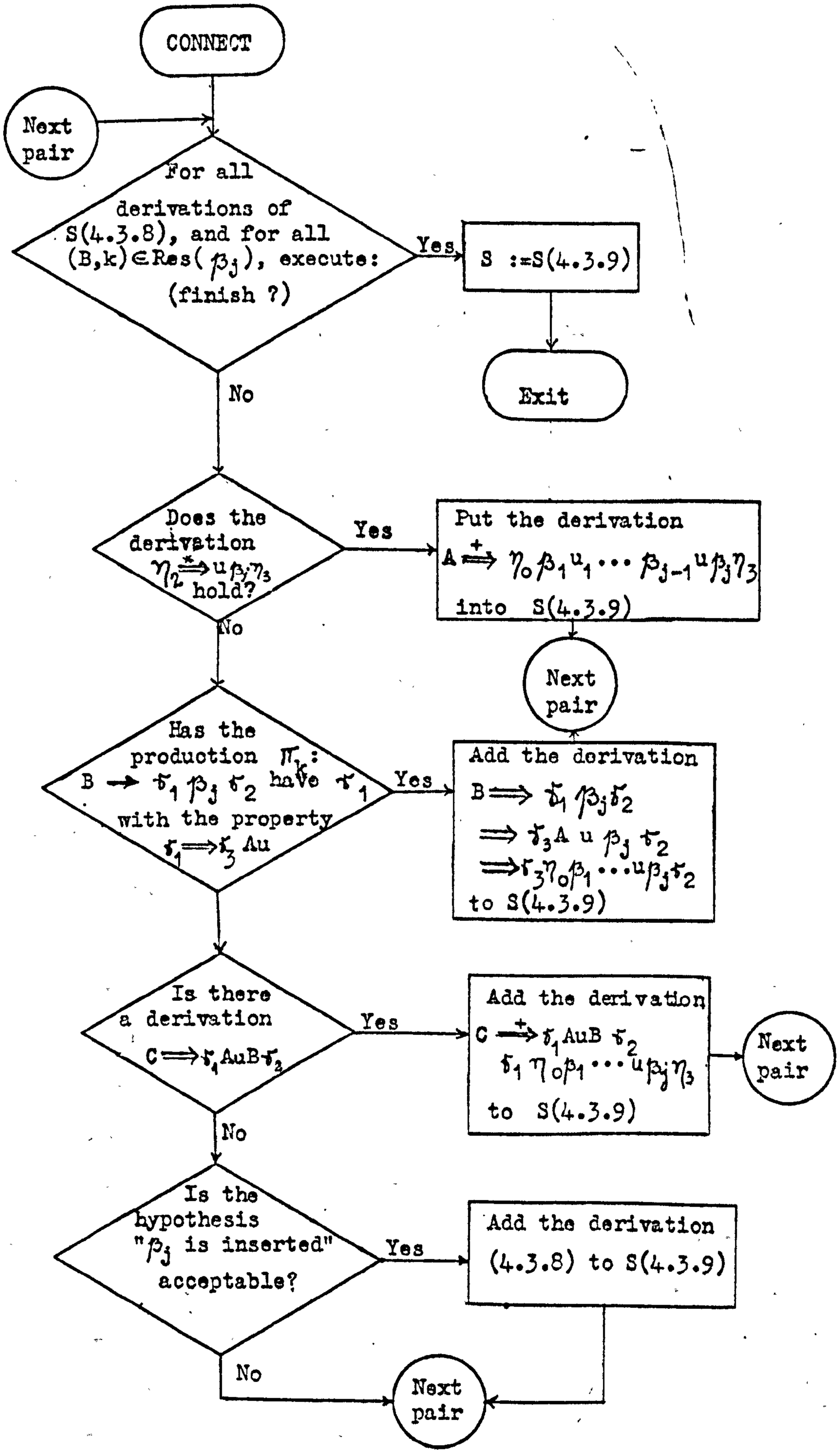

CONNECT
Next pair
For all derivations of S(4.3.8), and for all (B,k)∈Res(β_j), execute: (finish ?)
Yes
S :=S(4.3.9)
Exit
No
Does the derivation $\eta_2 \overset{*}{\Longrightarrow} u\beta_j\eta_3$ hold?
Yes
Put the derivation $A \overset{+}{\Longrightarrow} \eta_0\beta_1 u_1 \cdots \beta_{j-1} u \beta_j \eta_3$ into S(4.3.9)
No
Next pair
Has the production π_k: $B \to \tau_1 \beta_j \tau_2$ have τ_1 with the property $\tau_1 \Longrightarrow \tau_3 Au$
Yes
Add the derivation $B \Longrightarrow \tau_1\beta_j\tau_2 \Longrightarrow \tau_3 A u \beta_j \tau_2 \Longrightarrow \tau_3\eta_0\beta_1 \cdots u\beta_j\tau_2$ to S(4.3.9)
Is there a derivation $C \Longrightarrow \tau_1 AuB\tau_2$
Yes
Add the derivation $C \overset{+}{\Longrightarrow} \tau_1 AuB \tau_2$ $\tau_1\eta_0\beta_1 \cdots u\beta_j\eta_3$ to S(4.3.9)
Next pair
No
Is the hypothesis "β_j is inserted" acceptable?
Yes
Add the derivation (4.3.8) to S(4.3.9)
No
Next pair

Fig. 4.3.14.

If η contains more than one phrase error, it has the form

$$\eta = \beta_1 c_1 \beta_2 c_2 \cdots c_{l-1} \beta_l \quad , \qquad (4.3.10)$$

and $c_1 \beta_2 c_2$ must be a substring of a right hand side of a production, $c_2 \beta_3 c_3, \cdots, c_{l-2} \beta_{l-1} c_{l-1}$, must have the same property, $\beta_1 c_1$ must be a prefix of a right hand side of a production, and $c_{l-1} \beta_l$ must be a suffix of another one.

Let us consider again the example of fig. 4.3.2, i.e.

$$\eta = v\,(\,i\,,\,ar\,,\,ar\,.$$

All pairs of consecutive characters are related by $\doteq$, thus η is a potential simple phrase, but it is not a right hand side of any production. But v (i is a prefix of two productions

$$\pi_2 : ar \longrightarrow v\,(\,i\,,\,i\,)$$

$$\pi_3 : ar \longrightarrow v\,(\,i\,)$$

and ar,ar is a substring of the right hand side of the production

$$\pi_1 : ld \longrightarrow ar\,\lceil\,,\,ar\,\rceil\,.$$

It is clear that , must be considered in only one of them, and the production π_1 needs this comma, but π_3 does not need it. Thus

$$\eta = \beta_1 \beta_2$$

where β_1 = v (i is a substring in π_2 and π_3, and β_2 = , ar , ar , is a substring in π_1. To choose between π_2 and π_3, it is enough to compute the cost of changing β_1 into v (i , i) or v (i). Certainly here v (i) is the choice of minimum cost, but the chosen reduction must agree with the left and right context.

It follows that PHRASE must have two parts - one that detects the merging characters c_i and the decomposition (4.3.10), and another part, that connects these parts into a syntax tree, i.e. corrects the errors.

The string $\eta = b_1\ b_2 \cdots b_m$ must be decomposed as follows. First, the longest prefix $b_1\ b_2 \cdots b_k$, of η is sought, such that $b_1\ b_2 \cdots b_k$ is aprefix of a right hand side of a production. Since the last character b_k is the merging character, it is not known if it is part of the left phrase or of the right one. Thus $\beta_1 = b_1\ b_2 \cdots b_{k-1}$ and β_2 starts

with b_{k+1}. The end of β_2 is determined in the same way in which it was found for β_1. Thus the decomposition

$$\eta = \beta_1 c_1 \beta_2 c_2 \cdots c_{m-1} \beta_m$$

is obtained. For each β_i the set $\mathrm{Res}(\beta_i)$ is defined as before, and η must be corrected exactly as η was corrected when it contained errors marked by '?'.

One now starts all once again with a view to analysing a string into a simple precedence language, leaving open the option of using the error correcting procedure described above. During the parsing process, all correct simple phrases will be reduced to metasymbols. Three delimiters [, ? , and] , which are not terminals of the language, are used to mark the beginning of a phrase, the presence of an error, or the end of a phrase, respectively. Let d denote any delimiter. If the whole input string has been scanned, the parsing process reaches the state

$$(\# d_0 \beta_1 d_1 \beta_2 d_2 \cdots \beta_r d_r \mid \# \mid \varepsilon). \qquad (4.3.11)$$

Now the whole information available from the input string is stored in the stack.

If the delimiters $d_1 , d_2 , \ldots , d_{r-1} ,$ are all '?' , then the string

$$\beta_1 ? \beta_2 ? \cdots ? \beta_r$$

must be a wrong version of a sentence. From all derivations obtained by BUILD there must be retained that, or those, that have C = ' sentence metasymbol '. If there is more than one, then that one, which needs less corrections than the others, is choosen.

If $d_0 = d_1 = \ldots = d_k =$ '[' , we have, probably, a correct left context that can be used to correct the errors in the stack string

$$\eta = \beta_{k+1} ? \beta_{k+2} ? \cdots ? \beta_{k+1} ,$$

as can be seen in [L6].

If $d_1 = ?$, then we can use the right context to choose the correct derivations constructed for η by BUILD. This can be done in exactly the same manner as using the left context, but proceeding from right to left.

The error-correction can be postponed until the end of the scanning phase. Then the whole information is available in the stack. That is why the error correction is called global. For each substring β_j in (4.3.11), a resolution set Res (β_j) , is built. The reliability of the set Res (β_j) depends heavily on the string β_j , and on the delimiters d_{j-1} and d_j . If $d_{j-1} = d_j = '\lceil'$, the resolution set is considered very reliable. If d_{j-1}, $d_j = '?'$, the reliability of Res (β_j) depends on the length of β_j. If β_j is only a terminal character 'a' , the corresponding resolution set cannot be considered reliable. That is why, when the string $\beta_j = 'a'$ is enclosed within '?' marks, it is defined Res ('a') = $\{(a , o)\}$.

Still, we have not adhered to the principle of postponing the corrections until the end of the scanning phase. We have tried to correct the errors as early as possible. Very probably the string η , enclosed with $\lceil \rfloor$, contains only one ? mark. Since $\lceil$ marks the beginning, and $\rfloor$ marks the end of a phrase, we try to correct the errors at the level of a phrase, which seems reasonable. Certainly, we must not forget to use the surrounding context.

This error correction philosophy certainly implies more than trying different local corrections. The idea of changing a character around '?' marks, although a correct one, seems naive now. Although we are doing much the same thing, it is done at a higher level, at the phrase or sentence level.

An implementation of an error correcting parser is given in the next chapter.

4.4 A GLOBAL TOP-DOWN ERROR CORRECTING PARSER

4.4.1 A global top-down parser has been presented in 2.4. The problem faced by that parser at each step is to analyse (parse) a terminal string α as an A-phrase. When α is an A-phrase, the algorithm described there will give the derivation

$$A \overset{+}{\Longrightarrow} \alpha .$$

But what happens if α is an erroneous A-phrase? This is the problem faced by an error correcting parser which we shall try to attack in a top-down manner in the rest of this section.

As has already been said, although the errors happen, their occurrence is not too probable and this fact is used in the iterative procedure described below. We expect the input string to be a correct sentence and try to parse it. If we fail we first assume that one error has occurred and we try to find an A-phrase β such that $d(\beta, \alpha) = 1$. Here $d(\beta, \alpha)$ is the distance between the strings β and α, defined in 3.1. (For simplicity $u(e) = 1$ for all edit operations, but any weight u can be used). If we fail again we assume that two errors have occurred and try to find an A-phrase β such that $d(\beta, \alpha) = 2$. Note that if the length of the input string α is $n = |\alpha|$ and the length of the shortest A-phrase is l_o, then $d(\beta, \alpha) \leq n + l_o$, i.e. $n + l_o$ is an upper bound for the number of the errors that have occurred. This discussion is summarised in the flowchart of the fig. 4.4.1.

The whole problem has been reduced to a simpler one: is there an A-phrase β such that $d(\beta, \alpha) \leq Ner$, for a given number Ner? If there is one, construct the syntax tree of the A-phrase β.

We define

$$d(A, \alpha) = \min_{\beta} \{ d(\beta, \alpha) \mid \beta \text{ is an A-phrase} \}, \quad (4.4.1.)$$

i.e, $d(A, \alpha)$ is the minimum number of edit operations needed to change the string α into an A-phrase.

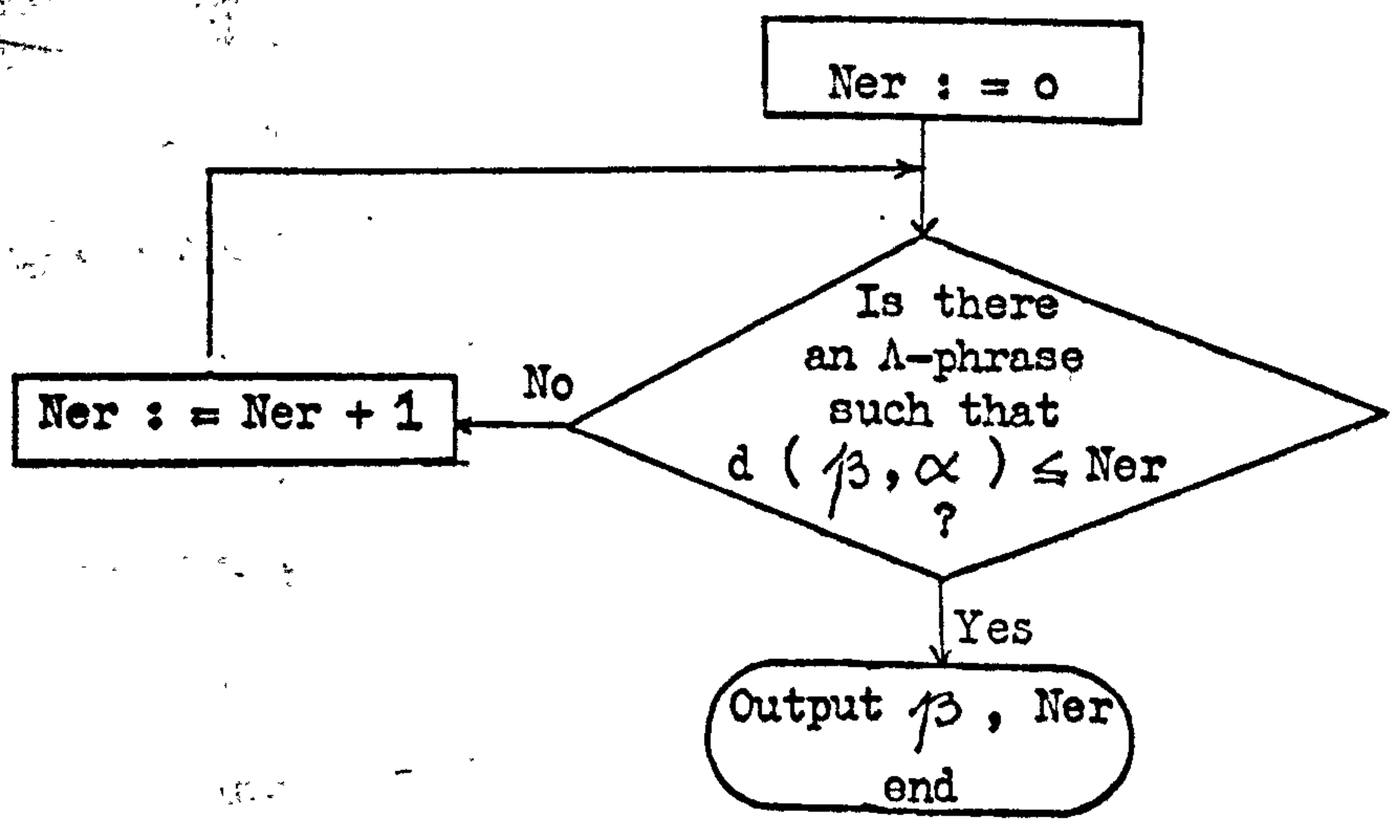


Fig. 4.4.1

Now, if $d(\beta,\alpha)\leq Ner$, then according to the definition 3.1.3 there exists a trace T, between β and α such that $u(T) < Ner$. Since β is an A-phrase, there is a derivation $A \overset{+}{\Longrightarrow} \beta$. Let

$$\pi_A : A \longrightarrow u_1 A_1 u_2 A_2 \cdots u_l A_l u_{l+1} ,$$

be the first production used in this derivation. Here u_i are terminal strings, maybe empty, and A_i are metasymbols. It follows that β can be decomposed as

$$\beta = u_1 \beta_1 u_2 \beta_2 \cdots u_l \beta_l u_{l+1} , \qquad (4.4.2)$$

where β_i are A_i-phrases, $i = 1, 2, \ldots, l$. By the trace T, to the decomposition (4.4.2) there corresponds a decomposition

$$\Delta_\alpha = \Delta(\alpha) : \quad \alpha = v_1 \alpha_1 v_2 \alpha_2 \cdots v_l \alpha_l v_{l+1}, \qquad (4.4.3)$$

of the string α, such that

$$d(\beta,\alpha) = \sum_{i=1}^{l+1} d(u_i, v_i) + \sum_{i=1}^{l} d(\beta_i, \alpha_i). \qquad (4.4.4)$$

The decomposition (4.4.3) and the formula (4.4.4) give an insight into the solution of our problem. We know the string α and we wish to find an A-phrase β, such that $d(\beta,\alpha)$ is not greater than Ner. Then, for all decompositions (4.4.3) of the string α, we must find that production π_A, for which

$$d(\pi_A, \Delta_\alpha) = \sum_{i=1}^{l+1} d(u_i, v_i) + \sum_{i=1}^{l} d(A_i, \alpha_i), \qquad (4.4.5)$$

is not greater than Ner. If we find one, the problem is solved.

Note that a procedure DIST (A , α , d) which computes d = d (A , α) , according to (4.4.5), will have to be defined recursively, since it calls itself for other values of the parameters.

The fact that we have a global error correcting parser, reflects itself in the sum $\sum_{i=1}^{l+1} d (u_i , v_i)$, the first part of the sum in (4.4.5). For a small number Ner , this sum will be greater than Ner for many productions π_A , and there is no need to proceed further, but to reject those productions.

Let $\pi(A , 1)$, $\pi(A , 2)$, ... , $\pi(A , l_A)$ be all A-productions. The procedure DIST (A , α, d) chooses in order an A-production , starting with $\pi(A , 1)$. To each A-production all decompositions Δ_α are examined, in a precise order. For each pair (A-production, Δ_α-decomposition), the distance σ(production, decomposition) is computed and DIST (A_k , α_k,d_k) is called, for k = 1 , 2 , ... , l . If $\sigma + d_1 + d_2 + \ldots + d_l$ became greater than Ner, the next decomposition Δ_α is chosen. If no other decomposition exists, the next A-production is tried. If no other A-production exists, then there is not an A-phrase β such that $d (\beta , \alpha) \leq$ Ner.

The description of the procedure DIST can be found in the flowchart of the fig. 4.4.2.

Something is lacking there. What is meant by the next decomposition Δ_α ? Since there is a need to examine all possible decompositions, they are order. Let

$$\Delta_\alpha(1) = u_1 \; u_2 \; u_3 \ldots u_m$$

and

$$\Delta_\alpha(2) = v_1 \; v_2 \; v_3 \ldots v_m ,$$

be two decompositions of the string α, into the same number m of sub-strings u_i , and v_i respectively. Let | u | denote the length of the string u . Then

$$\Delta_\alpha(1) < \Delta_\alpha(2),$$

iff there is an integer $i < m$ such that $|u_1| = |v_1| , |u_2| = |v_2| , \ldots , |u_i| = |v_i|$

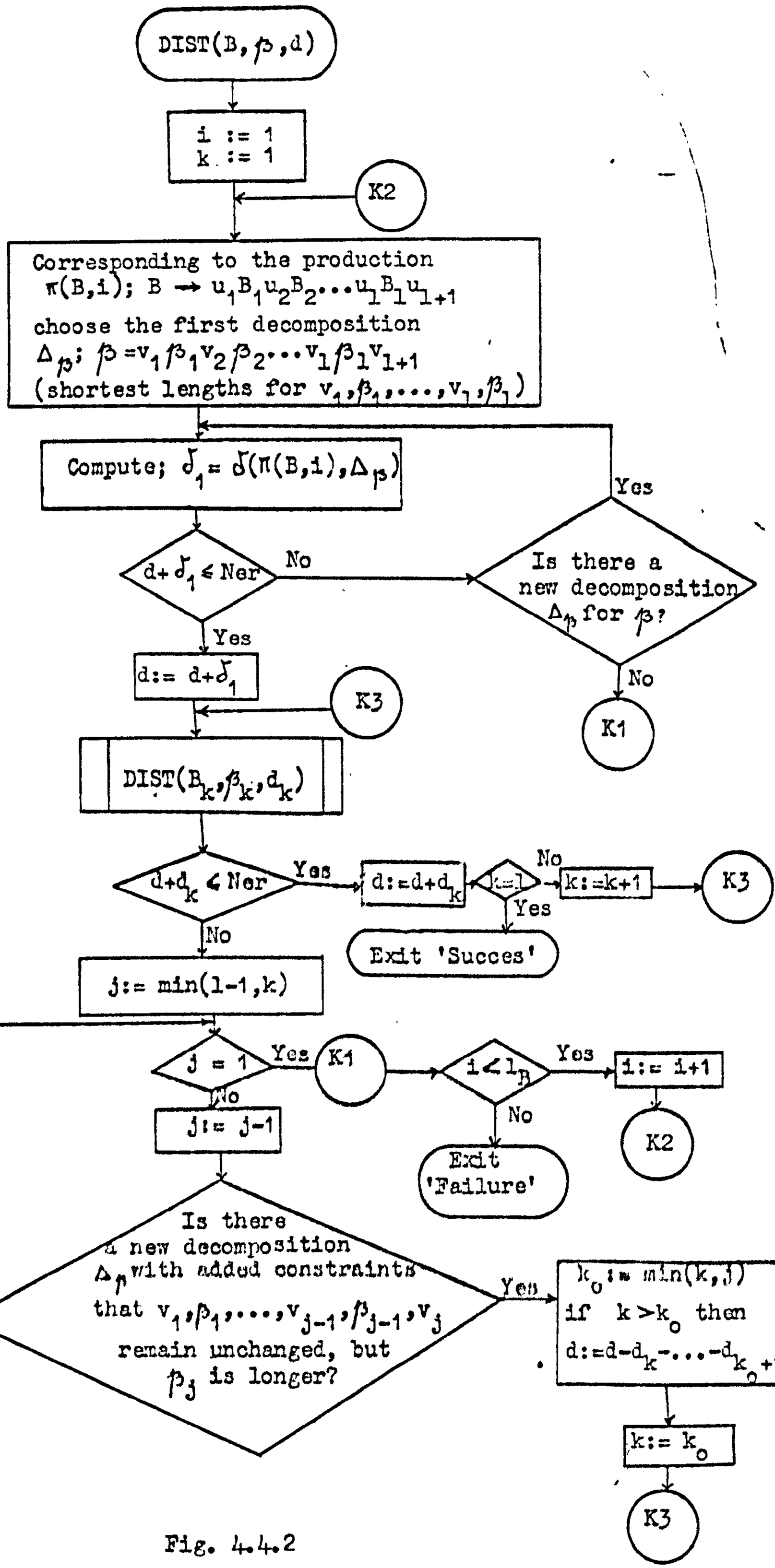

DIST(B, β, d)
i := 1
k := 1
K2
Corresponding to the production
π(B,i); B → u1B1u2B2...ul Bl ul+1
choose the first decomposition
Δβ; β = v1β1v2β2...vlβlvl+1
(shortest lengths for v1, β1, ..., vl, βl)
Compute; δ1 = δ(π(B,i), Δβ)
d+δ1 ≤ Ner
No
Is there a new decomposition Δβ for β?
Yes
No
K1
Yes
d := d+δ1
K3
DIST(Bk, βk, dk)
d+dk ≤ Ner
Yes
d := d+dk
i = l
No
k := k+1
K3
Yes
Exit 'Succes'
No
j := min(l-1, k)
j = 1
Yes
K1
No
j := j-1
i < lB
Yes
i := i+1
K2
No
Exit 'Failure'
Is there a new decomposition Δβ with added constraints that v1, β1, ..., vj-1, βj-1, vj remain unchanged, but βj is longer?
No
Yes
k0 := min(k, j)
if k > k0 then
d := d-dk-...-dk0+1
k := k0
K3

Fig. 4.4.2

but $|u_{i+1}| < |v_{i+1}|$. We say that $\Delta_\alpha(1)$ is the next decomposition after Δ_α, if

a) $\Delta_\alpha < \Delta_\alpha(1)$

b) there is not a decomposition $\Delta_\alpha(2)$ such that

$$\Delta_\alpha < \Delta_\alpha(2) < \Delta_\alpha(1).$$

Finally, many would object to the large number of possible decompositions of the string α into m substrings. But there is no need to examine all possible decompositions. Since there are no more than Ner errors, we must have $||u_i| - |v_i|| \leq$ Ner, and since α_i must be an A_i-phrase, the substring α_i must satisfy some restrictions. This is what Unger [U2] has called 'quick checks'.

4.4.2.

The interest of this section is to describe a procedure CORRECT that takes an input string w, and, using the method presented above, gives the syntax tree of the nearest sentence in the language.

Let $\delta(\pi_A, \Delta_\alpha) = \sum_{i=1}^{l+1} d(u_i, v_i)$. (4.4.6)

According to (4.4.5)

$$d(\pi_A, \Delta_\alpha) = \delta(\pi_A, \Delta_\alpha) + \sum_{i=1}^{l} d(A_i, \alpha_i).$$

Thus, when the production π_A is matched against the string α, using the decomposition Δ_α, only the first term $\delta(\pi_A, \Delta_\alpha)$ is computed, and added to d, which must be initially zero. Since, at a given step, it is assumed that $d(S_0, w) \leq Ner$ (S_0 is the sentence metasymbol), if $d > Ner$, then a new decomposition Δ_α must be tried. If there is no other decomposition, then next A-production is used. If there is no other A-production, but A is not the sentence metasymbol, then the goal α of the metasymbol A is incorrect, or probably the production that has introduced the metasymbol A is not a correct one. In other words, often there is need to backtrack, to choose other productions, or other goals (decompositions), and the syntax tree is constructed dynamically.

Let us first consider an example. The grammar

$G = (\{A, B\}, \{a, b\}, P, A)$, is given

with P the set of productions

$\pi_1 : A \longrightarrow aa$

$\pi_2 : A \longrightarrow ABA$

$\pi_3 : B \longrightarrow bb$

$\pi_4 : B \longrightarrow BB$.

The string w = aa ab ba aa must be analysed. This, evidently, is not a correct sentence of $L(G)$, since each sentence of $L(G)$ has

an even number of consecutive a's , and an even number of consecutive b's. Thus, for Ner = 1 , 2 , ... , an A-phrase β_{Ner} is sought, such that

$$d (w , \beta_{Ner}) \leq Ner$$

Let us say that the syntax tree already built is that of fig.4.4.3a.

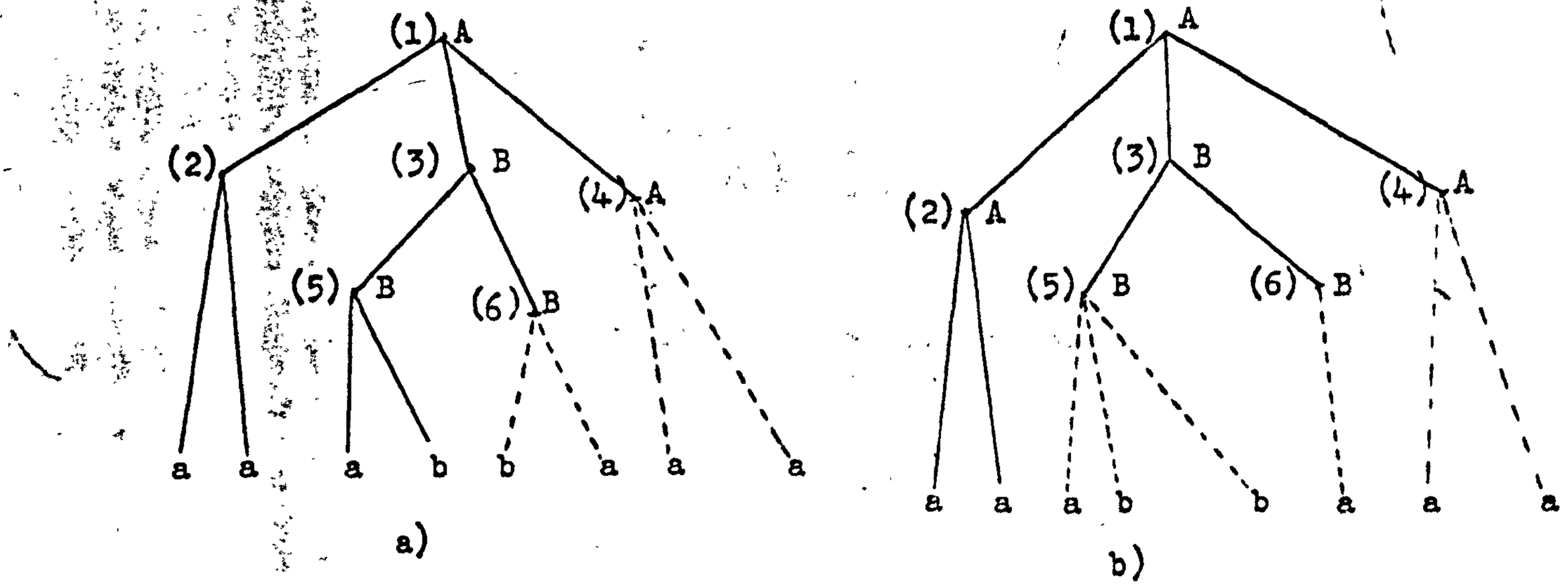


Fig. 4.4.3

The goal of the node (3) is the substring abba. Corresponding to the production $\pi_4 : B \longrightarrow BB$, this string has been decomposed as ab | ba , and the first substring is the goal of the node (5), and ba is the goal of (6). Node (5) has matched its goal with one error, d (B , ab) = 1 , and d became 1 , which is not greater than Ner = 1. But analysing the goal of node (6) one finds $d(\pi_3 , ba) = 1$, and d becomes 2 > Ner. No other decomposition of ba (in only one part!.) is possible, and the use of the production π_4 leads again to failure. Thus a new decomposition for the goal abba , of node (3) is sought. It is given in fig. 4.4.3b. Again d will be greater than Ner(= 1) and no other production for B , at the node (3),is possible. The next backtrack must be to the "father" of node (3), i.e. to the node (1), which must give other goals to its "sons", nodes (2), (3), and (4); i.e. a new decomposition of w is sought. But for all possible decompositions and productions d > Ner = 1. Then Ner is increased by 1. For Ner = 2, the syntax tree of fig. 4.4.3a, can be completed, since node (6)

will match its goal with one error, making $d = 2$, and the node (4) will match its goal without errors.

Since there is the possibility of failure, it must be clear which part of the syntax tree has been built, and where to backtrack. For this purpose each nonterminal node of the syntax tree is represented as a 7-tuple

$$node = (name , f , ob , fs , pr , goal , \delta) ,$$

where name is the name (number) of the node, f is its father (or first ancestor), ob is its elder brother, fs is its first son (or first descendent) pr is the production used by this node to match its goal, goal is the substring that must be matched (generated) by this node, and δ is the computed distance $\delta(pr , \Delta_\alpha)$, for the current decomposition of the goal α.

The syntax tree of the fig. 4.4.3a has the following representation (the question mark indicates data not yet known):

	name	f	ob	fs	pr	goal	δ
(1) = (	1 ,	ε,	ε,	,	π_2 ,	w ,	o)
(2) = (	2 ,	1 ,	ε,	ε,	π_1 ,	aa ,	o)
(3) = (	3 ,	1 ,	2 ,	5 ,	π_3 ,	abba ,	o)
(4) = (	4 ,	1 ,	3 ,	ε,	? ,	aa ,	?)
(5) = (	5 ,	3 ,	ε ,	ε,	π_4 ,	ab ,	1)
(6) = (	6 ,	3,	5 ,	ε,	π_3 ,	ba ,	1)

now found ↑

Node (4) must correspond to an A-phrase, but A is not present anywhere in the representation of this node, since the metasymbol is represented indirectly by the production. Therefore, when a new node is built, the production component cannot be ? , but the first A-production $\pi(A , 1)$.

If CORRECT (w) is the procedure that analyses the input string w and outputs the syntax tree of the nearest sentence in the language, its start is given in fig. 4.4.4. It builds the root of the syntax

tree, initialises d by o , and calls the subroutine NODE, for

A = 'sentence metasymbol', and α = w.

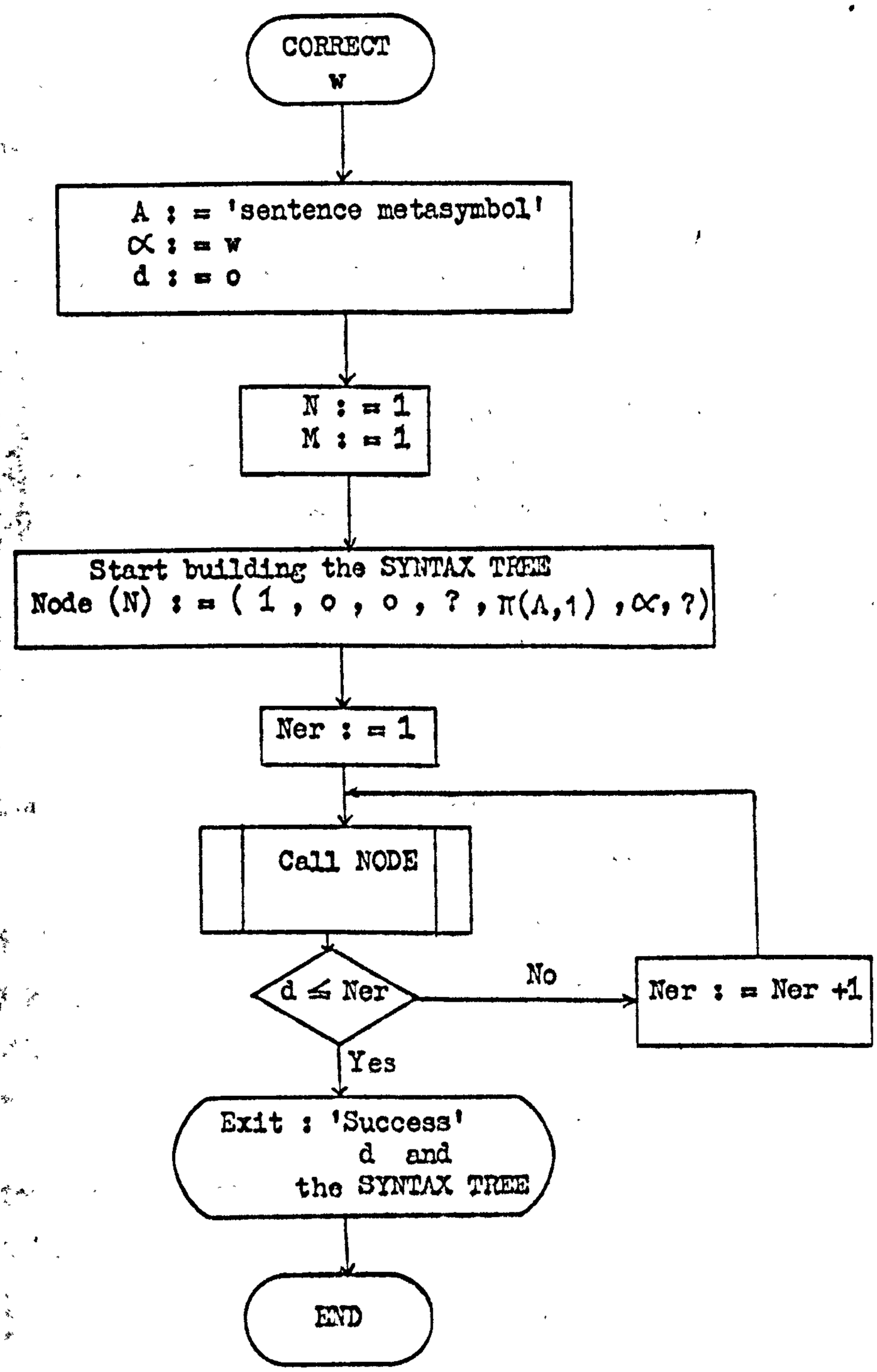


Fig. . 4.4.4

The subroutine NODE builds a very limited part of the syntax tree: the branches descending from the current node (M). 'A'- is the metasymbol corresponding to this node, and α is its goal. Thus the subroutine NODE, must choose an A-production

$$\pi_A : A \longrightarrow u_1 A_1 u_2 A_2 \dots u_l A_l u_{l+1} ,$$

starting with first one, $\pi(A, 1)$. Corresponding to this production, the subroutine NODE must initiate 1 new nodes in the syntax tree. Their father is the current node itself. Also a decomposition Δ_α of its goal α is sought, and the corresponding goals α_i are assigned to the corresponding, just initiated nodes. Also $d(\pi_A, \Delta_\alpha)$ is computed and, if $d + d(\pi_A, \Delta_\alpha)$ is less than Ner , then $d(\pi_A, \Delta_\alpha)$ is added to d. The current node is changed by its son, corresponding to A_1, A becomes A_1, and α becomes α_1. The subroutine NODE is called again. If the current node (M) has no sons, i.e. the production π_A has a terminal string in its right hand side then the next current node must be the younger brother of the father of (M).

If $d + d(\pi_A, \Delta_\alpha)$ is greater than Ner , then a new decomposition of α is sought, and new goals are assigned by NODE to its sons. If there is no new decomposition, then all sons of (M) are eliminated from the syntax tree, and a new A-production is considered. If there is no other production then NODE declares its failure to its father, which must give other goals, i.e. A receives the value Metasymbol (F(M)), where F(M) is the father of M, M receives the value F(M), and α is the goal of F(M). The procedure NODE is called again. In other words the procedure backtracks to a previous node and looks for another decomposition of the goal of that node.

The procedure NODE is given in fig. 4.4.5.

There (M) is the current node, (N) is the last initiated node, A is the metasymbol corresponding to the current node, i.e. A = LHS (Pr (M)), where Pr (M) is the current production used, and LHS (π) means left hand side of the production π. FS (M) is the first son of M , OB (M) is its older brother, and d (M) = $d(Pr(M), \Delta_\alpha)$.

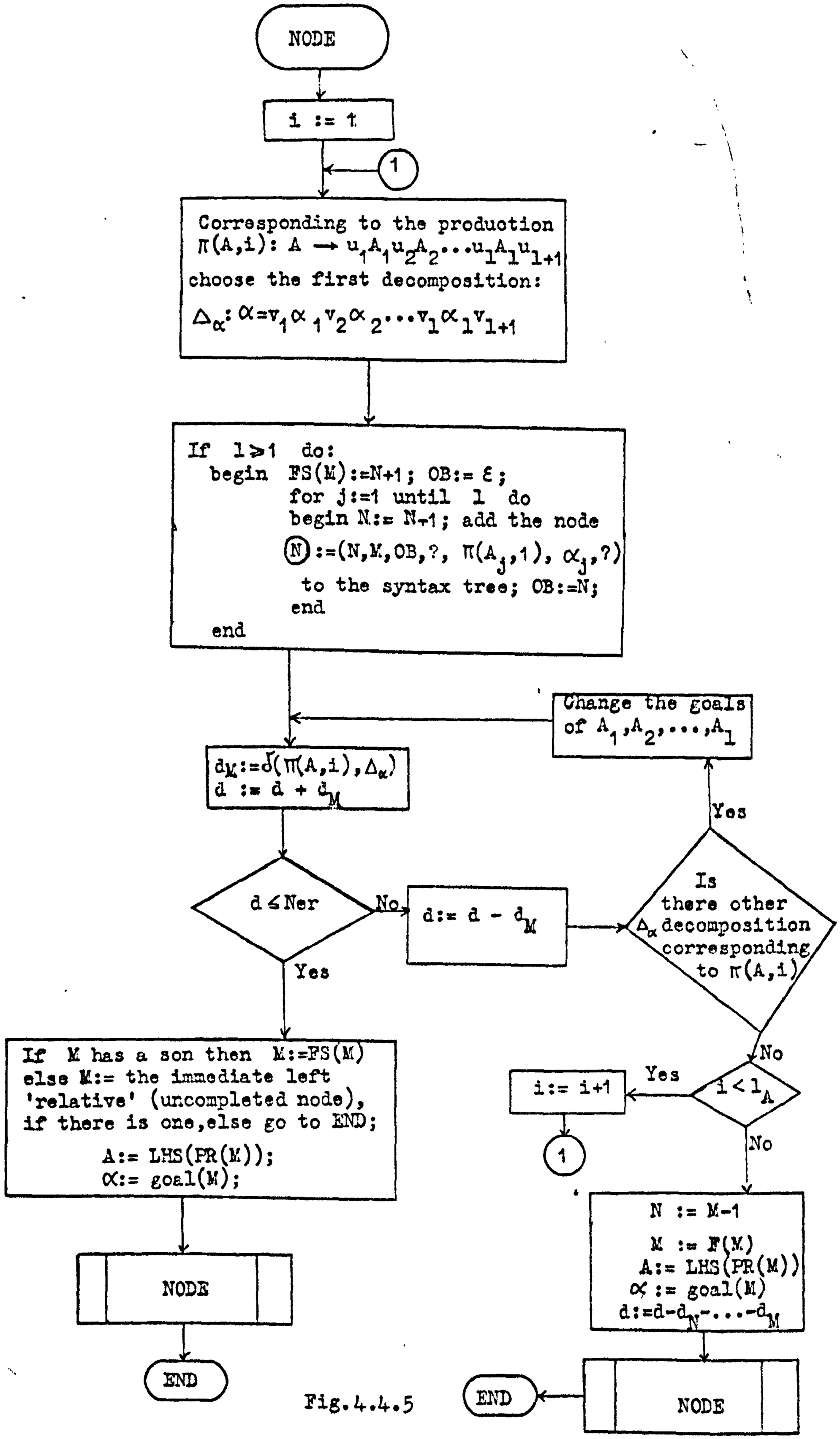

NODE
i := 1
1
Corresponding to the production
π(A,i): A → u1A1u2A2...ulAlul+1
choose the first decomposition:
Δα: α=v1α1v2α2...vlαlvl+1
If l≥1 do:
begin FS(M):=N+1; OB:= ε;
for j:=1 until l do
begin N:= N+1; add the node
(N):=(N,M,OB,?, π(Aj,1), αj,?)
to the syntax tree; OB:=N;
end
end
Change the goals of A1,A2,...,Al
dM:=δ(π(A,i),Δα)
d := d + dM
Yes
d ≤ Ner
No
d:= d - dM
Is there other Δα decomposition corresponding to π(A,i)
No
Yes
i:= i+1
i < lA
1
No
If M has a son then M:=FS(M)
else M:= the immediate left
'relative' (uncompleted node),
if there is one,else go to END;
A:= LHS(PR(M));
α:= goal(M);
N := M-1
M := F(M)
A:= LHS(PR(M))
α := goal(M)
d:=d-dN-...-dM
NODE
END
END

Fig.4.4.5

4.5. SUMMARY

Some algorithms for correcting sentences in three classes of languages have been presented in this chapter.

The global top-down error correcting parser presented in section 4.4 is the most general one, valid for any context free language. Being so general there is a need to improve its efficiency, and this can be done by combining it with other, more efficient error correcting parsers. After all, any A-phrase of the grammar $G = (N, T, P, S)$, can be viewed as a sentence of the grammar $G(A) = (N, T, P, A)$, as has previously been underlined in section 2.1. Thus, the importance of error correction in a regular language is evident, even if it is known that a programming language is not a regular one.

A simple precedence error correcting parser can also be used to improve the efficiency of the global top-down error correcting parser. This will be shown in the next chapter. But the simple precedence error correcting parser, presented in section 4.3, has its own importance, since there are simple precedence programming languages (EULER, for example, see Wirth and Weber [W4]).

V. THE DESIGN OF AN ERROR CORRECTING PARSER FOR BASIC-LIKE LANGUAGES.

In this chapter we are going to design a global error correcting parser for a context free language. To do this, the methods of analysing and correcting strings in a language, described in the previous chapter, are used.

Then a partial implementation of this parser is described and discussed. Some results and the programs are given in appendices.

As often happens, a real implementation is a trade-off between the theory and some practical constraints. In trying to implement an error correcting parser for a programming language two important factors must be taken into account: the speed of analysing strings, and the memory required for coding the algorithm. In some cases the available memory does not allow coding of all the parsing methods used in our design. Nevertheless, if there is enough memory available for coding, the designed algorithm may be implemented, because it may offer a speed advantage.

The global error correcting parser designed in this chaptor has been partially implemented and this is discussed in the second part of section 5.2. However, the only practical constraint that prevented full implementation was the short time available for coding and debugging the program.

5.1. THE GENERAL AND PARTICULAR FEATURES OF BASIC

Although the BASIC language is specifically referred to in the title of this chapter, this work could have been applied to many other programming languages. Nevertheless all examples are taken from BASIC, and the title expresses this fact.

The first important characteristic of BASIC is the presence of delimiters at the beginning and at the end of each statement. This fact permits one to analyse the correctness of each statement.

The syntax of BASIC programs is given in appendix 1a. There are two variants presented and both are very simple for parsing. The terminals for this language are the statements, shown in the appendix.

The language of all correct statements is described by the syntax in appendix 1b. The terminals of this language (26 letters, 10 digits, and the other special characters) are the basic symbols that may be corrupted by the noisy channel. This is the reason that one does not consider a lower level of analysis: the lexical analysis. Very often there is not enough contextual information to correct errors at the lexical level. That part of this language which is used by compilers and interpreters in lexical analysis is given in appendix 1c.

The language in appendix 1a is a very simple one, and easy to parse. This language does not have sufficient redundancy to permit error correction. We can acquire some clues from

a) the dependency between FOR and NEXT statements.

b) the END statement, which must always be the last one in the program.

c) some redundancy which exists between GOSUB and return statements.

d) the second variant in appendix 1a requires that the DIM and DEF statements must be at the boginning of the program.

But this information is not sufficient to correct errors. For example, although a NEXT-statement can be detected as missing, we do not know where it must be inserted. However, missing, inserted, or changed statement are errors that are very improbable, and it is almost impossible to correct them, using only the syntax.

The level at which the majority of errors are found and many of them can be corrected is the language of the "statements", given in appendix 1b. This is the language used by an interpreter, and from what has been said above one notes that the majority of errors correctable by a compiler are correctable by an interpreter. The error correction in the language L is discussed in what follows.

The language L is neither a regular one, nor a simple precedence language. It is not a regular language since it contains arithmetic expressions which are not regular expressions. It is easy to prove that the set

$$L = \{ a^n \, b \, e^n \mid n \geq 1 \},$$

is not a regular language. Also L is not a simple precedence language since the relations $= \dot{=} e$, and $= \gtrdot e$, both hold. Sometimes it is possible to modify the grammar into a simple precedence one, but we are not studying this problem here.

Although L is neither a regular nor a simple precedence language, it is a finite union of such languages. Note that the set of all DIM statements is a simple precedence language (this language was used as an example for section 4.3)

The lexicon of L can be described as in appendix 1c. The use of this lexical language permits one to reduce the length of the string which is analysed. The difficulty in analysing lexical words using the syntax of appendix 1c, is the absence of any delimiters to mark the beginning and the end of the strings. An important

characteristic of BASIC, although not good from the parsing point of view, is the free usage of the blank space. Although this language is used in the lexical analysis phase of compilers, it is difficult to use it in error correction. In fact the errors occur at the symbol level, with some missing, inserted or changed symbols. One believes that the language L must be built on these symbols, and not on the elements of the lexical language.

In the structure of BASIC, and many other programming languages, a major role is played by the reserved words, such as DIM, LET, READ, etc. ... . Always, when one is scanning a program, these are the words that help to identify immediately the type of statement encountered. Thus an initial top-down error-correcting parser seems natural.

5.2. STRUCTURAL DESIGN OF AN ERROR CORRECTING PARSER FOR BASIC

Some algorithms for error-correction in various classes of languages have been presented or reviewed in previous chapters. To implement a practical error correcting parser for BASIC, we have combined some of them, to improve the speed of analysing and correcting input strings.

In the section 4.4. we have developed a pure top-down error-correcting parser. It can analyse and correct strings in any context-free language Nevertheless, being so general, and by verifying a large number of decompositions of the string, the algorithm is an impractical one. Therefore it must be improved, and this can be done by combining it with some special parsers, for subclasses of languages easier to parse.

The most important part of the global top-down error correcting parser, was the subroutine NODE, that calls itself recursively. This subroutine was described in section 4.4. and its main task is to analyse the goal α of a metasymbol A as an A-phrase. Since, for several metasymbols 'A' of a grammar $G = (N , T , P , S)$, the grammars $G(A) = (N , T , P , A)$ are special ones (regular, simple precedence), it is suitable to analyse the goal α in this special grammar $G(A)$. Therefore a switch from top-down parsing to other methods of parsing, is indicated.

We must decide first which methods of parsing are to be considered for our design. Here, the four methods presented in chapter four are considered. Thus, for all metasymbols A, the type of parser required for the grammar G(A) must be known. This is given or computed, when the parser is designed. Let Kind (A) be defined as follows:

$$\text{Kind (A)} = \begin{cases} 1\ , & \text{if a simple precedence parser is used for analysing strings in } L(G(A)) \\ 2\ , & \text{if a parser for the regular language } G(A) \text{ is used to analyse a string in } L(G(A)) \\ 3\ , & \text{if a transducer has been built, for analysing strings in } G(A) \\ 4\ , & \text{otherwise.} \end{cases}$$

A program that checks if the grammar G (A) is a simple precedence grammar, is given in appendix 2a. In fact this program computes all precedence relations between the symbols of G (A) and gives the precedence matrix, if the grammar G(A) is simple precedence grammar. The result of running this program when the grammar is the grammar of appendix 1b, and A is the metasymbol corresponding to arithmetic expressions, is given in appendix 2b.

Let us turn our attention to the algorithm for analysing and correcting an input string w . The analysis starts in a top-down manner, and the procedure NEWNODE is called, as can be seen in fig. 5.2.1. This procedure checks Kind (A) to decide in which way the goal of the metasymbol A must be analysed. According to the result, it calls one of those four error-correcting parsers described in the fourth chapter. This can be seen in fig. 5.2.2.

The PRECEDENCE subroutine takes the goal α of the metasymbol A, and tries to analyse it in the simple precedence grammar G(A) , as has been described in section 4.3.

The REGULAR subroutine analyses the goal α of the metasymbol A, in the regular grammar G (A) as has been shown in section 4.2.

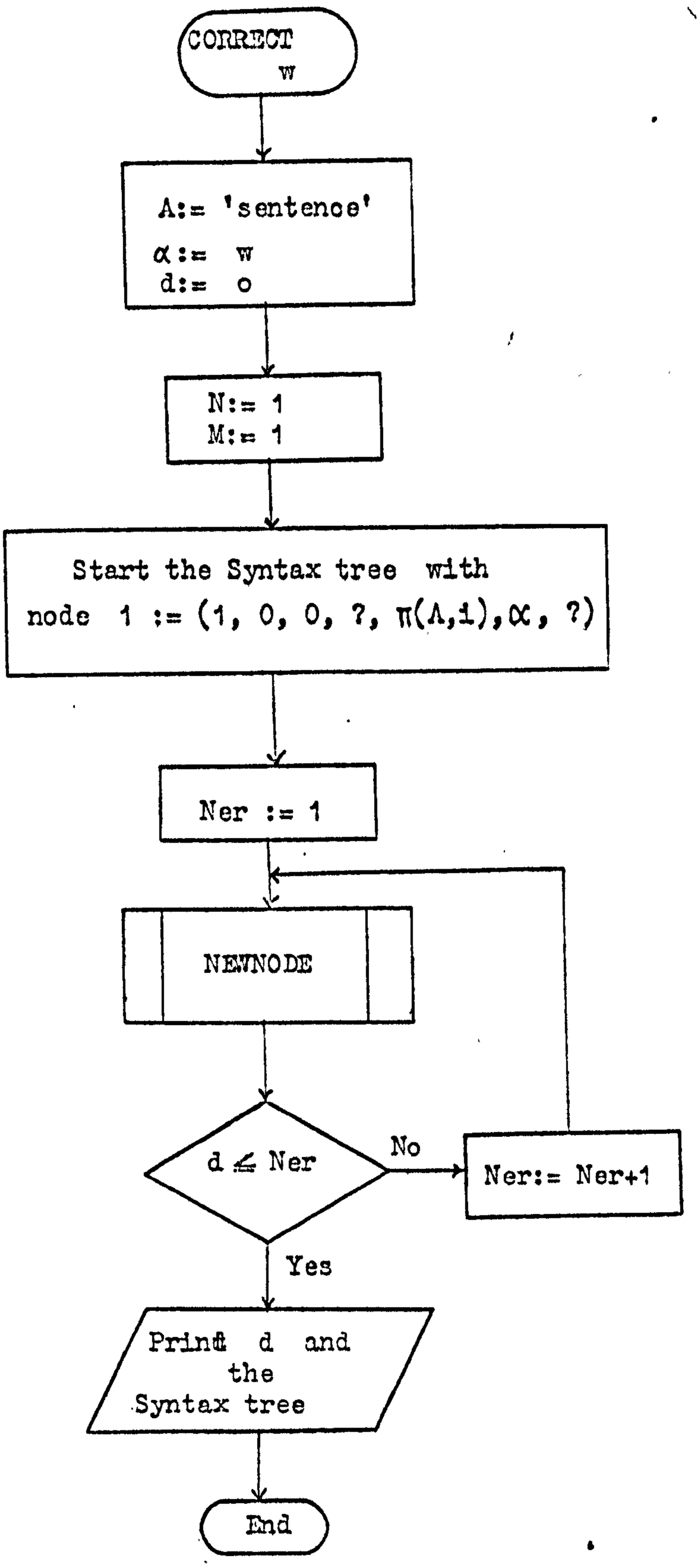

CORRECT
w
A:= 'sentence'
α:= w
d:= o
N:= 1
M:= 1
Start the Syntax tree with
node 1 := (1, 0, 0, ?, π(Λ,i),α, ?)
Ner := 1
NEWNODE
d ≤ Ner
No
Ner:= Ner+1
Yes
Print d and
the
Syntax tree
End

Fig.5.2.1

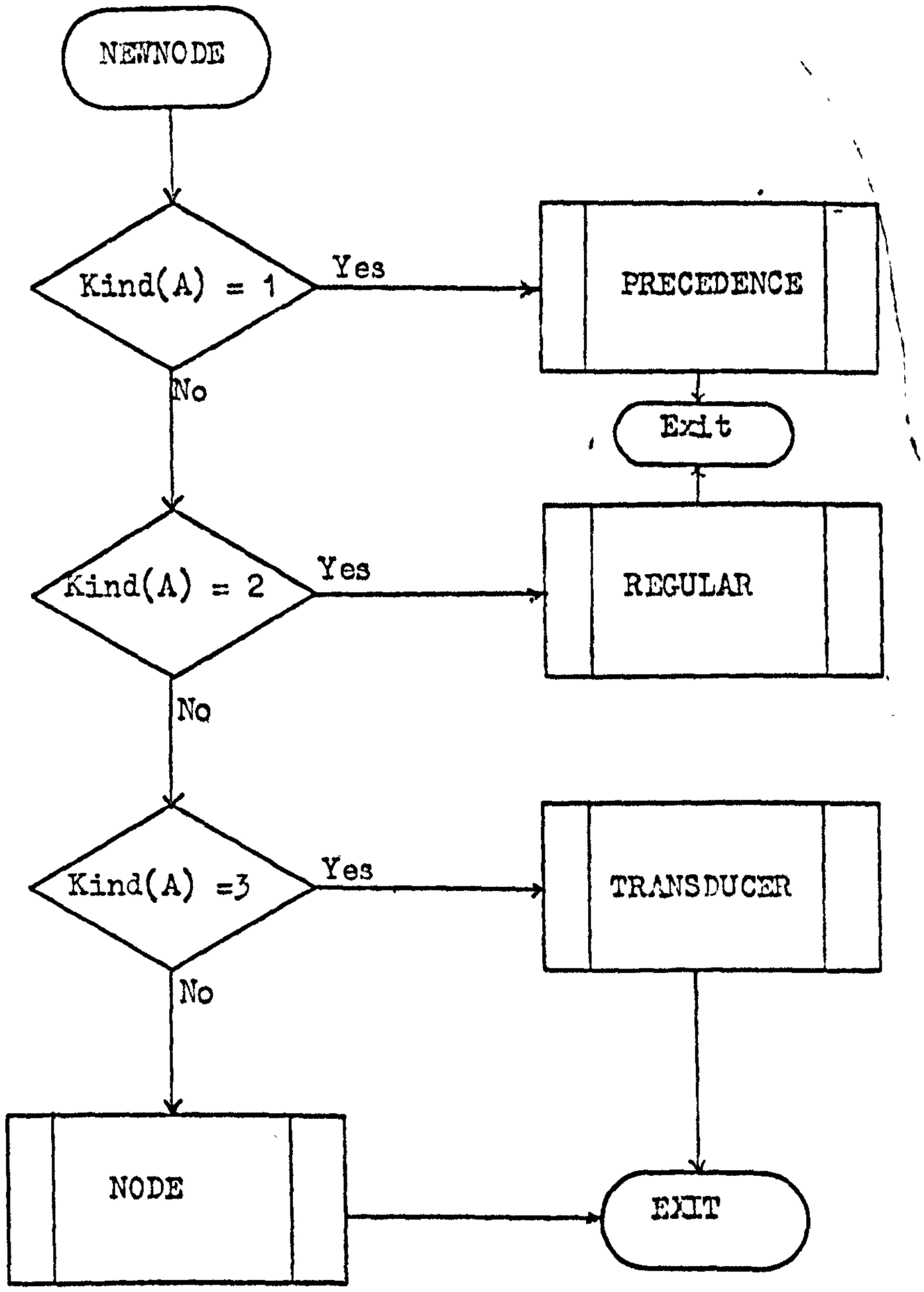

NEWNODE
Kind(A) = 1
Yes
PRECEDENCE
No
Exit
Kind(A) = 2
Yes
REGULAR
No
Kind(A) =3
Yes
TRANSDUCER
No
NODE
EXIT

Fig. 5.2.2

The TRANSDUCER subroutine transforms the input string α, into a sentence of L (G (A)), as described in section 4.1.

Finally, the subroutine NODE, the same as in section 4.4., continues to analyse the goal α of the metasymbol A, in a top-down manner.

The global top-down error correcting parser, described above, has been partially implemented as a FORTRAN program. This program is given in appendices 3 and 4.

The top-down parser must start with the sentence metasymbol so and the goal w, and it must choose an so-production. Instead of generating all decompositions of w for each so-production, the reserved words are used to find the most likely one. The algorithm for finding the distance between two strings, described in section 3.1., is used for this purpose. The subroutine DISTAN (appendix 3) implements this algorithm. When the corresponding reserved word is misspelled by one or two errors (even more for a longer word), there exists a big chance of success. Certainly if an intended reserved word is replaced by another reserved word, this method fails. In a complete implementation these errors may still be corrected, but the price paid for doing it is much higher, and the real gain is negligible, since these errors are very improbable. Moreover, any input string w can be transformed into a correct BASIC sentence by adding only three letters, since

REMw

is always a correct sentence of BASIC.

For this reason, the top-down part of the program attempts to determine the type of statment most similar to the input string (the subroutine MATCH). This has been done, and some results are given in fig. 5.2.3.

```
      DJIM K3(900),Q2(100),R4(100)#
misspeling! probably the sentence production is..
      D I M ld

      DEFFNG(Y)=Y*Y-7.09*Y+7##
      the sentence production is:
      D E F F N l f2

      OOR J=3TU34STIP4#
misspeling! probably the sentence production is..
      F O R afT O t2

      RAD  T(I),U(I,J)#
misspeling! probably the sentence production is..
      R E A D rl

      GOTUB90#
misspeling! probably the sentence production is..
      G O S U B i

      LESGB(5*H+3)=R-P#
misspeling! probably the sentence production is..
      L E T a#

      EMD #
misspeling! probably the sentence production is..
      E N D

      NE.XT V 3#
misspeling! probably the sentence production is..
      N E X T v

      REFURN#
misspeling! probably the sentence production is..
      R E T U R N

      GOTU50#
misspeling! probably the sentence production is..
      G O T O i

      PRUNT 'ABCDEF:',T-0#
misspeling! probably the sentence production is..
      P R I N T p1

      RETORE#
misspeling! probably the sentence production is
      R E S T O R E

      SROP#
misspeling! probably the sentence production is..
      S T O P
```

Fig. 5.2.3

Once that the type of the statement has been found , the parser (subroutine DIRIJ , in appendix 3) calls the other procedures of correction used in the design.

TRANSDUCER has been used only for DATA-lists , as described in section 4.1. This subroutine (called TRANSD in the program of appendix 3), transforms the strings following the reserved word DATA as can be seen in fig. 5.2.4.

Many parts of the language are regular expressions . However , since error correction in a regular language , described in section 4.2., does not raise any special implementation problems , it has not been tested.

Finally the subroutine PRECEDENCE (given in appendix 4) attempts to implement the global simple precedence error correcting parser described in section 4.3. The recovery part of the error correcting parser is called when the scanning phase has been completed , and the whole input string has been analysed and condensed. Unfortunately, this subroutine has not been competely debugged. The scanning phase (i.e. the implementation of the flowcharts given in fig.4.3.3. and fig.4.3.4) analyse a correct sentence, or condense any input string according to the reduction rules of the grammar, as can be seen in fig.5.2.5., and 5.2.6., respectively.

We already know (from chapter 4) how the errors must be corrected, and the partial results discussed above confirmed this. However , the error correcting parser designed in this section still remain to be completely implemented and tested in an environment of BASIC users.

In implementing the algorithm, the speed of parsing correct sentences remains the same as for the non-error correcting parser , which is a very important feature of the error correcting parser . But a price is paid by having an error correcting parser , because

```
      DAATA 44.87,87123,986N.45#
misspeling! probably the sentence production is..
      D A T A da

nr:   44.87
nr:   87.
nr:   23
nr:   986
nr:   .45

      DYTA23,981/,,675,221.98,--98#
misspeling! probably the sentence production is..
      D A T A da

nr:   23
nr:   981
nr:   675
nr:   221.98
nr:   -98

      D.ATA11444444.-998E432,89755#
misspeling! probably the sentence production is..
      D A T A da

nr:   11444444.
nr:   -998E43
nr:   89755

      D3TAA ;Z3.E-09123,F456#
misspeling! probably the sentence production is..
      D A T A da

nr:   3.E-09
nr:   123
nr:   456

      DACA  -8.76,.675E-04,56#
misspeling! probably the sentence production is..
      D A T A da

nr:   -8.76
nr:   .675E-04
nr:   56
```

Fig. 5.2.4

```
        LES68(5*H+3)=R-P$
misspeling! probably the sentence production is...
        L E T as
    let
stack: #  [  0
 stack: #  [  1  8
stack: #  [  v  (  [  5
 stack: #  [  v  (  [  i0
 stack: #  [  v  (  [  i
 stack: #  [  v  (  [  n
 stack: #  [  v  (  [  c
  stack: #  [  v  (  [  fn
  stack: #  [  v  (  [  mf
  stack: #  [  v  (  [  t0  *  [  H
  stack: #  [  v  (  [  t0  *  [  v
  stack: #  [  v  (  [  t0  *  [  ar
  stack: #  [  v  (  [  t0  *  [  fa
  stack: #  [  v  (  [  t0  *  mf
  stack: #  [  v  (  [  t0
  stack: #  [  v  (  [  t
  stack: #  [  v  (  [  e0  +  [  3
  stack: #  [  v  (  [  e0  +  [  i0
  stack: #  [  v  (  [  e0  +  [  i
  stack: #  [  v  (  [  e0  +  [  n
  stack: #  [  v  (  [  e0  +  [  c
  stack: #  [  v  (  [  e0  +  [  fn
  stack: #  [  v  (  [  e0  +  [  mf
  stack: #  [  v  (  [  e0  +  [  t0
  stack: #  [  v  (  [  e0  +  t
  stack: #  [  v  (  [  e0
  stack: #  [  v  (  t  )
  stack: #  [  ar  =  [  R
  stack: #  [  ar  =  [  1
  stack: #  [  ar  =  [  v
  stack: #  [  ar  =  [  ar
  stack: #  [  ar  =  [  fn
  stack: #  [  ar  =  [  mf
  stack: #  [  ar  =  [  t0
  stack: #  [  ar  =  [  t
  stack: #  [  ar  =  [  e0  -  [  P
  stack: #  [  ar  =  [  e0  -  [  1
  stack: #  [  ar  =  [  e0  -  [  v
  stack: #  [  ar  =  [  e0  -  [  ar
  stack: #  [  ar  =  [  e0  -  [  fn
  stack: #  [  ar  =  [  e0  -  [  mf
  stack: #  [  ar  =  [  e0  -  [  t0
  stack: #  [  ar  =  [  e0  -  t
  stack: #  [  ar  =  [  e0
  stack: #  [  ar  =  e
  stack: #  [  as
```

Fig.5.25

```
          DOM L(20,33+,F(97)#
misspeling! probably the sentence production is..
          D I M ld
          dim

stack: #  [  L
stack: #  [  v  (  [  2
stack: #  [  v  (  [  i0 0
stack: #  [  v  (  [  i0
stack: #  [  v  (  i  ,  [  3
   detection point: 12   DOML(20,33+,
stack: #  [  v  (  i  ,  [  i0 3  ?  ,  [  P
stack: #  [  v  (  i  ,  [  i0 3  ?  ,  [  v  (  [  9
stack: #  [  v  (  i  ,  [  i0 3  ?  ,  [  v  (  [  i0 7
stack: #  [  v  (  i  ,  [  i0 3  ?  ,  [  v  (  [  i0
stack: #  [  v  (  i  ,  [  i0 3  ?  ,  [  v  (  i  )
stack: #  [  v  (  i  ,  [  i0 3  ?  ,  a


          DJIM K3(900).Q2/100)#
misspeling! probable the sentence production is..
          D I M ld
          dim

stack: #  [  K  3
stack: #  [  v  (  [  9
stack: #  [  v  (  [  i0 0
stack: #  [  v  (  [  i0 0
stack: #  [  v  (  [  i0
   detection point: 13   DJIMK3(900).Q
   detection point: 16   DJIMK3(900).Q2/1
stack: #  [  v  (  i  )  ?  Q  2  ?  1
stack: #  [  v  (  i  )  ?  Q  2  ?  1  ]  0
stack: #  [  v  (  i  )  ?  Q  2  ?  1  ]  0  ]  0
stack: #  [  v  (  i  )  ?  Q  2  ?  1  ]  0  ]  (  ]  )
```

Fig.5.2.6

much more memory is required for coding the algorithm. The other important fact that came out from our implementation is the inadequacy of the FORTRAN language for dealing with sets, strings, lists, and other data structures. Certainly this fact contributed to the need for looking only at partial implementations within the time span available for this project.

VI CONCLUSION AND FUTURE WORK

> I possessed the book for years before I could make out what it meant. — Indeed, I did not understand it until I had myself independently discovered most of what it contained.
>
> Bertrand Russell

6.1. WHAT ARE THE RESULTS OF THIS THESIS?

We have investigated in this thesis the problem of error correction in some classes of languages. Various error correcting parsers for different classes of languages have been presented. Then the design and implementation of an error correcting parser for BASIC, based on these error correcting parsers, has been described and evaluated.

In dealing with programming languages we have studied their mathematical model: the class of context-free languages. This class is too general, and many subclasses of languages, with more attractive properties, are known. The simplest one, the class of regular languages, is not good enough to represent any existing programming language. But there exists a simple precedence programming language (EULER, for example [W3, W4]). Thus the problem of error correction in a simple precedence language has its own importance. Moreover, the algorithm for parsing a sentence in a simple precedence language is a very simple one.

The main results of this thesis can be found in the fourth chapter. First is the global error correcting parser for a simple precedence language. Second is the global top-down error correcting parser presented in section 4.4. and designed in section 5.2.

The problem we attempted to solve is how to use the whole information available from an input string in deciding how to correct the errors. Both error correcting parsers presented here give a solution to this problem. At the same time they proceed at the same speed as the original parsers in analysing correct sentences. This is so, because both parsers have two modes; a "standard mode" and an "error correcting mode." The standard mode contains the original parser, and analyses an input string until an error is encountered. Thus, if no error is detected, the sentence is analysed using only this mode, in the same way as the original parser.

The global simple precedence error correcting parser has two phases. A "scanning phase", that scans the input strings, detects simple phrases, and condenses them to corresponding metasymbols. Also the errors are detected, reported, and the detection points are marked. Then a "correcting phase" analyses the information stored in the stack and decides how to correct the errors. The first phase analyses any correct sentence and outputs its syntax tree, such that no price is paid (from the speed point of view) in having an error correcting parser (see section 4.3., pp. 63) .

The same can be said about the global top-down error-correcting parser. It analyses a correct sentence at the same speed as Unger's global top-down parser does. The only difference in implementation is in the size of memory; additional storage is required for coding the algorithm for error correction. In fact the error correcting parser includes Unger's global top-down parser (incase Ner=0)(see section 4.4).

At the end we can say that the problem of error correction is not a theoretical problem anymore, from the syntax correctness point of view. From the work of Aho-Peterson [A1], Levy [L7], Lyon [L10], and in this thesis, it results that there always exists an algorithm for correcting errors in a context-free language. That is, for any

input string w, there exists a sentence w_o in the language, and w_o can be found, such that

$$d(w , w_o) = \min_{u \in L} d(w,u),$$

where $d(w,u)$ is the distance between the strings w and u.

It was clear from the beginning that the problem can be solved in a finite number of trials (even if this number is very large). Such a solution is often called "trivial" in mathematics.

Nevertheless the problem is very important from the implementation point of view, for any error correcting compiler. There the speed of correction, and often the memory required for the algorithm, are important in the sense that alternative implementations may be assessed on the basis of these parameters.

We have not defined what an "optimal" error-correcting parser is. Interest was confined to the design of an acceptable one. First, a global simple precedence error correcting parser has been presented in section 4.3. This parser attempts to analyse the context of the entire string, and to use it in deciding how to correct the errors. Nevertheless, the parser tries to correct the errors as early as possible when no ambiguities arise. Then this parser together with other methods of error correction are used to improve the global top-down error-correcting parser for Basic, presented in the fourth and fifth chapters.

6.2. WHAT REMAINS TO BE DONE?

As Turing has said: "we can only see a short distance ahead." The whole subject of context-free languages and error correction in such a language is a relatively recent one. This thesis presents what is known to date on the subject. Although there still remain many things that can, and need to be done, now we can see a little further.

In trying to implement an error correcting parser for BASIC (using the FORTRAN language for programming) some very important problems were encountered.

In mathematics a problem is considered solved when an algorithm to solve this problem is found. We can all agree that this is a very important step. Nevertheless the existence of such an algorithm does not solve the problem completely. There is a need to write a program (software) to implement the algorithm. However, the way of writing software free of bugs is not the main problem. Books and articles about structured programming exist which attempt to give an answer to it (for example, Dahl, Dijkstra, and Hoare [D3]). Rather, if one is going to implement the work as done in this thesis, one must underline the need for better high-level languages. Often one works with sets, lists, or many other data structures. But it is an unpleasant task to write FORTRAN programs dealing with sets. An attempt to develop a set-theoretic high-level language is presently being investigated at Brunel University (see Pollard [P3]), and this might help with the future work in this field.

Certainly there are other things that can be done to optimise and implement different error correcting parsers. But it is felt that it is worth while to think about other related problems. These are the so called "Artificial Intelligence" problems. One is automatic program generation from some acceptable specifications.

A second is proving program correctness [B3, N4]. Also, studying programming languages, and studying natural languages are, somehow, related problems. A result in one can help in the other. Work on understanding natural languages has been done by Winograd [W5], or Marcus [M1].

We have seen that minimum distance error correction is not the best technique. Our purpouse was the syntax correctness of the program. Nevertheless a program that is syntactically correct is not always a correct program. Man-machine communication has been studied, but we have not used the fact that the information transmitted is meaningful. Thus, the natural continuation to our work would be a theory of semantic communication (see for example, Belis [B1]).

6.3. FINALLY, WHAT IS THE REAL NATURE OF THE PROBLEM ?

Cel ce gindeste singur si scormone lumina
A dat o viata noua si-un om de fier, masina,
Fiinta zamislita cu gindul si visarea,
Ne-nchipuit mai tare ca bratul si spinarea.

Tudor Arghezi

(Translated on page 161)

It is known that there are many grammars for the same language. It may be of some interest that a programmer does not think of a particular grammar when he corrects his program. He has learned the language, and he recognises and corrects it almost instantly. Also, when a child learns a language, he learns first the meaning of words, of what he hears and says. Later in school , after many years, he is taught "the grammar" of his language.

The problem that arises is how can one describe the process of learning, and how can one model it ? Thinking of these , and other "artificial intelligence" problems, and how to use computers to solve them, the questions: "what is intelligence? ", or "how does a machine store knowledge?", arise naturally. If a machine already can store information and can use it in making decisions (much more quickly and more precisely than a man can), one can say that it has some "intelligence".

There is a great deal of research in this direction. At Brunel University, for example, the "intelligent automaton group" study the property of adaptive networks (see for example [A4,A5,F2,W5]).
In [A5] Aleksander said: "Machines are now being built which for the first time exhibit "intelligent" behaviour at an 'economical price. They are considerably different from conventional computers , accumulating experience from their environment and scanning it in a manner analoguous to human thinking" and "concepts such as learning

of speech, language and music become clearly feasible".

In this work something has been learned about context free languages, the process of sentence recognition in such a language has been studied, and a global top-down error correcting parser for BASIC has been designed and partially implemented. Many other interesting problems have been met, and we have asked various other questions. However, one can say that the problem of error correction treated in this thesis is not an isolated problem, and the process of gaining knoledge is not a single problem, but a succesive entering of different related areas, at different levels of understanding.

The author's experience has been one of having his curiosity aroused by asuccesion of related fields, which, when entered , generate new questions and a desire to enter new fields. This, however, is no different from the human curiosity for acquiring knowledge. How to describe and understand the process of learning, its arousal and stimulation ? This is the most general "problem".

BIBLIOGRAPHY

A1. A.V.Aho,and T.G.Peterson, A minimum distance error correcting parser for context-free languages, SIAM J.on Computing,1(1972),305-312.

A2. C.N.Alberga, String similarity and misspellings, CACM,10(1967),5,302-313.

A3. I.Aleksander,and F.K.Hanna, Automata theory: an enganeering approach, Computer System Engineering Series, Crane Russak,1976.

A4. I.Aleksander, Artificial intelligence, Electronics and Powers, 22(1976),4,242-244.

A5. I.Aleksander, Electronics for Intelligent machines, New Scientist and Science Journal, 11.III.1971.

B1. M.Belis, A theory of semantic communication, Proceedings of the 3r World Congress of Cybernetics, Bucharest, 1975.

B2. C.R.Blair, A program for correcting spelling errors, Information and Control, 3(1960),60-67.

B3. R.M.Burstall, Some Techniques for proving correctness of Programs which alter Data Structures, Machine Intelligence, 7(1972),23-50.

C1. T.E.Cheatham,and T.Standish, Optimisation aspects of compiler-compilers, ACM Sigplan Notices,5(1970),10,10-17

C2. N.Chomsky, Three models for the description of language, IRE Trans.Inform.Theory, IT2(1956),113-124.

C3. N.Chomsky,and G.A.Miller, Finite state languages, Information and Control, 1(1958),91-112.

C4. R.W.Conway,and T.R.Wilcox, Design and implementation of a diagnostic compiler for PL/I, CACM, 16(1973),3,169-179.

D1. F.Damerau, A techinque for computer detection and correction of spelling errors, CACM,7(1964),3,171-176.

D2. L.Davidson, Retrieval of misspelled names in an airlines passenger reservation system, CACM,5(1962),3,169-171.

D3. O.-J.Dahl,E,W.Dijkstra,and C.A.R.Hoare, Structured programming, Academic Press, 1972.

D4. R.K.Donnely, An overview of associative processing for non-numerical applications, Technical Rep. C/R/039, Brunel Univ., 1976.

D5. J.Dyke,and M.Lea, An associative parallel processor for local editing applications, Digital Processes, 1(1975),2,89-101.

E1. J.Earley, An efficient context-free parsing algorithm, CACM 13(1970),2,94-102.

E2. B.Elspas,M.V.Green, and K.N.Levitt, Software reliability, Computer, 1(1971),1,21-27.

F1. J.Feldman,and D.Gries, Translator writing systems, CACM, 11(1968),2,77-113.

F2. C.J.Fernandez, Adaptive sequence recognition with memory elements PhD Thesis,Brunel University,(to be submitted).

F3. R.W.Floyd, Syntactic analysis and operator precedence, JACM, 10(1963),7,316-333.

F4. D.N.Freeman, Error correction in CORC- the Cornell computing Language, Proc.AFIPS,26(1964),15-34(or in EF23 pp.233-304).

F5. M.Frentiu, Error correcting codes, Minithesas, Brunel University,1975 (unpublished).

G1. S.Ginsburg, The mathematical theory of context-free languages, McGraw-Hill,Inc.,1966

G2. S.L.Graham,and S.P.Rhodes, Practical syntactic error recovery, CACM, 18(1975),11,639-650.

G3. D.Gries, The use of transition matrices in compiling, CACM, 11(1968),1,26-34.

G4. D.Gries, Compiler construction for digital computers, Wiley,1971.

H1. F.R.A.Hopgood, Compiling techniques, MacDonald,1969.

I1. E.T.Irons, A syntax directed compiler for Algol 60, - CACM,4(1961),51-55.

I2. E.T.Irons, An error correcting parse algorithm, CACM, 6(1963), 11,669-673.

J1. E.G.James,and D.F.Partridge, Adaptive correction of progran statements, CACM,16(1973),1,27-37.

L1. J.E.LaFrance, Optimisation of error recovery in syntax-directed parsing algorithms, SIGPLAN Notices,5(1970),12,2-17.

L2. J.E.LaFrance, Syntax directed error recovery for compilers, PhD Thesis, U.of Illinois,Urbana,Comp.Sci.Dep.,1971.

L3. R.Lawrance,and R.A.Wagner, An extension of the string-to-string correction problem, JACM,22(1975),2,177-183.

L4. J.A.N.Lee, Computer semantics, Van Nostrand Reinhold Company, 1972.

L5. J.A.N.Lee, The formal definition of the BASIC language, The computer Journal ,15(1972),1,37-41.

L6. R.P.Leinius, Error detection and recovery for syntax directed compiler systems, PhD Thesis, Computer Sci.Dep., Univ.Of Wisconsin, 1970.
(Abstract in Dissertation Abstract,31(1971),5923-B)

L7. J.P.Levy, Automatic correction of syntax errors, PhD Thesis, Cornell Univ.,Computer Sci.Dep.,1971.
(Abstract in Dissertation Abstract,32(1972),6972-B)

L8. D.Lewin, Theory and Design of Digital Computers,Nelson,1973.

L9. C.R.Litecky,and G.B.Davis, A study of errors,error proneness, and error diagnosis in COBOL, CACM,19(1976),1,33-37.

L10. G.Lyon, Syntax directed least-errors analysis for context free languages: a practical approach, CACM,1,3-14.

K1. D.E.Knuth, The art of computer programming, Addison-Wesley Pub.Comp., 1968.

M1. S.Marcus, Algebraic linguistics; analytical models, Academic Press, 1967.

M2. H.L.Morgan, Spelling correction in system programs,CACM, 13(1970),2,90-94.

M3. P.G.Moulton, and M.E.Muller, Ditran - a compiler emphasizing diagnostics, CACM, 10(1967),1,47-52.

N1. P.Naur,(ed.), Report on algorithmic language ALGOL 60, CACM, 3(1960), 5,299-314.

N2. P.Naur,(ed.), Revised report on the algorithmic language ALGOL 60, CACM,6(1963),1,1-17.

N3. A.Newell,and H.A.Simon, Computer Science as Empiracal Inquiry: Symbols and search, CACM,19(1976),3,113-126.

O1. Al.Ollongren, Definition of programming languages by interpreting automata, Academic Press, 1974

P1. T.G.Peterson, Syntax error detection,correction and recovery in parsers, PhD Thesis,Stevens Inst.of Tech., 1972.

P2. W.W.Peterson,E.J.Weldon, Error Correcting Codes, MIT Press,1972.

P3. B.W.Pollack, Compiler Techniques, Auerbach Computer Sci.series ,1972.

P4. R.Pollard, Set-theoretic high-level languages, PhD Thesis, Brunel University (to be submitted).

R1. E.M.Rieseman,and A.R.Hanson, A contextual postprocessing system for error correction using binary n-grams, IEEE trans. Comp.,C-23(1974),480-493.

S1. W.B.Smith, Error detection in formal languages, J.Comp.Syst. Sci., 4(1970),385-405.

T1. M.G.Thomason, Errors in regular languages, IEEE Trans.on Comp., C-23(1974),6,597-602.

T2. M.G.Thomason,and R.G.Gonzalez, Syntactic recognition of imperfectly specified patterns, IEEE Trans.on Comp.,C-24(1975),1,93-95.

T3. M.G.Thomason, Stochastic syntax-directed translation schemata for correction of errors in context fre languages, IEEE Trans. on El.Comp., EC-24(1975),1211-1213.

U1. J.R.Ullman, A binary n-gramm technique for automatic correction of substitution,deletion,insertion and reversal errors in words, Division of Comp.Sci.,National Physical Laboratory, Teddington,Middlesex,1976.

U2. S.H.Unger, A global parser for context-free phase structure grammars, CACM,11(1968),4,240-246.

W1. R.A.Wagner, Order-n correction for regular languages, CACM, 17(1974), 5,265-268.

W2. R.A.Wagner,and M.J.Fischer, The string to string correction problem, JACM,21(1974),1,168-173.

W3. N.Wirth, A programming language for the 360 Computers, JACM, 15(1968), 1,37-74.

W4. N.Wirth,and H.Weber, Euler- a generalisation of Algol and its formal definition, CACM,9(1966),1-2,13-23 and 89-99.

W5. T.Winograd, Understanding natural language, Edinburgh Press, 1972.

end!

APPENDIX 1.a

Variant 1.

<program> ::= <statement list> <end> <data list>

<statment list> ::= <statment>

::= <statment list> <statment>

::= for-st <statment list> next-st

<statment> ::= goto-st

::= gosub-st

::= if-st

::= let-st

::= input-st

::= read-st

::= print-st

::= rem-st

::= restore-st

::= return-st

::= dim-st

::= def-st

<end> ::= end-st

<data list> ::= (data-st)$^{\infty}_{0}$

Variant 2

<program> ::= (<declaration>)$^{\infty}_{0}$ <statment list> <end>

<data list>

<declaration> ::= dim-st

::= def-st

and all the other metasymbol-productions are the same as in variant 1.

APPENDIX 1.b

```
,0 -- F R I N T pl
s0 -- L E ] qs
-0 -- [ F Do [ H E N t2
,0 -- F O K of T O t2
,0 -- O E X T v
,0 -- R E A D ri
,0 -- U E F F N I f2
,0 -- D A T A da
,0 -- O I M [d
s0 -- R E M st
,0 -- E N D
s0 -- G O T O a
s0 -- G O S U B i
s0 -- S T O P
s0 -- R E T U R N
s0 -- R E S T O R E
s0 -- I N P U T ri
s0 -- ns
pl -- pr
pl -- pr , pl
pr -- ' st '
pr -- e
st -- ch
st -- st ch
as -- ar = e
ar -- v ( e , e )
ar -- v ( e )
ar -- v
Do -- e or e
n2 -- 10
n2 -- s0
at -- v = e
t2 -- e S T E P e
t2 -- e
rl -- ar
rl -- ar , rl
f2 -- ( v ) = e
fu -- fu ( e )
fo -- ( e )
fa -- ar
fa -- c
mf -- fa
mf -- fa ! fa
t0 -- mf
t0 -- t0 * mf
t0 -- t0 / mf
t -- t0
e -- e0
e0 -- e0 ad t
e0 -- ad t
e0 -- t
ld -- ld , a
ld -- a
a -- v ( i0 , i0 )
a -- v ( i0 )
v -- l
v -- l d
i0 -- i
i -- d
i -- i d
du -- ds
du -- do , ds
```

```
53    as --- ad n
64    as --- n
65    or --- <
66    or --- =
67    or --- >
68    or --- < =
69    or --- < >
70    or --- > =
71    fu --- S O N
72    fu --- S I N
73    fu --- T A N
74    fu --- A T N
75    fu --- A B S
76    fu --- C O S
77    fu --- E X P
78    fu --- F N I
79    fu --- I N T
80    fu --- L O G
81    fu --- R N D
82    ad --- +
83    ad --- -
84    n  --- 1
85    n  --- . 1
86    n  --- 1 .
87    n  --- 1 . 1
88    .  --- n he
89    .  --- n
90    he --- E ad d2
91    he --- E d2
92    d2 --- d d
93    d2 --- d
94    d  --- 0
95    d  --- 1
96    d  --- 2
97    d  --- 3
98    d  --- 4
99    d  --- 5
100   d  --- 6
101   d  --- 7
102   d  --- 8
103   d  --- 9
104   1  --- A
105   1  --- B
106   1  --- C
107   1  --- D
108   1  --- E
109   1  --- F
110   1  --- G
111   1  --- H
112   1  --- I
113   1  --- J
114   1  --- K
115   1  --- L
116   1  --- M
117   1  --- N
118   1  --- O
119   1  --- P
120   1  --- Q
121   1  --- R
122   1  --- S
123   1  --- T
124   1  --- U
```

125	l	V
126	J	W
127	l	X
128	l	Y
129	l	Z
130	ch	l
131	ch	d
132	ch	qd
133	ch	qr
134	ch	(
135	ch	)
136	ch	*
137	ch	+
138	ch	,
139	ch	/
140	ch	h
141	ch	j
142	ch	?

APPENDIX 1.c

```
l       ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

d       ::= 0|1|2|3|4|5|6|7|8|9

r       ::= < | > | = | <= | >= | <>

add     ::= + | -

mult    ::= * | /

ch      ::= l | d | add | mult | r | , | . | ( | ) | ; | :

v       ::= ld | l

i       ::= d | dd

no      ::= i | i.i | .i | i.

n       ::= no | no exponent

exponent::= E add d d | E add d | E d d | E d

fu      ::= ABS | ATN | COS | EXP | INT | LOG | RND | SQR |
            | SIN | FN1

w       ::= DATA | DEF | DIM | FOR | TO | STEP | IF | THEN |
            | GOTO | GOSUB | NEXT | LET | INPUT | READ | REM |
            | PRINT | RESTORE | RETURN | STOP | END
```

APPENDIX 2.a

```
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 C,OR,OR1,FIRST,LAST,V3,A,MI,N
      COMMON ME(10,20),NS1(40),NS2(40),META
     1/B1/M(48,46)
     1/B2/V3(60),WS(100),G(100)
     1/B3/NE(40),NLL(40),FIRST(40,60),LAST(40,60)
     1/B4/LG(150),GR(150,10),GR1(150,10)
     1/B5/NP,NRV,NRT,LEN

C     this program builds the precedence matrix for the grammar
C     that has the metasymbols ME(A,2),...,ME(A,ML(A,1))
C     the A-productions are numbered NS1(A),...,NS2(A)
C     there are NP productions and NRT terminals in the grammar
C     the length of the  I-th production is LG(I)
C     NRV  is the number of nonterminals
C     first NRV symbols are nonterminals,last NRT -terminals

      CALL ASSIGN(1,'DAT6.MIL ',0)
      WRITE(7,500)
500   FORMAT(2X,'OUTPUT DEVICE?'/)
      CALL ASSIGN(6,'X ',-1)
      CALL ASSIGN(4,'DAT.MIL ',0)

      WRITE(7,1000)
1000  FORMAT(4X,'  PROGRAM F60:=PRECED.MATRIX'/)

      READ(1,10)NP,LEN,NRV,NRT,META
      WRITE(7,70)NP,LEN,NRV,NRT,META
70    FORMAT(2X,'NP,LEN,NRV,NRT,META:=',6I5)
      WRITE(6,71)
71    FORMAT(2X,'          the grammar: ')
      DO 201 I=1,NP
      READ(1,10)LG(I),(GR(I,J),J=1,LG(I))
201   CONTINUE
10    FORMAT(18I4)
15    FORMAT(18(A2,2X))
      READ(1,10)NS1
      READ(1,10)NS2
      WRITE(7,72)
      WRITE(7,73)NS1
      WRITE(7,73)NS2
72    FORMAT(2X,'   NS1:..NS2: '/(18I4))
73    FORMAT(6X,18I3)
      READ(1,15)(WS(I),I=1,100)
      WRITE(7,74)
      WRITE(7,75)WS
74    FORMAT(5X,' WS::= ')
75    FORMAT(18(2X,A2))
      WRITE(7,76)
76    FORMAT(12X,'ME*:=')
      READ(1,10)((ME(I,J),J=1,18),I=1,6)
      WRITE(7,10)((ME(I,J),J=1,18),I=1,6)
      READ(1,222)V3
222   FORMAT(72A1)
      WRITE(7,233)
223   FORMAT(' V3= ',60A1)
      WRITE(7,223)V3
      WRITE(7,233)
233   FORMAT(1HO,'NEXT'/)
      READ(1,10)C
      WRITE(7,10)C
```

```
      DO 600 I=1,LEN
      DO 600 J=1,LEN
600   M(I,J)=0

      DO 90 I=1,NP
      DO 91 J=1,LG(I)
91    GR(I,J)=C(GR(I,J))
      WRITE(7,902)I,LG(I),(GR(I,J),J=1,LG(I))
90    CONTINUE

902   FORMAT(5X,10I4)
      DO 29 I=1,40
      NLL(I)=0
29    NL(I)=0

      LIM=ME(META,1)
      DO 30 J=2,LIM
      NK1=ME(META,J)
      DO 30 I=NS1(NK1),NS2(NK1)
      WRITE(6,31)NK1,I,(V3(GR(I,K)),K=1,LG(I))
30    CONTINUE
31    FORMAT(4X,2I4,2X,9A2)

      CALL MACROS(GR,NL,FIRST)
C
      DO 610 I=1,NP
      GR1(I,1)=GR(I,1)
      DO 620 J=2,LG(I)
      IND=LG(I)+2-J
620   GR1(I,J)=GR(I,IND)
610   CONTINUE
      CALL MACROS(GR1,NLL,LAST)

C
      DO 40 NCOUNT=1,5
      WRITE(7,229)NCOUNT
229   FORMAT(4X,'NCOUNT=',I6)

      LIMO=ME(META,1)
      DO 45 K1=2,LIMO
      K=C(ME(META,K1))


C       first
C     if B is in FIRST(A) and C is in FIRST(B) then
C     add C to FIRST(A)

      LIM1=NL(K)
      IF(LIM1.EQ.0)GOTO 46
      DO 50 L=1,LIM1
      IND=FIRST(K,L)
      IF(IND.GT.NRV)GOTO 50

      LIM2=NL(IND)
      IF(LIM2.EQ.0)GOTO 46
      DO 52 I=1,LIM2
      A=FIRST(IND,I)
      CALL CHECK(K,NL,FIRST,A)
52    CONTINUE

50    CONTINUE
```

```
C last
C
46      LIM1=NLL(K)
        IF(LIM1.EQ.0)GOTO 45
        DO 62 L=1,LIM1
        IND=LAST(K,L)
        IF(IND.GT.NRV)GOTO 62

        LIM2=NLL(IND)
        IF(LIM2.EQ.0)GOTO 45
        DO 64 I=1,LIM2
        A=LAST(IND,I)
        CALL CHECK(K,NLL,LAST,A)
64      CONTINUE

62      CONTINUE
C
45      CONTINUE
40      CONTINUE
        CALL SCRIE

        CALL MATRIX

C
        WRITE(6,341)
341     FORMAT(1H1,4X,'precedence matrix'//)
        WRITE(6,351)(V3(I),I=1,LEN)
351     FORMAT(3X,18(1X,3A1))
        DO 350 I=1,LEN
        WRITE(4,360)V3(I),(M(I,J),J=1,LEN)
        WRITE(6,360)V3(I),(M(I,J),J=1,LEN)
360     FORMAT(2X,A1,18(1X,3I1))
350     CONTINUE
        END


        SUBROUTINE SCRIE
        IMPLICIT INTEGER (A-Z)
        LOGICAL*1 C,GR,GR1,FIRST,LAST,V3,A,MI,M
        COMMON ME(10,20),NS1(40),NS2(40),META
        1/B1/M(48,48)
        1/B2/V3(60),WS(100),C(100)
        1/B3/NL(40),NLL(40),FIRST(40,60),LAST(40,60)
        1/B4/LO(150),GR(150,10),GR1(150,10)
        1/B5/NP,NRV,NRT,LEN
        WRITE(6,56)

56      FORMAT(1H1,5X,'first & last'//)
82      FORMAT(5X,A1,' : ',6X,25(A1,1X)/(10X,25(A1,1X)))
        DO 584 I=1,NRV
        IF(NL(I).EQ.0)GOTO 584
        WRITE(6,82)V3(I),(V3(FIRST(I,K)),K=1,NL(I))
584     CONTINUE
        WRITE(6,57)
57      FORMAT(/////)
        DO 684 I=1,NRV
        IF(NLL(I).EQ.0)GOTO 684
        WRITE(6,82)V3(I),(V3(LAST(I,K)),K=1,NLL(I))
684     CONTINUE
        RETURN
        END
```

```
      SUBROUTINE MACROS(GRAM,NLG,ARRAY)
C     this subroutine takes B from all productions  A --- B a
C     and put it in FIRST(A,?)

      COMMON ME(10,20),NS1(40),NS2(40),METa
     1/B5/NP,NRV,NRT,LEN
      LOGICAL*1 GRAM(150,10),ARRAY(40,60),A
      DIMENSION NLG(40)
      LIM=ME(METa,1)
      DO 30 J=2,LIM
      NK1=ME(METa,J)
      DO 30 I=NS1(NK1),NS2(NK1)
      IND=GRAM(I,1)
      A=GRAM(I,2)
      CALL CHECK(IND,NLG,ARRAY,A)
30    CONTINUE
      RETURN
      END



      SUBROUTINE CHECK(IND,NLG,ARRAY,A)
      COMMON ME(10,20),NS1(40),NS2(40),META
      LOGICAL*1 ARRAY(40,60),A
      DIMENSION NLG(40)
C     if A  is not already in ARRAY(IND,?) put it there
C     here ARRAY  is  FIRST or LAST  respectively
      M5=NLG(IND)
      IF(M5.EQ.0)GOTO 120
      DO 100 I=1,M5
      IF(ARRAY(IND,I).EQ.A)GOTO 110
100   CONTINUE
120   NLG(IND)=NLG(IND)+1
      ARRAY(IND,NLG(IND))=A
110   RETURN
      END



      SUBROUTINE MATRIX
C     if A is not already in ARRAY(IND,?) put it there
C     here ARRAY is FIRST or LAST respectively
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 C,GR,GR1,FIRST,LAST,V3,A,AL,M
      COMMON ME(10,20),NS1(40),NS2(40),META
     1/B1/M(48,48)
     1/B2/V3(60),WS(100),C(100)
     1/B3/NL(40),NLL(40),FIRST(40,60),LAST(40,60)
     1/B4/LG(150),GR(150,10),GR1(150,10)
     1/B5/NP,NRV,NRT,LEN
C
      LIM=ME(META,1)
      DO 330 J2=2,LIM
      NK1=ME(META,J2)
      DO 330 IC=NS1(NK1),NS2(NK1)
      IF(LG(IC).EQ.2) GOTO 330
      DO 340 J1=3,LG(IC)
      K=GR(IC,J1-1)
      L=GR(IC,J1)
```

```
C       there is a production  A -- K L  a
C       in this case   K = L  and   K  < all  LJ in FIRST(L,?)

        CALL VERIF(M,K,L,1)
        IF(L.GT.NRV.OR.NL(L).EQ.0)GOTO 340
        L1=L
        DO 340 J=1,NL(L1)
        LJ=FIRST(L1,J)
        CALL VERIF(M,K,LJ,2)
340     CONTINUE

330     CONTINUE

        LIM=ME(META,1)
        DO 372 J2=2,LIM
        NK1=ME(META,J2)
        DO 372 IC=NS1(NK1),NS2(NK1)
        IF(LG(IC).EQ.2)GOTO 372
        DO 372 J1=3,LG(IC)
        K=GR(IC,J1-1)
        L=GR(IC,J1)
        IF(K.GT.NRV)GOTO 372
        K1=K
C       since K is a metasymbol for all LK in LAST(K,?)
C       1)  LK > L    and   2)   LK  >  LJ  for all LJ in
C       FIRST(L,?)

        DO 384 J=1,NLL(K1)
        LK=LAST(K1,J)
        CALL VERIF(M,LK,L,3)
        IF(L.GT.NRV)GOTO 384
        DO 386 J9=1,NL(L)
        LJ=FIRST(L,J9)
        CALL VERIF(M,LK,LJ,3)
386     CONTINUE
384     CONTINUE

372     CONTINUE
        RETURN
        END



        SUBROUTINE VERIF(M,ML,MC,MI)
Cc      m(ml,mc) is made  mi  if it is  0
        IMPLICIT INTEGER (A-Z)
        LOGICAL*1 M(48,48),MI,UU,C,V3
        COMMON /B2/V3(60),WS(100),C(100)
        IF(M(ML,MC).EQ.0)GOTO 400
        IF(M(ML,MC).EQ.MI)GOTO 420
        WRITE(6,410)V3(ML),V3(MC),MI,M(ML,MC)
410     FORMAT(5X,'double relation for M(',A1,2X,A1,'),   MI:',I4
       1M=',I4)
400     M(ML,MC)=MI
420     RETURN
        END
```

APPENDIX 2.b

the grammar:

```
 1   48    e x
 7   49    x x + 1.
 7   50    x 1 1.
 7   51    x 1
13   47    1 u
12   38    f u ( e )
12   39    f ( e )
12   40    f r
12   41    f 1
11   71    u S G N
11   72    u S I N
11   73    u T A N
11   74    u A T N
11   75    u A B S
11   76    u C O S
11   77    u E X P
11   78    u F N I
11   79    u I N T
11   80    u L O G
11   81    u M N U
10   42    m f
10   43    m f - f
 6   26    r v ( e , e )
 6   27    r v ( e )
 6   28    r v
32   56    v 1
32   57    v 1 d
26   88    c n
26   89    c n
27   84    n 1.
27   85    n . 1.
27   86    n 1. .
27   87    n 1 . 1.
31   58    1 J
15   44    w m
15   45    w w * m
15   46    w w * m
28   59    J d
28   60    j j d
33  104    1 A
33  105    1 B
33  106    1 C
33  107    1 D
33  108    1 E
33  109    1 F
33  110    1 G
33  111    1 I
33  112    1 L
33  113    1 M
33  114    1 N
33  115    1 O
33  116    1 P
33  117    1 Q
33  118    1 R
33  119    1 S
33  120    1 T
33  121    1 X
```

APPENDIX 2.1

first & last

```
e :  x + t
r :  v 1 A B C D E F G I L M N O P Q R S T X
x :  x + t
m :  f u ( r c S T A C E F I L R v n 1 i . B D G M N O
     P Q X j d
u :  S T A C E F I L R
f :  u ( r c S T A C E F I L R v n 1 i . B D G M N O P
     Q X j d
t :  w m f u ( r c S T A C E F I L R v n 1 i . B D G M
     N O P Q X j d
w :  m w f u ( r c S T A C E F I L R v n 1 i . B D G M
     N O P Q X j d
c :  n i . j d
n :  i . j d
j :  d j
i :  j d
v :  1 A B C D E F G I L M N O P Q R S T X
1 :  A B C D E F G I L M N O P Q R S T X



e :  x t w m f ) r c v h n 1 d i . A B C D E F G I L M
     N O P Q R S T X j
r :  ) v 1 d A B C D E F G I L M N O P Q R S T X
x :  t w m f ) r c v h n 1 d i . A B C D E F G I L M N
     O P Q R S T X j
m :  f ) r c v h n 1 d i . A B C D E F G I L M N O P Q
     R S T X j
u :  N S P I T G D A B C E F I L M O Q R X
f :  ) r c v h n 1 d i . A B C D E F G I L M N O P Q R
     S T X j
t :  w m f ) r c v h n 1 d i . A B C D E F G I L M N O
     P Q R S T X j
w :  m f ) r c v h n 1 d i . A B C D E F G I L M N O P
     Q R S T X j
c :  h n i . j d
n :  i . j d
j :  d
i :  j d
v :  1 d A B C D E F G I L M N O P Q R S T X
1 :  A B C D E F G I L M N O P Q R S T X
double relation for M(N   T),    MI:    1 M=    2
double relation for M(N   D),    MI:    1 M=    2
double relation for M(I   N),    MI:    3 M=    1
```

APPENDIX 2.b

precedence matrix

	eso	rxm	uft	w<+	cnj	ivl	()*	,.d	ABC	DEF	GIL	MNO	PQR	STX	=#?	h~^
e	000	000	000	000	000	000	010	100	000	000	000	000	000	000	000	000
s	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
o	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
r	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	030
x	000	000	000	001	000	000	030	300	000	000	000	000	000	000	000	000
m	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	000
u	000	000	000	000	000	000	100	000	000	000	000	000	000	000	000	000
f	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	010
t	000	000	030	003	000	000	030	300	000	000	000	000	000	000	000	000
w	000	000	030	003	000	000	031	300	000	000	000	000	000	000	000	000
<	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
+	000	202	221	200	222	222	200	022	222	222	222	222	222	222	000	000
c	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	030
n	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	130
j	000	000	030	003	000	000	033	331	000	000	000	000	000	000	000	330
i	000	000	030	003	000	000	033	310	000	000	000	000	000	000	000	330
v	000	000	030	003	000	000	133	300	000	000	000	000	000	000	000	030
l	000	000	030	003	000	000	333	301	000	000	000	000	000	000	000	030
(	100	020	002	002	000	000	000	000	000	000	000	000	000	000	000	000
)	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	030
*	000	201	220	000	222	222	200	022	222	222	222	222	222	222	000	000
,	100	020	002	002	000	000	000	000	000	000	000	000	000	000	000	000
.	000	000	030	003	002	100	033	302	000	000	000	000	000	000	000	330
d	000	000	030	003	000	000	333	333	000	000	000	000	000	000	000	330
A	000	000	030	003	000	000	333	303	010	000	000	010	000	010	000	030
B	000	000	030	003	000	000	333	303	000	000	000	000	000	100	000	030
C	000	000	030	003	000	000	333	303	000	000	000	001	000	000	000	030
D	000	000	030	003	000	000	333	303	000	000	000	000	000	000	000	030
E	000	000	030	003	000	000	333	303	000	000	000	000	000	001	000	030
F	000	000	030	003	000	000	333	303	000	000	000	010	000	000	000	030
G	000	000	030	003	000	000	333	303	000	000	000	010	000	000	000	030
I	000	303	333	303	333	333	333	333	333	333	333	333	333	333	000	030
L	000	000	030	003	000	000	333	303	000	000	000	001	000	000	000	030
M	000	000	030	003	000	000	333	303	000	000	000	000	000	000	000	030
N	000	000	030	003	000	001	333	303	222	122	222	222	222	212	000	030
O	000	000	030	003	000	000	333	303	000	000	100	000	000	100	000	030
P	000	000	030	003	000	000	333	303	000	000	000	000	000	000	000	030
Q	000	000	030	003	000	000	333	303	000	000	000	000	000	000	000	030
R	000	000	030	003	000	000	333	303	000	000	000	010	000	000	000	030
S	000	000	030	003	000	000	333	303	000	000	110	000	000	000	000	030
T	000	000	030	003	000	000	333	303	100	000	000	010	000	000	000	030
X	000	000	030	003	000	000	333	303	000	000	000	000	100	000	000	030
=	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
#	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
?	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
h	000	000	030	003	000	000	033	300	000	000	000	000	000	000	000	030
~	000	200	210	000	222	222	200	022	222	222	222	222	222	222	000	000
^	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000

```
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 GR,STRING,PARSE,RZ,NRP,WORD,STACK,V3,WD,WR,
     1WDIM,WDAT,AD,OD,NREX,W,WLET,NSY,WCODE,A,B,FIRST
      COMMON  STRING(30),LONG
     1/B0/MP,MPOS(100),GR(500),NPL1(40),NPL2(40),NSY(20,6)
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B3/NCD,NRP(12,12),WDIM(100)
     1/B4/WDAT(100),NLL,NC,AD(9,6),OD(9,6)
     1/B5/M,WD(10),NL,WR(10),LLL,A,B
     1/B6/W(100,2)
     1/B9/NEX,NREX(48,48),WLET(100)
     1/B10/V3(50)
     1/B11/NRV,FIR(40),FIRST(200)
     1/B12/NN(20),NNN(20)
      DIMENSION WCODE(100)
      CALL ASSIGN(1,'DAT1.MIL ',0)
      CALL ASSIGN(2,'DAT2.MIL ',0)
      CALL ASSIGN(3,'DAT3.MIL ',0)
      CALL ASSIGN(4,'DAT4.MIL ',0)
      WRITE(7,777)
777   FORMAT(5X,'OUTPUT DEVICE: '/)
      CALL ASSIGN(6,'X ',-1)
C
      WRITE(6,999)
999   FORMAT(1HO,5X,'DATE :  ,  TIME :'/)
 10   FORMAT(18I4)
 12   FORMAT(5X,'RZ:',15I4)
 18   FORMAT(5X,20I2)
 20   FORMAT(72A1)
 22   FORMAT(18(2A1,2X))
 25   FORMAT(36I2)
 60   FORMAT(10X,40A1)
 70   FORMAT(10X/)
322   FORMAT(16(1X,3I1))
      READ(7,20)STRING
      WRITE(6,20)STRING
      WRITE(6,70)
      WRITE(6,70)
      READ(1,10)N1,N2,N3,NCD,NLL,NC,NEX
      DO 19 I=1,NCD
19    READ(1,25)(NRP(I,J),J=1,NCD)
      READ(1,25)WDIM
      READ(1,25)WDAT
      READ(1,10)WLET
      READ(1,25)((AD(K,L),L=1,NC),K=1,NLL)
      READ(1,25)((OD(K,L),L=1,NC),K=1,NLL)
      READ(1,20)V3
      DO 321 I=1,NEX
321   READ(4,322)(NREX(I,J),J=1,NEX)
      MPOS(1)=1
      READ(2,10)MP,N
      DO 9 J=1,MP
      READ(2,10)LOG,(GR(I),I=MPOS(J),MPOS(J)+LOG-1)
9     MPOS(J+1)=MPOS(J)+LOG
      READ(2,10)NPL1
      READ(2,10)NPL2
      READ(2,22)((W(I,J),J=1,2),I=1,100)
      READ(2,10)((NSY(J,K),J=1,18),K=1,6)
      READ(3,10)NRV
Continue on the next page
```

```
      FIR(1)=1
      DO 446 J=1,NRV
      READ(3,10)LUNG,(FIRST(I),I=FIR(J),FIR(J)+LUNG-1)
446   FIR(J+1)=FIR(J)+LUNG
C     CALL SCRIED

      DO 444 I=1,100
444   WCODE(I)=I
      DO 449 I=1,20
      NNN(I)=0
449   NN(I)=0

      DO 100 NCOUNT=1,25
      WRITE(6,70)
      NPSC=0
      NPS=1
      NP=1
      WRITE(6,10)NCOUNT
      READ(3,20)(STRING(K),K=1,30)
      WRITE(6,60)(STRING(I),I=1,30)
      LUNG=0
      DO 901 K=1,30
      IF(STRING(K).EQ.32)GOTO 901
      LUNG=LUNG+1
      STRING(LUNG)=STRING(K)
      IF(STRING(K).EQ.35)GOTO 902
901   CONTINUE
902   WRITE(6,10)LUNG,(STRING(K),K=1,LUNG)
      DO 903 K=1,10
903   WORD(K)=STRING(K)

C     LUNG is the length of the condensed string
C     i.e. without space character

      NL=7
      CALL MATCH2(WCODE,5,7)
      NR1=RZ(1)
      IF(NR1.EQ.0)GOTO 23
      CALL DIAGNO(24)
      GOTO 102
23    NER=1
      NL=9
      CALL MATCH2(WCODE,5,3)
      NR1=RZ(1)
      CALL DIAGNO(25)
102   WRITE(6,60)(W(GR(I),1),W(GR(I),2),I=MPOS(NR1)+1,MPOS(NR1+1
      L3=RZ(3)
      PARSE(1)=NR1
      CALL DIRIJ(NR1,L3)
      WRITE(6,70)
100   CONTINUE
      CALL EXIT
      END
```

Continue on the next page

```
      SUBROUTINE TRANSD(WCOD,NL,NC,AD,OD)
      LOGICAL*1 GR,STRING,PARSE,RZ,STATE,
     1WORD,STACK,WCOD,AD,OD,V3
      COMMON  STRING(30),LUNG
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
      DIMENSION WCOD(100),AD(NL,NC),OD(NL,NC)
      STATE=1
      WRITE(6,325)
325   FORMAT(3X/)
      L=0
      NO=RZ(3)+1
      DO 300 I=NO,LUNG
      INP=WCOD(STRING(I))
      NG=OD(STATE,INP)
      STATE=AD(STATE,INP)
      IF(STATE.EQ.0)GOTO 310
C
      GOTO(311,312,313,314,315,316,300,318)NG
311     L=L+1
312     WORD(L)=STRING(I)
        GOTO 300
313     L=2
        WORD(1)=STRING(I)
        WORD(2)=STRING(I-1)
        GOTO 300
314     L=L+1
        WORD(L)=48
317     FORMAT(5X,'nr:',2X,12A1)
315     WRITE(6,317)(WORD(J),J=1,L)
316     L=0
        GOTO 300
318     WRITE(6,317)(WORD(J),J=1,L)
        L=1
        WORD(L)=STRING(I)
C
300     CONTINUE
310     WRITE(6,320)STATE,I,STRING(I),INP,L
320     FORMAT(5X,15I4)
        RETURN
        END

C
      SUBROUTINE DIAGNO(NUM)
      LOGICAL*1 GR,STRING,PARSE,RZ,NRP,GREX,WORD
     1,STACK,WDIM,WDAT,AD,OD,NREX,W,WLET,NSY
      COMMON  STRING(30),LUNG
     1/B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B3/NCD,NRP(12,12),WDIM(100)
     1/B4/WDAT(100),NL,NC,AD(9,6),OD(9,6)
     1/B6/W(100,2)
     1/B9/NEX,NREX(48,48),WLET(100)
     GOTO(1,2,3,4,5,6,7,8,9,10,11,12,12,14,15,16,17,18,18,18,21,2
     1,24,25,26,27)NUM
1     WRITE(6,101)
101   FORMAT(5X,'print:')
      RETURN
2     WRITE(6,102)
102   FORMAT(5X,'let')
      RETURN
3     WRITE(6,103)
```

Continue on the next page

```
103     FORMAT(5X,'if:')
        RETURN
4       WRITE(6,104)
104     FORMAT(10X,'start precedence parser'/)
        RETURN
5       WRITE(6,105)
105     FORMAT(5X,'connect:')
        RETURN
6       WRITE(6,106)
106     FORMAT(5X,'dete:')
        RETURN
7       WRITE(6,107)
107     FORMAT(5X,'stack')
        RETURN
8       WRITE(6,107)
108     FORMAT(5X,'data::')
        RETURN
9       WRITE(6,109)
109     FORMAT(10X,'dim'/)
        RETURN
10      WRITE(6,107)
110     FORMAT(5X,'transducer')
        RETURN
11      WRITE(6,111)
111     FORMAT(5X,'end')
        RETURN
12      WRITE(6,112)
112     FORMAT(5X,'go-st'/)
        RETURN
14      WRITE(6,114)
114     FORMAT(5X,'old deriv')
        RETURN
15      WRITE(6,107)
115     FORMAT(5X,'new deriv')
        RETURN
16      WRITE(6,116)
116     FORMAT(5X,'recover')
        RETURN
17      WRITE(6,117)
117     FORMAT(5X,'stack at the end of scanning phase')
18      RETURN
21      WRITE(6,121)STRING
121     FORMAT(10X,'string:',30A1)
        RETURN
22      WRITE(6,122)
122     FORMAT(10X,'rest:',30A1)
        RETURN
23      WRITE(6,123)
123     FORMAT(10X,'success:',30A1)
        RETURN
24      WRITE(6,124)
124     FORMAT(10X,'the sentence production is:',30A1)
        RETURN
25      WRITE(6,125)
125     FORMAT(5X,'misspeling! probable the sentence production is..')
        RETURN
26      IER=RZ(3)
        WRITE(6,126)IER,(STRING(J),J=1,IER)
126     FORMAT(5X,'detection point:',I3,(3X,30A1))
        RETURN
27      WRITE(6,127)
127     FORMAT(10X,'the resolution set is:',30A1)
        RETURN
        END
```

```
C
      SUBROUTINE SCRIED
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 GR,STRING,PARSE,RZ,NRP,WORD,STACK,V3,WG,WK,
     1WDIM,WDAT,AD,OD,NREX,W,WLET,NSY,WCODE,A,B,FIRST
      COMMON STRING(30),LUNG
     1/B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B3/NCD,NRP(12,12),WDIM(100)
     1/B4/WDAT(100),NLL,NC,AD(9,6),OD(9,6)
     1/B5/M,WG(10),NL,WK(10),LLL,A,B
     1/B6/W(100,2)
     1/B9/NEX,NREX(48,48),WLET(100)
     1/B10/V3(50)
     1/B11/NRV,FIR(40),FIRST(200)
      DIMENSION WCODE(100),AAM(10)
   10 FORMAT(18I4)
   11 FORMAT(5X,2I4,5X,10(2A1,2X))
   13 FORMAT(1H ,10X,'the vocabularyof the grammar')
   15 FORMAT(10X,'the precedence matrix for DIM ')
   18 FORMAT(10X,15I3)
   24 FORMAT(//10X,'the productions of the grammar')
   72 FORMAT(5X,I4,10(4X,2A1))
  112 FORMAT(3X//)
  117 FORMAT(1X,A1,2X,10(5I1,1X))
      WRITE(6,10)MP,N2,NCD,NLL,NC,NEX
      WRITE(6,15)
      DO 17 I=1,NCD
   17 WRITE(6,18)(NRP(I,J),J=1,NCD)
      WRITE(6,13)
      DO 16 I=1,20
   16 WRITE(6,72)I,((W(J,K),K=1,2),J=I,100,20)
      WRITE(6,24)
      DO 12 I=1,MP
      LIM1=MPOS(I)
      LIM2=MPOS(I+1)-1
   12 WRITE(6,11)I,LIM1,(W(GR(J),1),W(GR(J),2),J=LIM1,LIM2)
      WRITE(6,112)
      WRITE(6,112)
      WRITE(6,18)WDIM
      WRITE(6,112)
      WRITE(6,10)(NPL1(K),K=1,33)
      WRITE(6,10)(NPL2(K),K=1,33)
      WRITE(6,112)
      WRITE(6,10)((NSY(J,K),J=1,18),K=1,6)
      WRITE(6,112)
      DO 111 K=1,NLL
  111 WRITE(6,118)(AD(K,L),OD(K,L),L=1,NC)
  118 FORMAT(5X,8(2I3,2X))
      WRITE(6,112)
      WRITE(6,18)WDAT
      WRITE(6,112)
      WRITE(6,18)WLET
      WRITE(6,112)
      DO 446 J=1,NRV
  446 WRITE(6,18)FIR(J),(FIRST(K),K=FIR(J),FIR(J+1)-1)
      WRITE(6,112)
      WRITE(6,121)V3
  121 FORMAT(4X,10(5A1,1X))
      DO 119 I=1,NEX
  119 WRITE(6,117)V3(I),(NREX(I,J),J=1,NEX)
      RETURN
      END
```

```
      SUBROUTINE DIRIJ(NR1,L3)
      LOGICAL*1 RZ,NRP,WDIM,NREX,WLET,STRING,WDAT,AD,DD,WORD
     1,WG,WR,META,A,B
      COMMON STRING(30),LUNG
     1/B2/WORD(10),RZ(20)
     1/B3/NCD,NRP(12,12),WDIM(100)
     1/B4/WDAT(100),NLL,NC,AD(9,6),DD(9,6)
     1/B5/M,WG(10),NL,WR(10),LLL,A,B
     1/B9/NEX,NREX(48,48),WLET(100)

      WRITE(6,12)NR1,L3
12    FORMAT(2X,'DIRIJ:',1014)
      GOTO(121,122,123,124,125,126,127,128,129,130,131,132,133
     1,133,133,133)NR1
121   NH=L3
      IF(STRING(NH).NE.39)GOTO 141
142   NH=NH+1
      IF(NH.GE.LUNG)GOTO 143
      IF(STRING(NH).NE.39)GOTO 142
      IF(STRING(NH).EQ.44)NH=NH+1
141   CALL PARSPR(1,NH,NEX,NREX,WLET)
143   RETURN
122   CALL DIAGNO(2)
      CALL PARSPR(2,L3,NEX,NREX,WLET)
      RETURN
123   CALL DIAGNO(3)
      CALL PARSPR(3,L3,NEX,NREX,WLET)
      RETURN
124   CALL DIAGNO(1)
      RETURN
125   CALL DIAGNO(1)
      RETURN
126   CALL DIAGNO(6)
145   CALL PARSPR(6,L3,NEX,NREX,WLET)
      IF(L3.LT.LUNG)GOTO 145
      RETURN
127   CALL DIAGNO(1)
      RETURN
128   CALL TRANSD(WDAT,NLL,NC,AD,DD)
      RETURN
129   CALL DIAGNO(9)
      CALL PARSPR(4,L3,NCD,NRP,WDIM)
      RETURN
130   CALL DIAGNO(10)
      RETURN
131   CALL DIAGNO(11)
      RETURN
132   CALL DIAGNO(12)
      RETURN
133   CALL DIAGNO(13)
      RETURN
134   CALL DIAGNO(14)
      RETURN
135   CALL DIAGNO(15)
      RETURN
136   CALL DIAGNO(16)
      RETURN
137   CALL DIAGNO(17)
      RETURN
138   CALL DIAGNO(18)
      RETURN
      END
```

```
      SUBROUTINE DISTAN(M,CUVG,NL,CUVS,NDD,D,NEW)
      LOGICAL*1 CUVG(10),CUVS(10),D(10,10)
      DO 51 I=1,10
      D(I,1)=I-1
      D(1,I)=I-1
51    CONTINUE

      DO 53 I=2,M+1
      DO 53 J=2,NL+1
      M1=D(I-1,J-1)
      M2=D(I-1,J)+1
      M3=D(I,J-1)+1
      IF(CUVG(I-1).EQ.CUVS(J-1))GOTO 55
      M1=M1+1
55    D(I,J)=M1
      IF(D(I,J).LE.M2)GOTO 57
      D(I,J)=M2
57    IF(D(I,J).LE.M3)GOTO 53
      D(I,J)=M3
53    CONTINUE
      IF(NCAZ.EQ.1)GOTO 63
      NDD=D(M+1,1)
      NN1=2
      DO 59 J=2,NL+1
      IF(D(M+1,J).GT.NDD)GOTO 59
      NDD=D(M+1,J)
      NN1=J
59    CONTINUE
      NEW=NN1
      RETURN

63    NDD=D(M+1,NL+1)
      RETURN
      END
C
C
      SUBROUTINE TRACE(M,NL,D1,TR,NT)
      LOGICAL*1 D1(10,10),TR(20)
      NT=0
      DO 67 I=1,M+1
67    WRITE(6,69)(D1(I,J),J=1,NL+1)
69    FORMAT(5X,10I3)
      I=M+1
      J=NL+1
65    IF(I.EQ.1)GOTO 71
      IF(D1(I,J).NE.D1(I-1,J)+1)GOTO 71
      I=I-1
      WRITE(6,63)I,J
      GOTO 65
71    IF(J.EQ.1)GOTO 61
      IF(D1(I,J).NE.D1(I,J-1)+1)GOTO 73
      J=J-1
      WRITE(6,63)I,J
      GOTO 65
73    I=I-1
      J=J-1
      TR(NT+1)=I
      TR(NT+2)=J
      NT=NT+2
      WRITE(6,63)I,J
      GOTO 65
63    FORMAT(1X,' I=',I4,4X,'J=',I4)
61    RETURN
      END
```

```
      SUBROUTINE MATCH2(WCOD,META,NCAZ)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 GR,NSY,RZ,D,D1,TR,WG,RES,WCOD,WORD,WR,META,A,B,W
      COMMON /B0/MF,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
     1/B2/WORD(10),RZ(20)
     1/B5/M,WG(10),NL,WR(10),LLL,A,B
     1/B6/W(100,2)
     1/B8/MK,RES(10,8),MRESP(10,8),MLG(10),BETA(10,8),BETB(10,8)
      DIMENSION D(10,10),D1(10,10),TR(20),WCOD(100)

C     tries to match substrings against grammar
      MK1=1
      NS=META
333   FORMAT(5X,'match.:',12I3)
      NS=NL
      NLUNG=0
      DO 109 K=1,NL
109   WR(K)=WCOD(WORD(K))
      WRITE(7,333)(WR(K),K=1,NL)
      NDIST=10
      LIM=NSY(1,META)
      DO 100 NK=2,LIM
      NK1=NSY(NK,META)
      DO 100 I=NPL1(NK1),NPL2(NK1)

      CALL RHSPR(I,NL,M,WG)
      DO 116 L=1,M
116   WG(L)=WCOD(WG(L))

      GOTO(118,120,122,123,124,121,119)NCAZ
Cc    Diferent subroutines  NCAZ:=1,2,3,4,5

Cc    NCAZ=6 is for suffix of a production
121   CALL SUFFIX
      IF(LLL.NE.NL)GOTO 100
      BETA(MK,MK1)=MPOS(I+1)-LLL
      BETB(MK,MK1)=MPOS(I+1)-1
      GOTO 104

Cc    NCAZ=3 is for not fixed length: N,NEW (122)
122   NL=M+2
Cc    NCAZ=1 is for fixed length N (118)
118   CALL DISTAN(M,WG,NL,WR,NDD,D,NEW)
      IF(NDD.GT.NDIST)GOTO 100
      IF(NDD.EQ.NDIST.AND.NS.GE.NL)GOTO 100
      NDIST=NDD
      NLUNG=M
      NS=NEW
      NI=I
      DO 128 J=1,M+1
      DO 126 L=1,NL+1
126   D1(J,L)=D(J,L)
128   CONTINUE
      IF(NCAZ.EQ.3)NL=NEW
      GOTO 100
Cc    (119) is for not fixed length (NL)
Cc    match without errors (NCAZ=2) (120)
119   NL=M
120   IF(M.NE.NL)GOTO 100
      CALL PREFIX
      IF(LLL.LT.0)GOTO 100
      NI=I
      RZ(3)=M
      GOTO 127
```

Continue on the next page

```
Cc      NCAZ=5; is for substring of a production
124     CALL SUBSTR
        IF(LLL.LT.0)GOTO 100
        BETA(MK,MK1)=MPOS(I)+1+M
        BETB(MK,MK1)=MPOS(I)+LLL-1
        GOTO 104

Cc      NCAZ=4 is for prefix of a production
123     CALL PREFIX
        IF(LLL.NE.NL)GOTO 100
        BETA(MK,MK1)=MPOS(I)+1
        BETB(MK,MK1)=MPOS(I)+LLL
104     NI=I
        RES(MK,MK1)=GR(MPOS(I))
        MRESP(MK,MK1)=I
        MLG(MK)=MK1
        RZ(3)=M
C       WRITE(6,93)GR(MPOS(I)),1,MK,MLG(MK),BETA(MK,MK1),BETB(MK,MK1)
93      FORMAT(2X,10I4)
        MK1=MK1+1

100     CONTINUE

C       end of grammar
        IF(NCAZ.EQ.2)GOTO 129
        IF(NCAZ.EQ.7)GOTO 129
        IF(NCAZ.GE.4)GOTO 127
        CALL TRACE(NLUNG,NS,D1,TR,NT)
25      FORMAT(2X,'trace:',18I3)
        WRITE(6,25)(TR(K),K=1,NT)
        RZ(3)=TR(2)
132     FORMAT(25X,2A1,' -- ',29A1)

127     K1=MPOS(NI)
        WRITE(6,132)(W(GR(J),1),W(GR(J),2),J=K1,K1+M)
        RZ(1)=GR(MPOS(NI))
        RZ(2)=NLUNG
        RETURN
129     RZ(1)=0
        RZ(2)=0
        RZ(3)=M
        RETURN
        END
C


        SUBROUTINE SUFFIX
        LOGICAL*1 WG,WR,WORD,RZ,META,A,B
        COMMON /B2/WORD(10),RZ(20)
        1/B5/M,WG(10),NL,WR(10),LLL,A,B
        DO 137 J=1,M
137     RZ(M+1-J)=WG(J)
        DO 139 J=1,NL
139     RZ(11+NL-J)=WR(J)
        DO 135 J=1,10
        WR(J)=RZ(J+10)
135     WG(J)=RZ(J)
        CALL PREFIX
        RETURN
        END
```

Continue on the next page

```
      INTEGER FUNCTION METASY(A)
      LOGICAL*1 A
      NU=0
      IF(A.LT.34)NU=1
      METASY=NU
      RETURN
      END

      SUBROUTINE PREFIX(META)
      LOGICAL*1 WG,WR,META,A,B
      COMMON /RS/M,WG(10),NL,WR(10),LLL,A,B
      MIN=M
      IF(M.GT.NL)MIN=NL
      DO 10 I=1,MIN
      IF(WG(I).NE.WR(I))GOTO 12
   10 CONTINUE
      LLL=MIN
      RETURN
   12 LLL=-1
      RETURN
      END

      SUBROUTINE SUBSTR(META)
      LOGICAL*1 WG,WR,META,A,B
      COMMON /RS/M,WG(10),NL,WR(10),LLL,A,B
      I=1
   12 IF(WR(1).EQ.WG(I))GOTO 10
      I=I+1
      IF(I.LE.M)GOTO 12
   16 LLL=-1
      RETURN
   10 M=I
      DO 14 J=2,NL
      I=I+1
      IF(WG(I).NE.WR(J))GOTO 16
   14 CONTINUE
      LLL=I
      RETURN
      END

      SUBROUTINE RHSPR(I,NL,M,WG)
      LOGICAL*1 WG(10),GR,NSY,WORD,RZ
      COMMON /BO/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,4
     1/RP/WORD(10),RZ(20)
C     finds the RHS of the I-th production
      M=0
      DO 102 K=MPOS(I)+1,MPOS(I+1)-1
      IF(GR(K).EQ.91)GOTO 103
      M=M+1
      WG(M)=GR(K)
  102 CONTINUE
      RETURN
  103 M1=0
      K=K+1
  112 M1=M1+1
      RZ(M1)=GR(K)
      K=K+1
  108 IF(GR(K).NE.93)GOTO 112
      INT=(NL-M)/M1
      IF(INT.EQ.0)GOTO 114
      DO 116 J=1,INT
      DO 116 L=1,M1
      M=M+1
  116 WG(M)=RZ(L)
  114 RETURN
      END
```

```
CCCCC      SIMPLE PRECEDENCE PARSER     CCC
C
      SUBROUTINE PARSPR(META,NPUINT,NDI,MATR,WCOD)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 STRING,GR,NSY,PARSE,WORD,STACK,MATR
     1,RZ,WCOD,W,RES,BETA,WG,WR,META,A,B
      COMMON STRING(30),LONG
     1/B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B5/M,WG(10),NL,WR(10),LLL,A,B
     1/B6/W(100,2)
     1/B7/DR,DER(10),DSL(10),ANC(80),DPR(80),SUC(80),ETA(10,8)
     1,ETB(10,8),DRR(10),DNL(10),NDR
     1/B8/MK,RES(10,8),MRESP(10,8),MLG(10),BETA(10,8),BETB(10,8)
      DIMENSION MATR(NDI,NDI),WCOD(100)


C
Cc      parsing a simple precedence grammar
Cc      I  is a pointer in the string
C       NPS  is a pointer in the stack
Cc      I1  is the topmost stack symbol
Cc      I2  is the incomming input symbol
Cc      MATR  is the precedence matrix of dimension NDI
Cc

      STACK(1)=35
      NPS=1
      I1=35
      LIM=RZ(3)+1
      I=LIM
  145 I2=STRING(I)
      IF(MATR(WCOD(I1),WCOD(I2)).NE.0)GOTO 130
      I=I+1
      GOTO 145
  144 I1=I2
  244 I=I+1
      I1=I2
      I2=STRING(I)
      IF(WCOD(I2).EQ.NDI)GOTO 244
  114 FORMAT(2X,'I1=',I4,' I2=',I4,' REL=',I2,2I4,' NPS=',I4,' I=',
  130 LAB=MATR(WCOD(I1),WCOD(I2))+1
      WRITE(7,114)I1,I2,WCOD(I1),WCOD(I2),LAB,NPS,I
      GOTO(106,112,113,110)LAB

Cc    a simple phrase has been detected
Cc      'C'  marks its beginning
Cc      if  I1<. I2  then  'C'  and I2 are put in the stack (113)
Cc      if  I1  is not related with I2  then '?'  and I2 -.-  (106)
Cc      if  I1 .> I2 a phrase has been detected in the stack(110)
Cc      and the subroutine  REDUCE  is called
  110 WRITE(6,115)(W(STACK(K),1),W(STACK(K),2),K=1,NPS)
      IF(STACK(NPS).EQ.META)GOTO 146
      PARSE(100)=I
      CALL REDUCE(NDI,MATR,WCOD,META,I2)
  104 NPOINT=RZ(3)
      IF(RZ(1).EQ.0)GOTO 112
      I1=STACK(NPS)
      GOTO 130
```

```
Cc      dn error has been discovered
118     FORMAT(12X,I3,2X,A1)
106     NPS=NPS+1
        STACK(NPS)=63
        RZ(3)=1
        CALL DIAGNO(26)
        GOTO 112

113     NPS=NPS+1
        STACK(NPS)=91
112     NPS=NPS+1
        STACK(NPS)=I2
115     FORMAT(2X,'Stack:',(17(1X,2A1)))
        WRITE(6,115)(W(STACK(K),1),W(STACK(K),2),K=1,NPS)
        IF(I.LT.LUNG)GOTO 144
        IF(NPS.LE.3)GOTO 111
        I2=33
        CALL RECOVE(NDI,MATR,WCOD,META,I2)
111     RETURN
146     NPOINT=I
        RETURN
        END


        SUBROUTINE REDUCE(ND1,MATR,WCOD,META,I2)
        IMPLICIT INTEGER (A-Z)
        LOGICAL*1 STRING,GR,NSY,PARSE,WORD,STACK,MATR
        1,RZ,WCOD,META,RES,WG,WR,A,B
        COMMON STRING(30),LUNG
        1/B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
        1/B1/NP,PARSE(100),NPS,STACK(60)
        1/B2/WORD(10),RZ(20)
        1/B5/M,WG(10),NL,WR(10),LLL,A,B
        1/B8/MK,RES(10,8),MRESP(10,8),MLG(10),METR(10,8),METB(10,8)
        DIMENSION MATR(ND1,ND1),WCOD(100)
        NER=0
        RZ(1)=0
        DO 154 J=NPS,1,-1
        IF(STACK(J).EQ.65)GOTO 170
        IF(STACK(J).EQ.91)GOTO 156
154     CONTINUE
156     NO=J+1
        IF(NER.NE.0)GOTO 170
        K=0
        DO 158 L=NO,NPS
        K=K+1
158     WORD(K)=STACK(L)
        NL=K
        CALL MATCH2(WCOD,META,2)
        NR1=RZ(1)
        IF(NR1.EQ.0)GOTO 162
        IF(MATR(WCOD(STACK(J-1)),WCOD(NR1)).EQ.2)GOTO 152
        J=J-1
152     NPS=J+1
        STACK(NPS)=NR1
        NP=NP+1
        PARSE(NP)=RZ(4)
        RETURN
```

```
Cc      a phrase error has been discovered
162     CALL PHRASE(NDI,MATR,WCOD,META)
        RETURN

Cc      errors on the stack
170     NPS=NPS+1
        STACK(NPS)=93
C       CALL RECOVE(NDI,MATR,WCOD,META)
        RETURN
        END
```

```
      SUBROUTINE PHRASE(NDI,MATR,WCOD,META)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 MATR(NDI,NDI),WCOD(100),STRING,RZ,WORD
     1,META,WG,WR,A,B
      COMMON STRING(30),LUNG
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B5/M,WG(10),NL,WR(10),LLL,A,B
      CALL MATCH2(WCOD,META,4)
      NPS=NO
      STACK(NPS)=RZ(1)
      RETURN
      END
```

```
      SUBROUTINE  RECOVE(NDI,MATR,WCOD,META)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 STRING,GR,NSY,PARSE,WORD,STACK,MATR
     1,RZ,WCOD,META,W,RES,ETA,ETB,WG,WR,A,B
      COMMON STRING(30),LUNG
     1/B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,..
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B5/M,WG(10),NL,WR(10),LLL,A,B
     1/B6/W(100,2)
     1/B7/DR,DER(10),DLS(10),ANC(80),DPR(80),SUC(80),ETA(10,8)
     1,ETB(10,8),DRR(10),DNL(10),NDR,NCC
     1/B8/MK,RES(10,8),MRESP(10,8),MLG(10),BETA(10,8),BETB(10,8)
     1/B12/NN(20),NNN(20),DEL,DEL1
      DIMENSION MATR(NDI,NDI),WCOD(100)


      DO 79 J=1,10
      ETA(J,8)=35
79    MLG(J)=0
      MK=0
      CALL DIAGNO(17)
      WRITE(6,125)NPS,(W(STACK(J),1),J=1,NPS)
125   FORMAT(5X,32A2)
      DEL=STACK(2)
      I=2
      NCC=0

122   I=I+1
      NL=0
      IF(MK.NE.1)GOTO 154
```

```
Cc      initially DERIV=RES1
        DR=MLG(1)
        CALL DIAGNO(14)
        DO 10 J=1,DR
        NCC=NCC+1
        DER(J)=J
        ANC(J)=0
        DPR(J)=MRESP(1,J)
        DLS(J)=J
        LIM=BETB(1,J)+1
        LM=MPOS(DPR(J)+1)-1
        L=1
        WRITE(7,41)LIM,LM,DPR(J),J
41      FORMAT(10I4)
        IF(LIM.GT.LM)GOTO 14
        DO 12 K=LIM,LM
        ETA(J,L)=GR(K)
12      L=L+1
14      ETA(J,L)=35
        SUC(J)=0
        WRITE(6,34)J,NCC,DER(J),DLS(J),(W(ETA(J,L1),1),L1=1,L)
10      CONTINUE
        WRITE(6,37)
        DO 30 K=1,NCC
30      WRITE(6,36)ANC(K),SUC(K),DPR(K)
        1,(W(GR(L1),1),L1=MPOS(DPR(K)),MPOS(DPR(K)+1)-1)

154     NL=NL+1
        WORD(NL)=STACK(I)
        IF(I.GE.NPS)GOTO 156
        I=I+1
        IF(STACK(I).NE.63.AND.STACK(I).NE.91.AND.STACK(I).NE.93)GOTO
        DEL1=STACK(I)
124     MK=MK+1
        WRITE(6,123)MK,(W(WORD(K),1),K=1,NL)
123     FORMAT(5X,'    beta ',I2,' is ', 10A2)
        IF(DEL.EQ.91.OR.MK.EQ.1)GOTO 77
        IF(DEL1.EQ.93)GOTO 152

C       substring
        IF(NL.EQ.1)GOTO 78
        CALL DIAGNO(27)
        CALL MATCH2(WCOD,META,5)
        GOTO 153

78      MK=MK+1
        BETA(MK,J)=10
        BETB(MK,J)=499
        RES(MK,1)=WORD(1)
        MRESP(MK,1)=0
        MLG(MK)=1
        GOTO 153

C       prefix
77      CALL DIAGNO(27)
        CALL MATCH2(WCOD,META,4)
        IF(MK.EQ.1)GOTO 123
        GOTO 153

C       suffix
152     CALL DIAGNO(27)
        CALL MATCH2(WCOD,META,6)
```

```
153     CALL CONECT(META,NDI,MATR)
        DEL=DEL1
        WRIIE(6,32)DR,NDR
32      FORMAT(40X,8I4)
34      FORMAT(30X,'j,ncc,eta:',4I4,5X,12A1)
37      FORMAT(30X,'anc,suc,prod:')
36      FORMAT(28X,3I4,2X,10A1)

        CALL DIAGNO(15)
        DO 42 J=1,NDR
42      WRITE(6,34)J,NCC,NCC,NCC,(W(ETB(J,L1),1),L1=1,2)
        WRITE(6,37)
        DO 43 K=1,NCC
43      WRITE(6,36)ANC(K),SUC(K),DPR(K)
        1,(W(GR(L1),1),L1=MPOS(DPR(K)),MPOS(DPR(K)+1)-1)
        DR=NDR
        DO 46 J=1,DR
        DER(J)=DRR(J)
        DLS(J)=DNL(J)
        DO 48 L=1,8
48      ETA(J,L)=ETB(J,L)
46      CONTINUE
        GOTO 122
156     RETURN
        END


        SUBROUTINE AFLA(META,AA,BB)
        IMPLICIT INTEGER (A-Z)
CC      finds a production  IP = u IA v IB w  such that
Cc      A is in FIRST(IA) and  B is in FIRST(IB)
Cc      there are NN(1) such productions denoted by NN(2),..
        LOGICAL*1 WG,WR,RES,IA,IB,GR,NSY,META,A,B,AA,BB
        COMMON /B0/MF,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
        1/B5/M,WG(10),NL,WR(10),LLL,A,B
        1/B12/NN(20),NNN(20),DEL,DEL1
        A=AA
        B=BB
        LIM=NSY(1,META)
        NN(1)=1
        DO 100 NK=2,LIM
        NK1=NSY(NK,META)
        DO 100 I=NPL1(NK1),NPL2(NK1)
        CALL RHSPR(I,NL,M,WG)
        J=1
15      IF(METASY(WG(J)).EQ.1)GOTO 10
        IF(J.GE.M)GOTO 100
        J=J+1
        GOTO 15
10      IA=WG(J)
11      IF(J.GE.M)GOTO 100
        J=J+1
        IF(METASY(WG(J)).NE.1)GOTO 11
        IB=WG(J)
        WRITE(6,920)A,B,IA,IB,I,NV
920     FORMAT(9X,'a,b,ia,ib,i,nv',8I4)
        CALL FRST(IA,A,NV)
        IF(NV.NE.1)GOTO 30
        CALL FRST(IB,B,NV)
        IF(NV.EQ.0)GOTO 30
        NN(1)=NN(1)+1
        NN(NN(1))=I
30      IA=IB
        GOTO 11
100     CONTINUE
```

```
      IF(NN(1).EQ.1)GOTO 20
      CALL DIAGNO(23)
      WRITE(6,905)(NN(J),J=1,NN(1))
905   FORMAT(5X,10I4)
      RETURN
20    WRITE(6,919)
919   FORMAT('  fail')
      END


      SUBROUTINE WHICH(FA,GA)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 GR,NSY,FA,GA,A,B,WG,WR
      COMMON /B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,4
     1/B5/M,WG(10),NL,WR(10),LLL,A,B
     1/B12/NN(20),NNN(20),DEL,DEL1
C     finds a derivation  FA ==>*  u GA r

      NNN(1)=1
      NK1=FA
      DO 100 I=NPL1(NK1),NPL2(NK1)!Examine all FA-productions
      DO 102 K1=MPOS(I)+1,MPOS(I+1)-1!find a metasymbol in RHS(I)
      IF(METASY(GR(K1)).NE.1)GOTO 102
      CALL FRST(GR(K1),GA,NV)
      IF(GR(K1).EQ.GA)GOTO 104
      IF(NV.EQ.0)GOTO 100
      WRITE(6,921)FA,GA,GR(K1),NV,I
921   FORMAT(12X,'father Ga,gr :',8I5)
      IF(GR(K1).EQ.FA)GOTO 100
104   NNN(1)=NNN(1)+1
      NNN(NNN(1))=I
      GOTO 100
102   CONTINUE
100   CONTINUE
      WRITE(6,912)GA,FA,(NNN(J),J=1,NNN(1))
912   FORMAT(3X,'result:A,FA,nn:',1214/27X,10I4)
      RETURN
      END


      SUBROUTINE FRST(FA,GA,NV)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 GA,FA,FIRST
      COMMON /B11/NRV,FIR(40),FIRST(200)
      NK1=FA
      IF(FIR(NK1+1).LE.FIR(NK1))GOTO 106
      DO 10 I=FIR(NK1),FIR(NK1+1)-1
      IF(FIRST(I).EQ.GA)GOTO 104
10    CONTINUE
106   NV=0
      RETURN
104   NV=1
      RETURN
      END
```

```
      SUBROUTINE CONECT(META,NDI,MATR)
      IMPLICIT INTEGER (A-Z)
      LOGICAL*1 STRING,GR,NSY,PARSE,WORD,STACK,MAIR
     1,RZ,WCOD,META,W,RES,ETA,ETB,WG,WK,A,B
      COMMON STRING(30),LONG
     1/B0/MP,MPOS(150),GR(500),NPL1(40),NPL2(40),NSY(20,6)
     1/B1/NP,PARSE(100),NPS,STACK(60)
     1/B2/WORD(10),RZ(20)
     1/B5/M,WG(10),NL,WK(10),LLL,A,B
     1/B6/W(100,2)
     1/B7/DR,DER(10),DLS(10),ANC(80),DPR(80),SUC(80),ETA(10,8)
     1,ETB(10,8),DRR(10),DNL(10),NDR,NCC
     1/B8/MK,RES(10,8),MRESP(10,8),MLG(10),BETA(10,8),BETB(10,8)
     1/B12/NN(20),NNN(20),DEL,DEL1
      DIMENSION MATR(NDI,NDI),WCOD(100)
      CALL DIAGNO(5)
      NDR=0


      DO 135 K=1,DR! There are DR possible derivations
      MP=DPR(DER(K))
      A=GR(MPOS(MP))
      IF(ETA(K,1).EQ.35)GOTO 139
      C1=0
      DO 10 L=1,8
      IF(ETA(K,L).EQ.35)GOTO 12
      IF(METASY(ETA(K,L)).EQ.1.AND.C1.EQ.0)C1=ETA(K,L)
10    WG(L)=ETA(K,L)
12    M=L-1

C     There are MLG(MK) elements in the set Res(MK)
      DO 135 J=1,MLG(MK)
      MQ=MRESP(MK,J)
      B=GR(MPOS(MQ))

Ca    check eta ==>* u1 beta r1
      IF(BETA(MK,J).GT.BETB(MK,J)) GOTO 14!insertion'


Ca2
137   IF(B.NE.C1.OR.C1.EQ.0)GOTO 159
      NDR=NDR+1
      DRR(NDR)=DER(K)
      NCC=NCC+1
      ANC(NCC)=NCC-1
      DPR(NCC)=MQ
      DNL(NDR)=NCC
      DO 20 I=L,8
20    ETB(NDR,I+1-L)=ETA(K,L)
      IF(DEL.EQ.91)GOTO 135
Ca1
      NL=0
159   WRITE(6,92)MK,J,BETA(MK,J),BETB(MK,J)
92    FORMAT(2X,'MK,J,beta',614)
      DO 16 L=BETA(MK,J),BETB(MK,J)
      NL=NL+1
16    WR(NL)=GR(L)
      CALL SUBSTR
      WRITE(6,91)NL,LLL
91    FORMAT(20X,'n1,111:=',216)
      IF(LLL.LT.0)GOTO 157
14    NDR=NDR+1
      DRR(NDR)=DER(K)
      DO 18 L=M,LLL
18    ETB(NDR,L+1-M)=ETA(J,L)
      GOTO 139
```

```
Cb      check: is there a metasymbol needed on
Ch       the left of beta in RHS of rule G.
157     IF(MPOS(MQ)+1.GT.BETA(MK,J))GOTO 139
        DO 147 L=MPOS(MQ)+1,BETA(MK,J)
        IF(METASY(GR(L)).EQ.0) GOTO 147
        AI=GR(L)
C       is AI = a v ?
        CALL WHICH(AI,A)
        NCO=NCC+1
        NDR=NDR+1
        NCC=NCC+1
        ANC(NCO)=NCC
        DPR(NCC)=MQ
        SUC(NCC)=NCO
        DRR(NDR)=NCC
        DNL(NDR)=NCC-1
        LIM=BETB(MK,J)
        LM=MPOS(MQ+1)-1
        L=1
        IF(LIM.GT.LM)GOTO 84
        DO 82 L=LIM,LM
        ETA(J,L)=GR(K)
82      L=L+1
84      ETA(J,L)=35
147     CONTINUE

Cd      check C ==> r1 A u B r2
139     NL=5
        WRITE(6,901)A,B
901     FORMAT(6X,'A,B:=',2I6)
        CALL AFLA(META,A,B)
        IP=NN(2)
        IF(NN(1).LE.1)GOTO 135
        NDR=NDR+1
        NCO=NCC+1
        NC1=NCC
        NCC=NCC+1
        DRR(NDR)=NCC
        DPR(NCC)=IP
        SUC(NCC)=NCO
        SUC(NC1)=NCC+1
72      CALL WHICH(IA,A)
74      CALL WHICH(IB,B)

C14     insertion hypothesis

135     CONTINUE
        RETURN
        END
```

He, who is thinking alone, and is scratching the light,

Has given a new life and an iron man, the machine,

A being invented by dream and vision

Unbelivable stronger than his arm and shoulder.

Tudor Arghezi

(from "He, who is thinking alone")