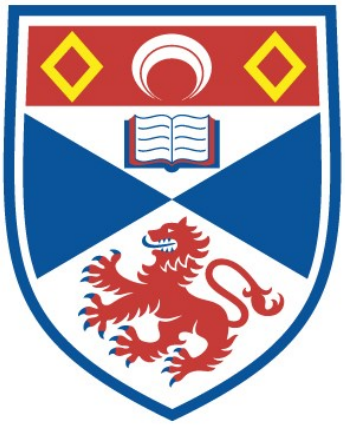
AN EXTENSIBLE SYSTEM FOR THE AUTOMATIC TRANSMISSION OF A CLASS OF PROGRAMMING LANGUAGES

Najam Perwaiz

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1975

An Extensible System
for
The Automatic Translation
of a
Class of Programming Languages

N. Perwaiz

This thesis is dedicated

to

my parents,

and to,

my teachers

for their training enabled me

to produce this work.

ABSTRACT

This thesis deals with the topic of programming linguistics. A survey of the current techniques in the fields of syntax analysis and semantic synthesis is given. An extensible automatic translator has been described which can be used for the automatic translation of a class of programming languages.

The automatic translator consists of two major parts : the syntax analyser and the semantic synthesizer. The syntax analyser is a generalised version of LL(K) parsers, the theoretical study of which has already been published by Lewis and Stearns and also by Rosenkrantz and Stearns. It accepts grammar of a given language in a modified version of the Backus Normal Form (MBNF) and parses the source language statements in a top down, left to right process without ever backing up.

The semantic synthesizer is a table driven system which is called by the parser and performs semantic synthesis as the parsing proceeds. The semantics of a programming language is specified in the form of semantic productions. These are used by the translator to construct semantic tables.

The system is implemented in SNOBOL4 (SPITBOL version 2.0) on an IBM 360/44 and its description is supported by various examples. The automatic translator is an extensible system and SNOBOL4, the implementation language appears as its subset. It can be used to introduce look ahead in the parser, so that backup can be avoided. It can also be used to introduce new facilities in the semantic synthesizer.

I hereby declare that the conditions of the Ordinance and regulations for the degree of Doctor of Philosophy (Ph.D.) at the University of St. Andrews have been fulfilled by the candidate, Najam Perwaiz.

28. XII. 74

I hereby declare that this thesis has been composed by myself; that the work of which it is a record has been done by myself; and, that it has not been accepted in any previous application for any higher degree. This research was undertaken on 1st October, 1971, the date of my admission as a research student.

ACKNOWLEDGEMENTS

I should like to thank Professor A.J. Cole, head of the department of Computational Science, University of St. Andrews for providing me the opportunity to do this work and his guidance throughout this project.

I think I will be foregoing the most pleasant duty of my postgraduate years, if I do not thank my supervisor Michael Weatherill for his help and support throughout this project. Without his encouragement, excellent supervision, advice and healthy criticism this work would not have been possible. I would particularly like to thank him for his many weekends spent on reading early drafts of this thesis.

I also thank Prof. J.W.F. Mulder, head of the department of linguistics, University of St. Andrews for his useful critical comments, and I thank Mr. D.A. Turner and Dr. R. Fisher for their valuable comments. I also acknowledge the help given to me by Dr. R. Erskine, the computing manager by allowing my "hands on the machine".

Finally I thank Mrs. Alice Murray for typing this thesis.

ABSTRACT

This thesis deals with the topic of programming linguistics. A survey of the current techniques in the fields of syntax analysis and semantic synthesis is given. An extensible automatic translator has been described which can be used for the automatic translation of a class of programming languages.

The automatic translator consists of two major parts : the syntax analyser and the semantic synthesizer. The syntax analyser is a generalised version of LL(K) parsers, the theoretical study of which has already been published by Lewis and Stearns and also by Rosenkrantz and Stearns. It accepts grammar of a given language in a modified version of the Backus Normal Form (MBNF) and parses the source language statements in a top down, left to right process without ever backing up.

The semantic synthesizer is a table driven system which is called by the parser and performs semantic synthesis as the parsing proceeds. The semantics of a programming language is specified in the form of semantic productions. These are used by the translator to construct semantic tables.

The system is implemented in SNOBOL4 (SPITBOL version 2.0) on an IBM 360/44 and its description is supported by various examples. The automatic translator is an extensible system and SNOBOL4, the implementation language appears as its subset. It can be used to introduce look ahead in the parser, so that backup can be avoided. It can also be used to introduce new facilities in the semantic synthesizer.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 PROJECT SURVEY:-

This project in its final form evolved from an attempt to develop an English like programming language for school and other non-specialist students. This language was to be called "Processing of Arithmetic and Textual Expressions" (PATE). After the implementation of the text processing facilities of the PATE processor (described in full in chapter 8), it was felt that for most of the sophisticated developments of such a language, an automatic mechanism based on some formal grammar was desirable. This could then be used for implementing and testing the different features of PATE. The investigation of this topic bore interesting results and forms the bulk of the work described in this thesis.

In dealing with programming languages and their translators we are concerned with their inherent structural properties and the kinds of transformations which the

structure may initiate or undergo when it enters into a computation. The inherent structural properties are referred to as syntactic properties, and the transformational properties of the structures are referred to as semantic properties.

For example the set of all representations of programs in a specific programming language is called its syntax. The representation of the effect of executing the programs in a programming language is called the semantics of the programming language.

It is convenient to have a formal method for representing the syntactic and the semantic properties of classes of programming languages.

The notations in which the syntax and the semantics are defined are known as metasyntactic and metasemantic languages respectively. A combination of the two is called a metalanguage. We want to use the metalanguage as a vehicle for constructing programming language translators and hence will refer to it as the metatranslation language (MTL). As we are

only concerned with writing compilers and interpreters for high level programming languages, the word translator refers to these two types of programs only. Other sorts of software e.g. assemblers to which the term has been applied may present different problems.

Initially all translators were written in assembler language. Although all types of time and space optimizations are possible in assembler language programming, since it is rather cumbersome, experience shows that all the code does not get due care. Recently there has been a big trend towards writing translators in high level programming languages. We believe this is a step forward and contend that by selecting a suitable high level language for implementation, similar and perhaps even better results can be obtained with considerably less programming effort.

The general subject of interest in this dissertation is "programming linguistics" which we consider to be a science concerning the design and specification of programming

languages and the translation and subsequent evaluation and execution of programs in these languages. In particular we are primarily interested in the problem of automatic translator translation. We define automatic translator translation loosely as the process of using a computer to perform some stages of the work involved in writing a translator. The program which performs this task is called an automatic translator. It has two parts: the syntax analyzer and the semantic synthesizer.

For the purposes of this research such a system has been implemented in SNOBOL4 [GRISWORLD 70] (SPITBOL version) [DEWAR 71].

Some special purpose high level programming languages have previously been designed for writing systems programs. A class of these languages with special facilities for compiler writing is called compiler compilers. An ideal compiler compiler is one which has formal syntax and semantics as its input and whose output consists of a compiler written in some already implemented language. The

existing compiler compilers however do not achieve so much as this. While we would claim to have achieved more, we must admit that this ideal has not been reached. A compiler compiler normally acts as a high level language in which other compilers are written and at least parts of it reside in the core as an integral part of the compiler. Some compiler compilers have embedded in them some automatic syntax analysis mechanism, hence automating this part of the task.

Our automatic translator is a high level problem oriented language, the problem being to write translators for programming languages. One belief fundamental to our work is that the context free grammars (defined in chapter 2) can continue to be used in a natural and convenient way as a basis for the specification of significant portions of the syntax and translation of programming languages. Furthermore we find that a well designed contex free grammar makes a concise, readable and useful syntactic reference for a language from which operator precedences and associativities

and other properties can be quickly and easily determined.

The automatic translator being described is a table driven system. The syntax and the semantics of a given programming language are read and internal tables constructed from the information thus acquired. A mechanism has been provided for semantic extensibility (explained in chapter 7). Different extension programs can be written in SNOBOL4 and they are compiled at the execution time of the automatic translator. These programs provide extra information to the translator.

During the development stages of a translation system for a language, the grammar would be read and the tables constructed before the source language statements are read for each run. However a fully debugged system will have the tables embedded. Each source statement is read and processed individually. The syntax analyzer recognizes it and as soon as a sufficient amount of information is available the semantic synthesizer is called for action.

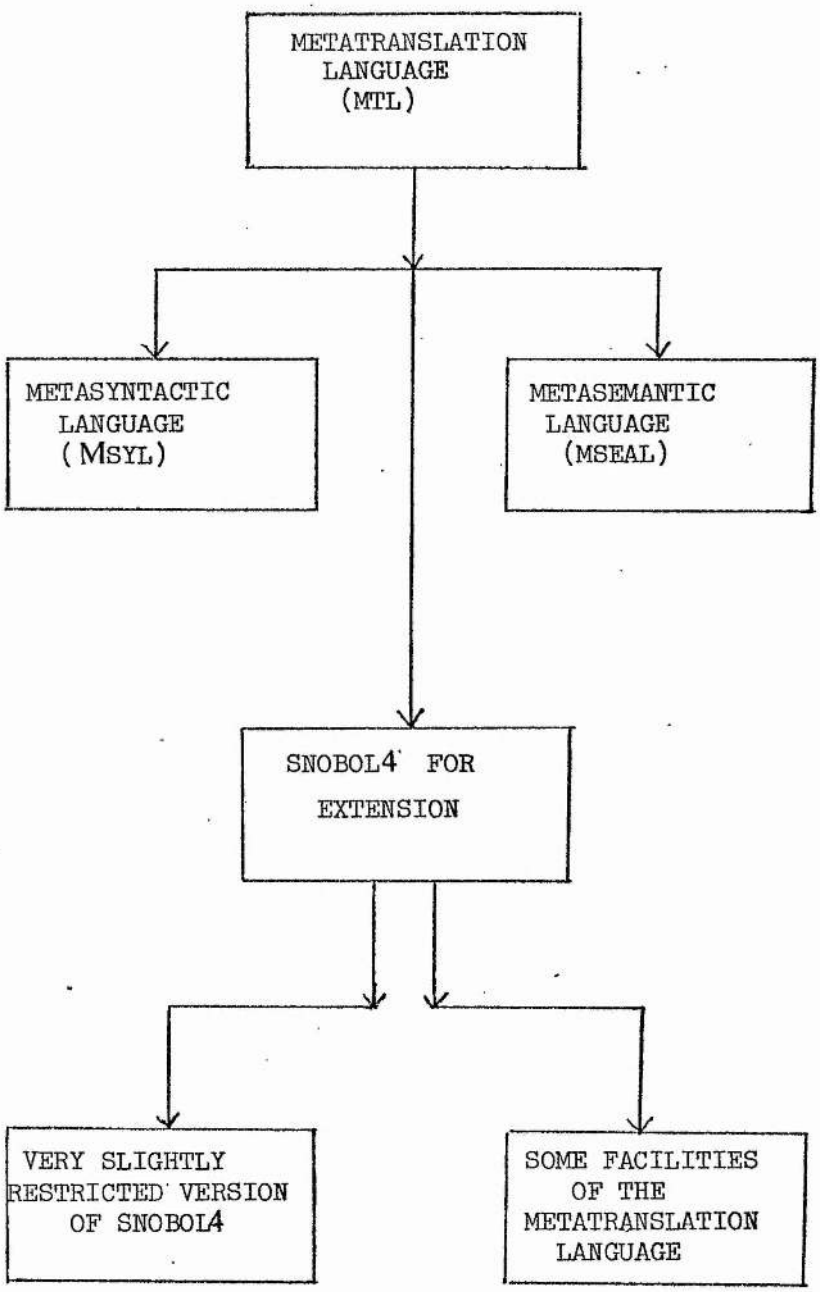


FIG. 1.1 . The relation between the parts of the Meta-translation language.

1.2 SYNOPTIC VIEW OF THESIS:-

In the second chapter of this thesis we study Chomsky's classification of grammars, particularly the context free grammars and survey methods of analysing context free grammars.

We do not consider context free grammars in general any further and are mainly interested in a generalised version of Lewis and Stearns' [LEWIS 68] LL(k) grammars. These form a fairly large subset of context free grammars which can be parsed without back-up.

In the third chapter we describe the metasyntactic language for our parser. This is a modified version of Backus Normal Form (MBNF) which includes left recursive productions. Techniques are described to improve the efficiency of the parser and to reduce the length of look ahead. It is also shown how look ahead can be introduced and scanning of the source text controlled by using SNOBOL4 extension programs.

The 4th chapter covers the implementation of the parser. The source language symbols are recognized in a predictive fashion. The parser uses a syntax graph and a syntax analysis stack. Starting from the root of the syntax graph, it traverses different nodes following predefined hierarchically ordered paths without ever backing up and recognizes the source language symbols in the process.

Our metasemantic language (MSEAL) is explained in the fifth chapter. Each production of MSEAL consists of three fields : environment field, action field and code field.

The environment field is used to determine the instant at which the particular semantic production is to be used.

The action field consists of a sequence of statements specifying actions to be taken when the environment field is recognized in processing source text. High level commands have been provided to facilitate the construction by the user of commonly used

data structures.

Extra power is provided by the code field. The user can specify a codestring in this field. On meeting certain commands in the action field the corresponding code field is executed and code generated.

The implementation of MSEAL is discussed next. Using the semantic statements, tables are constructed. As the recognition of a source statement proceeds, these tables are checked and at an appropriate stage some semantic statement executed.

To provide extra power for the complete translation system (MTL), it has been designed to be semantically extensible. SNOBOL4, the implementation language for the MTL processor, appears as a subset of MTL. Methods have been provided to use SNOBOL4 for specifying both the syntax and the semantics of a programming language. It is also possible to extend semantically the facilities available in the action field of MSEAL productions. Various MTL system variables are used to provide communication

between MTL and SNOBOL4.

The description and implementation of the extension mechanism is given in the seventh chapter of this thesis.

In the next chapter we discuss the programming language PATE. It had its basis in SNAP [BARNETT 69] and is a language for arts students. It was implemented at the start of this project and for the reasons described in that chapter, it was implemented in FORTRAN IV[IBM 360/370].

Finally, we conclude by discussing the results obtained.

1.3 CONDITIONAL EXPRESSIONS AND OTHER NOTATION.

Greek letters represent terminal strings. $|\beta|$ denotes the length of the string β. $\beta{:}n$ refers to the left-hand n symbols of β if $|\beta| \geq n$ and to β otherwise. The empty string is Λ. Lower case letters represent terminals and upper case letters are nonterminals. Underlined letters may

represent either. The node of the syntax graph representing $\underline{X}$ (or X or x) is written as $\underline{X}_{NODE}$ (or X_{NODE} or x_{NODE}). Rules of the metasyntactic language are marked by an asterisk.

While BNF is suitable for the representation of a single context free grammar, a formal method to represent sets of such grammars is desirable. It will enable us to discuss the behaviour of precisely defined sets of grammars in the context of our parsing algorithm. For this purpose we use a notation for a conditional expression which was suggested by that of McCarthy [McCARTHY 60]. It not only fulfils the above requirement but also specifies the order in which different productions may be recognised, and hence reveals certain features of the recognition algorithm. The source language statement is considered as a sentence of a context free language whose grammar is written in BNF and which is being recognised by a top down left to right process.

Our conditional expression is written as follows:

(1.3.1)
$$C_n^1\ L_n^1\ V_n^1,\ C_n^2\ L_n^2\ V_n^2,\ C_n^3\ L_n^3\ V_n^3\ \ldots\ldots$$

$$C_{n-1}^1\ L_{n-1}^1\ V_{n-1}^1\ C_{n-1}^2\ L_{n-1}^2\ V_{n-1}^2,\ C_{n-1}^3\ L_{n-1}^3\ V_{n-1}^3\ \ldots\ldots$$

$$C_{n-2}^1\ L_{n-2}^1\ V_{n-2}^1,\ C_{n-2}^2\ L_{n-2}^2\ V_{n-2}^2,\ C_{n-2}^3\ L_{n-2}^3\ V_{n-2}^3\ \ldots\ldots$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$C_0^1\ L_0^1\ V_0^1,\ C_0^2\ L_0^2\ V_0^2,\ C_0^3\ L_0^3\ V_0^3\ \ldots\ldots$$

The occurrence of the triple $C_i^m\ L_i^m\ V_i^m$ has the effect of returning the value V_i^m when the condition C_i^m is found to be satisfied. C_i^m has the form $(\alpha_i = \beta)$ and is satisfied if the substring under consideration, α_i, has the form β. Each triple has a level which is given by the subscript i of the condition C_i. L is defined below.

Evaluation of the conditional expression takes place in descending order of level and looking from left to right at each level. The value V of the whole expression is the concatenation of

(1.3.2) V_{n-p}, V_{n-p}" ..... $V_{n-p}{}^{n}$

where the value of $V_{n-p}{}^{r}$ is the concatenation of

(1.3.3) $V^{t}_{n-p}{}^{r}$ $V^{t'}_{n-p}{}^{r}$ $V^{t"}_{n-p}{}^{r}$ .....

In (3) the value of p increases with superscript r; p and t are such that V consists solely of source language symbols.

L_i may be either "->" or "->>". In both cases one and only one condition at the current level may be satisfied. The symbol "->" implies that the triple is applied once only, whereas "->>" implies that the triple is applied n times where $n \geq 0$.

It follows that if no L_i has the form "->>" then exactly one condition must be satisfied at the level i. This notation can be further generalised by ranking the conditional expressions themselves and treating them in the same manner as that of the above mentioned triples.

EXAMPLE 1.3.1

One might wish to specify the following rules about the recognition procedure for the grammars of the form:

(1.3.4) $\langle x \rangle = a$

(1.3.5) $\langle x \rangle = \langle x \rangle\ aa$ Gl.1

(a) Find the left recursion

(b) Process (5) the correct number of times,

(c) Process (4) and recognise the source string with the help of already processed productions.

Using (1) these rules can be stated as

(1.3.6) $[\ (\alpha_2=\Lambda)\rightarrow n=0,\ (\alpha_1=\beta_1)\rightarrow\rightarrow\ n=n+1, (\alpha_0=\beta_0)\rightarrow \beta_0\beta_1^n\ ]$

for a set of all left recursive grammars of which the above grammar is a member. (For Gl,1 β_1=aa and β_0=a).

Suppose aaaaa is a sentence of L (Gl.1) and is to be recognised using (6).† The cursor is considered to be on the left of the source statement. The condition to be tested first is $\alpha_2 = \Lambda$ since 2 is the highest level in (6), any part of the string aaaaa or a

† Reference to a relation inside the current section is implied.

null string can be considered as the value of α (α_2 as it is represented in (6)). So ($\alpha_2 = \Lambda$) is satisfied and n is initialised to 0, but the cursor is still on the left of the source string (step a). The next condition to be tested is ($\alpha_1 = \beta_1$). If we take α_1, to be aa, the condition is satisfied, n is set to 1 and the cursor position remains unchanged (step b). The condition ($\alpha_1 = \beta_1$) is tested again. Since α_1 can still have the value aa, it is satisfied. n is set to 2 and the cursor position still remains unchanged (step b). The condition† ($\alpha_1 = \beta_1$) is attempted again but it can not be satisfied since α_1 can only have the value a. The value of α and the cursor position remain unchanged. The condition ($\alpha_0 = \beta_0$) is now attempted and satisfied. At this state $\beta_0\beta_1^n$ is recognized and the cursor moves to right of the source sentence and the recognition of the source sentence is successfully completed. (step c) The value of $V^3_{n-p}r$ (refer to (3)) is $\beta_0\beta_1^n$ which is the same as aaaaa. In (2) it is represented by V_{n-n}. Other values in (2) are $V_{n-(n-1)} = (n = n+1)$

$V_{n-(n-2)} = (n = 0)$

† Since it is not possible to determine whether ($\alpha = \beta$) is satisfied or not, the conventional top down parsers do not accept left recursion.

The only value consisting solely of the source language symbols is that of V_{n-n} and hence it is the value of V.

EXAMPLE 1.3.2

A production containing direct right recursion can be represented as

(1.3.1) $[(\alpha_1 = \beta_1) \rightarrow\gg \beta_1, (\alpha = \beta) \rightarrow \beta]$

For the right recursive grammar

$<x> = a <x> \mid b$ G1.2

$\beta_1 = a \quad \beta = b$

EXAMPLE 1.3.3

1.4 A set of productions representing embedded recursion might be written as

(1.4.1) $(\alpha_2 = \Lambda) \rightarrow n = 0, (\alpha_1 = \beta_1) \rightarrow\gg \beta_1$ and $n = n + 1,$

$(\alpha = \beta) \rightarrow\gg \beta\ \beta_2^n]$

$\beta_2 = f(\beta_1)$

The following is a grammar representing

embeded recursion

<x> = a <x> b | c Gl.3

In this grammar

β_1 = a β = c β_2 = b

CHAPTER II

On the Syntactic description and Parsing Programming languages

2.1 INTRODUCTION:-

To make clear the reasons for our choice of context free grammar we begin by surveying grammatical models which have been used in earlier projects. These projects fall into two classes - those where sentences must be generated and those which require recognition of correct sentences. Existing grammatical theories have helped researchers in making some progress in the field of machine translation but the net result is far from satisfactory, because either they are not powerful enough or are too difficult to be handled by computers (see survey articles by Sage [SAGE 67] and Satterthwait [SATTERTHWAIT 66] and see article by Floyd [FLOYD 64]). The generative mechanism has been used by researchers [FREDMAN 62, 71 etc] who want to use computers as a tool for studying properties of grammars.

Some question answering systems have been developed which use grammars both for recognising input sentences and generating grammatically correct sentences in reply. PROSE [VIGOR 69] is such a system. It uses Hay's Dependency grammar [HAYS 64] as its grammatical model. However as far as programming languages are concerned Chomsky's models [CHOMSKY 57] of generative grammars have obtained the widest acceptance. By this we do not imply that his models are sufficient for all further developments in programming languages but we only note that the structure of most of the programming languages so far in existance, either intentionally or unintentionally has been designed so as to fit Chomsky's models.

We will follow Chomsky in describing his generative models. Similar systems to accept only correct sentences are well known. The simplest model discussed by Chomsky is the finite state grammar. This can be described in the form of a machine that can be in any one of a finite number of different internal states. This machine switches from one state to another by producing a certain symbol. One of these states is distinguished

as the initial state while another is the final state. Beginning from the initial state, if the machine runs through a sequence of states and reaches the final state, it will generate a sequence of symbols known as a sentence. The complete set of sentences that can be produced in this way is called a finite state language and the machine is known as a finite state grammar. The recognition of a finite state language may be performed by a finite automaton which will in general be nondeterministic. It has been proved [RABIN AND SCOTT 59] that every nondeterministic finite automaton can be represented by some deterministic finite automaton, which we know can always be simulated.

Unfortunately only a small number of programming languages are finite state. Any attempt to construct finite state grammar for others will run into serious difficulties. For example general bracketed expressions require a context free grammar and the same is true for many features of well structured programming languages.

2.2 PHRASE STRUCTURE GRAMMARS:-

Suppose S is the initial symbol and

(2.2.1) $IC_1(S)$, $IC_2(S)$, $IC_3(S)$ ----------$IC_n(S)$

are its immediate constituents [BLOOMFIELD 33], derived using rules usually known as productions.

Let us write (1) as follows

(2.2.2) S_1^1, S_2^1, S_3^1 ----------S_n^1

Their immediate constituents will be as follows, although any one of them can be null.

(2.2.3) $IC_1(S_1^1)$, $IC_2(S_1^1)$, $IC_3(S_1^1)$ ----------$IC_n(S_1^1)$

$IC_2(S_2^1)$, $IC_2(S_2^1)$, $IC_3(S_2^1)$ ----------$IC_n(S_2^1)$

$IC_1(S_3^1)$, $IC_2(S_3^1)$, $IC_3(S_3^1)$ ----------$IC_n(S_3^1)$

$$IC_1(S_n^1),\ I_2(S_n^1),\ IC_3(S_n^1)\ \text{----------}IC_n(S_n^1)$$

If we write the non-null constituents as

(2.2.4) $$S_1^2,\ S_2^2,\ S_3^2\ \text{----------}S_n^2$$

and continue the process we will finally reach

(2.2.5) $$S_1^n,\ S_2^n,\ S_3^n\ \text{----------}S_n^n$$

so that they do not have any constituents.

If we call the above model a context free grammar, the set of all the representations of S (with subscript and superscript including the start symbol) is known as its vocabulary. All symbols that can not be further broken down are terminals and rest of the vocabulary is formed by nonterminals. A sentence is a string of terminals which can be derived from S with the productions concerning immediate constituents. The language is the set of all the sentences that can be produced from the grammar.

Formally a context free grammar is a quadruple $G(V_T, V_N, S, P)$, where V_T is a finite set of terminals, V_N a finite set of nonterminals with $V_n \cap V_T = \Lambda$, $V = V_N \cup V_T$. S is the initial symbol and $S \in V_N$. P is a finite set of productions of the form $A \rightarrow \omega$ where the left part is $A \in V_N$ and the right part $\omega \in V^*$ where V^* denotes a string of symbols of V.

A string α is called a sentential form if α is derivable from the initial symbol S. A sentence is a sentential form consisting only of terminals. The language L (G) is a set of all the sentences that can be generated from the grammar G.

It should be noted that the finite state grammars form a proper subset of the context free grammar in the sense that every finite state grammar has an equivalent context free grammar while the converse is not true.

Let G be a grammar. We say that the string X directly produces the string ω, written

$$X \Rightarrow \omega$$

if X -> ω

If X => ω^{n-1} => ω^{n-2} =>ω^{n-3} .... => ω

then X =>+ ω

If either X => ω

or X =>+ ω

then X =>* ω

If X => ω ... where three dots "..." represent a string possibly empty.

then X FIRST ω

If X FIRST ω^{n} FIRST ω^{n-1} FIRST ω^{n-2}

---------- FIRST ω.

then X FIRST + ω

If either X FIRST ω

or X FIRST + ω

then X FIRST* ω

Let ω = x u y be a sentential form.
Then u is called a phrase of the sentential form ω for a nonterminal U,

if S => x Uy and
U =>+ u, u is called a simple phrase

if S => *xUy and U => u.

A handle of any sentential form is a leftmost simple phrase.

If U => + ... U...

we say the grammar is recursive in U.

If U => + U ... it is left recursive;

If U => + ...U it is right recursive.

A sentence of a grammar is ambiguous if there exist more than one derivations for it. A grammar is ambiguous if it can generate an ambiguous sentence.

"Phrase Structure grammars" is a name given by Chomsky to what the Bloomfieldian linguists [LYONS 70] originally called the immediate constituent analysis. They are also commonly known as context free grammars (CFG). These grammars are formally equivalent [GAIFMAN 65] to Hays [HAYS 64] dependency grammars. Two grammars are equivalent if both produce precisely the same set of sentences with the same ambiguities. Chomsky provided a formalization of CFG [CHOMSKY 57] and demonstrated that, in spite of being more powerful than finite state grammars in the sense that more languages can be described by this model CFG's have certain limitations.

1) A suitable CFG is capable of generating almost all the sentences of English but in many cases fails to generate all structural descriptions which may result in ambiguous meaning. For instance the sentence "Flying planes may be dangerous" can be generated by a phrase structure grammar but its two quite different descriptions can not be distinguished by this model of grammar.

2) CFG's do not provide a method to show semantic relations between different sentences. For example there is no way of stating that if one of the following statements is true, the validity of the other statement is implied.

   a) Yesterday I rode a horse.
   b) I rode a horse yesterday.

3) The syntax of programming languages can be represented in CFG by writing it in BNF [BACKUS 59] but there is no formal way of including semantics in the grammar. This inadequacy has serious implications in compiler writing.

4) When CFG is to be used for syntactic analysis, it is sometimes stored in machine in the form of a syntax tree (or a syntax graph). There are sentences which use common vocabulary and are semantically related but large parts of their syntax trees are separate - hence wastage of space. For example

a) I shall give the girl, a book.
b) I shall give a book to the girl.
c) Shall I give a book to the girl?

A transformational grammar as defined by Chomsky assigns to each sentence it generates, both deep structure and surface structure analysis and systematically relate the two. Deep "connections" between sentences which cut across the surface grammar are transformational rules. The phrase structure rules are used to generate underlying strings and on applying transformational rules on these strings we obtain sentences.

The first two points mentioned in connection with CFG can be accounted for by transformational grammars but it is difficult to say anything about the last two points, since they have not been studied in great detail. Attempts were made to devise a formal method of representing languages in transformational grammar. Further research was abandoned since the productions required for such a representation grew exceedingly complex.

Transformational grammars have not been used much for analysis of programming languages and most of the work concerning them is confined to using computers as a tool for linguistic research [FRIEDMAN 71] . It is mainly due to the complexity of transformational rules. It is argued that programming languages do not require the "power" of transformational grammar, since they normally do not posess active, passive, exclamatory and similar interrelated sentences. We note, however, a recent paper describing work at the University of California [DEREMER 74] .

2.3 PARSING TECHNIQUES:-

Parsing techniques can be devided into two main categories: bottom up or data directed methods and top down or goal oriented methods. There are parsing methods which do not fall in any one of these categories, however, most of them are very ad hoc and do not form a model of any significant generality. We quote Conway's [CONWAY 63] remarks about his parsing technique, which requires the construction of so called no-backup diagrams:

> "The catch in all this is that a set of no-backup diagrams for a given language is constructed by a process which is neither straightforward nor easy to describe .....".

We therefore will confine our discussion to the two main categories of parsers mentioned above.

2.4 BOTTOM-UP TECHNIQUES:-

In this method we analyse the given language by repeatedly finding the handle β of the current sentential form and reducing it to a nonterminal B using a production

$$B \rightarrow \beta .$$

The problem with the bottom-up method is to find the handle and then to know which nonterminal to reduce it to.

Wirth and Weber [WIRTH 66 see also HASKELL 74] have developed a bottom-up parsing technique for a class of CFG's. In this class no two productions have the same right hand sides and at most one so-called precedence relation holds between any two symbols of the vocabulary. This class is known as simple precedence or (1,1) precedence grammar. The precedence relations are stored in a matrix.

The above mentioned authors have also pointed out that the space required for the matrix is very large (of the order of n^2, where n is the number of symbols in the vocabulary)

in most practical programming languages. To counteract that, they developed the notion of precedence functions which reduces the space requirement. Formal methods have now been presented to calculate precedence relations [MARTIN 68] and precedence functions [BELL 69]. However, it is not possible to construct a simple precedence grammar for every CFG and further, it is not possible to derive precedence functions for all simple precedence grammars.

A technique which is similar to the simple precedence technique but requires a considerably smaller matrix is called operator precedence and the grammars it handles are called operator precedence grammars. This technique has been given this name because terminals of the grammar play the part of operators and nonterminals are treated as operands. The parsing algorithm used for simple precedence grammars is applicable except that all relations are among the terminals only.

Obviously this requires a smaller matrix.

Higher order precedence techniques can parse a bigger subset of CFG then the simple precedence methods, but these methods are normally too demanding on space. In many cases the space requirements can be reduced to a reasonable limit by using some ad hoc techniques.

Another difficulty with precedence parsing techniques is that all the right parts of productions must be unique. Attempts to get rid of this restriction led to the development of bounded context grammars. [PAUL 62 , FLOYD 64 , IRONS 64 , GRIES 71] In the bounded context schemes we use symbols on either side of the deleted handle to find out what it should be reduced to.

Operator precedence grammars are only a special case of bounded context grammars. Parsing algorithms for bounded context grammars use a three column table in addition to the usual space requirement for holding all the rules of grammar.

The construction of tables is always complex enough but gets even more difficult in certain cases ((m,n) bounded context grammars). Also there is no direct way of finding out whether a grammar is bounded context or not. Knuth [KNUTH 65] has investigated LR(k) grammars.

A grammar is called LR(k) if, for $\omega_1, \omega'_2, \omega_3, \omega'_3$ in V_T^*, A in V_N, and P, P' in G, $S \Rightarrow \omega_1 A \omega'_3$, $A \Rightarrow_P \omega'_2$, $A \Rightarrow_{P'} \omega'_2$ and $(\omega_2\ \omega_3)$; $k = (\omega'_2\ \omega'_3)$: k imply $P = P'$.

Stated informally in terms of parsing, an LR(k) grammar is context-free grammar such that for any word in its language each production in its derivation can be identified and its descendants determined with certainty by inspecting the word from its beginning (left) to the kth symbol beyond the right most descendant.

Knuth points out that almost all unambiguous grammars which can be processed by some left to right process are LR(k). In fact

precedence grammars $\subset$ bounded context grammars $\subset$ LR(k) grammars.

Knuth has also shown how one can determine whether a grammar is LR(k) for a given k. The problem of deciding, for a given grammar G, whether or not there exists a $k \geq 0$ such that G is LR(k), is however undecidable. He has shown a CFG, named by him as LR(k,t) grammar, for which we must back-up by a finite amount. He also points out that the parse time for LR(k) grammars is essentially proportional to the length of the string to be parsed.

Deremer [DEREMER 69] has given a practical algorithm for parsing LR(k) grammars. The complications involved in constructing a parser for LR(k) grammar vary directly as the complexity of the grammar and Deremer defined a hierarchy of LR(k) grammars given in the ascending level of complexity by

1 LR(0)

2 SLR(k) †

3 LALR(k) †

4 LR(k)

† Simple LR(k)

†† Look ahead (LR(k)

Three different subsets of LR(k) grammars are defined in terms of their parsing algorithms. Deremer has shown that SLR parsers for SLR(k) grammars can parse a large number (if not all) of languages that can be handled by precedence techniques or bounded context grammars.

Knuths LR(k,t) grammars overlap with grammars which are LR(k) and are not LALR(k) in Deremer's terminology. The parsing technique for this class of grammars has not been described with the same details as given for SLR(k) and LALR(k) parsers. Nevertheless one thing is clear: as pointed out by Deremer, it is exceedingly difficult to construct a parser for this class of grammars.

A biblography on LR(k) grammars appears in a tutorial paper by Aho and Johnson [AHO 74]

2.5 TOP DOWN TECHNIQUES

These techniques work by starting from the initial symbol of the grammar and recognizing a sentence by working through its productions. In his survey of parsing techniques Floyd [FLOYD 64] has mentioned only backup oriented techniques. However the notion of no-backup techniques was in existance long before [KANNER 59] the publication of his paper.

Methods have been described [LIETZKE 64] to perform top down analysis by scheduling different procedures. The idea is that an appropriate procedure should be called at the appropriate place and every procedure should have a specific task to perform. These techniques though adequate for certain languages do not form a general model for any appreciable subset of CFG's.

The Global parsing technique [UNGER 68] works without backup but is not capable of handling grammars with cyclic nonterminals. Also, it requires the whole of the sentence

to be available to the parser before parsing begins. Unger has given various "quick checks" to be performed to make the parser efficient but many of them are reported not to have been studied in detail.

Lewis and Stearns [LEWIS 68] have defined syntax oriented transductors which perform both syntactic and semantic analysis. This model, however, has not been used in any compiler so far to the author's knowledge.

LL Parsers [LEWIS 68, ROSENKRANTZ 69] (discussed in the next chapter) can be considered as the top down counterpart of Knuths LR techniques. The grammars that can be handled by LL parsers are known as LL(k) grammars (defined in the next chapter). An LL(k) grammar is a CFG such that for any word in its language, each production in its derivation can be identified with certainty by inspecting the word from its beginning (left end) to the k-th symbol beyond the beginning of the production. Thus when a nonterminal is to be expanded during a top down parse,

the portion of the input string which has been processed so far plus the next k input symbols determine which production must be used for the nonterminal.

2.6 COMMENTS ABOUT PARSING TECHNIQUES:-

The efficiency of different parsers is notoriously difficult to compare. It is not only dependent on a particular language but also on the manner in which its grammar is written.

It may seem reasonable that in making the above mentioned comparison among different parsers, language should be a constant factor and its grammar in each case be written so as to suit the particular parser. However, no general conclusion can be drawn from such a comparison. Since, for almost any general parsing method known, there are languages (or sentence in languages) which make it drastically inefficient. Comparison of different parsers on theoretical grounds is very difficult, if not impossible since

many algorithms differ from each other substantially.

Griffiths and Petrick [GRIFFITHS 65] have made a comparison of parsing techniques and have concluded that top-down parsers are grossly inefficient timewise as compared to bottom up parsers. Brooker [BROOKER 67] has criticised their conclusion on the grounds that the grammars of a large number of programming languages can be written so as to make their top-down parsers efficient. We agree with Brooker's remarks and add that a better general purpose parser is one which is efficient for bigger subsets of languages, and for more languages. Hence, a fairly general purpose no-backup parser is better than the backup oriented parser, the reason being that in a backup oriented parser, efficiency will be achieved by writing the grammar so as to minimize backup while a no-backup parser by definition possesses this property in its extreme form.

Space efficiency is as important as time efficiency. A parser can be inefficient in space either because it is too big by itself or because it requires a large space

to store information produced during the parse of a given sentence. The former inefficiency is common in bottom up parsers where large matrices are needed either for some type of precedence relations or for storing look ahead symbols. The latter inefficiency is usually found in backup oriented parsers where a lot of information is required, in case the parser has to backup.

Horning and Lalonde [HORNING 71] have made an empirical comparison of the time and the space efficiencies of two general classes of bottom up parsing techniques, namely precedence techniques and LR parsers. But due to the reasons given above they do not claim to have reached any definite conclusion about the relative efficiencies of precedence techniques and LR parsers in general. However, they claim that by using Deremer's LALR† algorithms and after using various optimizations suggested by him and on including new optimizations we get a parser which is worth considering when selecting a parsing technique for a compiler.

† The investigation is limited to LALR parsers.

Ease of use is another important feature in parsers. A technique could be quite difficult to use if it is capable of treating only a special class of grammars defined by conditions which are not easy to state directly. Some parsers are very difficult to construct even if it is known that the grammar being treated is suitable for them. Both of these problems are common with bottom up parsers. In this regard a parser which can accept grammar of a given language in some modified version of BNF can be quite useful. A top down technique is valuable for this purpose and provides a natural way of constructing internal tables. It has been used in many such systems [METCALFE 65].

A good parser should also be able to give reasonable syntactic diagnostics, since they are essential for program debugging, no-backup parsers are an asset in this respect.

With these comments we shall attempt to define an ideal parser. The nearer to this definition the better.

An ideal parser for CFG's is one which accepts all grammars written in BNF, requires the minimum possible space necessary to store the parser, and requires no space for storing specifications of parsing and parses without backup giving prompt and precise diagnostics.

CHAPTER III

GENERALISED LL(k) PARSER

3.1 INTRODUCTION:-

This chapter is devoted to the discussion of the metasyntactic language. Rules of the metasyntactic language are given and, where necessary, they are explained with the help of examples. It is shown how using some of these rules, the value of k can be reduced. It is also shown that in spite of the fact that Rosenkrantz and Stearns have proved that the left recursive grammars are not LL(k), this restriction is not valid for our algorithm. The class of grammars defined by the property that they are accepted by our algorithm therefore constitute an extension of the LL(k) class.

3.2 RECOGNITION OF LL(k) GRAMMARS:-

As defined by Lewis and Stearns [LEWIS 68] a grammar is LL(k) if, for all $\omega_1, \omega_2, \omega_2', \omega_3, \omega_3'$

in V^*_T, A in V_N, and p, p′ in G,

$S \Rightarrow \omega_1 A \omega_3$, $S \Rightarrow \omega_1 A \omega_3'$,

$A \underset{p}{\Rightarrow} \omega_2$ $\quad$ $A \underset{p'}{\Rightarrow} \omega'_2$

and $(\omega_2\omega_3)\colon k = (\omega_2'\omega_3')\colon k$ imply $p = p'$

Stated informally a grammar is LL(k) if a production and its leftmost descendant can be identified from the symbols to the left of this leftmost descendant and the k symbols which follow (counting the leftmost descendant terminal as the first symbol).

Before we show that a deterministic pushdown machine can be constructed to recognise the language generated by a given LL(k) grammar we shall prove a lemma.

Let L be the language generated from an LL(k) grammar G using nonterminals V_N and terminals V_T. Define L_A as follows: For A in V_N, let L_A be a set of words in V^*_T generated by G using starting symbol A; for A in V_T, $L_A = \{A\}$
If R is a subset of V^*_T, let $R{:}k = \{\omega{:}k \mid \omega \text{ in } R\}$

LEMMA. If G is LL(k), then for all A in V_N, ω in $V^*_T\colon k$, and $R \subseteq V^*_T\colon k$ satisfying $\mathrm{RCR}(\omega_1) = \{\omega_3{:}k \mid S \Rightarrow \omega_1 A \omega_3\}$ for some ω_1 in V^*_T, there exists

at most one production p such that $A \Rightarrow^* \omega_2(p)$ and $(\omega_2\omega_3):k = \omega$ for some ω_2 and ω_3 in V^*_T such that ω_3 is in R.

PROOF. Suppose that $A \Rightarrow \omega_2(p)$, $A \Rightarrow \bar{\omega}_2(\bar{p})$, and $(\omega_2\omega_3):k = (\bar{\omega}_2\bar{\omega}_3):k = \omega$ for some $\omega_2, \bar{\omega}_2, \omega_3, \bar{\omega}_3$, p, and $\bar{p}$ such that $\omega_3:k$ and $\bar{\omega}_3:k$ are in R where $R \subseteq R(\omega_1)$ for some ω_1. It follows that there must be ω'_3 and $\bar{\omega}'_3$ in V^*_T such that $S \Rightarrow^* \omega_1 A \omega'_3$, $S \Rightarrow^* \omega_1 A \bar{\omega}'_3$, $\omega'_3:k = \omega_3:k$, and $\bar{\omega}'_3:k = \bar{\omega}_3:k$. The last two relations imply that $(\omega_2\omega'_3):k = (\bar{\omega}_2\bar{\omega}'_3):k = \omega$, and the fact that $p = \bar{p}$ follows from the definition of LL(k). Thus there is at most one such p.

The importance of this lemma can be stated informally as follows: The definition of LL(k) grammars specifies that for any sequence in the language, a production can be correctly identified from the sequence ω_1 of symbols to the left of its first descendant and the k symbols which follow. The lemma states that the production can also be identified using only the k symbols which follow and the set $R(\omega_1)$ where $R(\omega_1)$ is the set of all k symbol sequences which can follow the rightmost descendant of that production. In other words, we can bound the

amount of information that must be remembered about the initial sequence.

3.3 DESCRIPTION OF THE PUSHDOWN MACHINE.

The finite-state control has enough memory to store an input string of length k or less and to perform such obvious tasks as reading in the first k inputs. The tape symbols are ordered pairs (A,R), where A is an element of $V_N \cup V_T$ and R is a subset of $V_T^* : k$. We design the machine so that if some r + k inputs have been read in, the input string stored in the finite-state control is ω and the top tape symbol is (A,R), then the following are true.

(1) The word ω stored in the finite-state control is the string consisting of the (r + 1)-th input through the (r + k)-th input. If the input word only has r + k′ symbols for $k' < k$, then ω is the last k′ symbols of the input word. In this latter case, it is convenient to say that k - k′ blank inputs have been read in after the completion of the input word as indicated by the special end-of-tape marker. These implicit blanks play the same role as the

"$\dashv$" of Knuth.

(2) The symbol A represents the fact that the descendants of an A follow the rth input symbol. If A is a terminal symbol, this means that the (r + 1)-th input symbol must be an A. The symbol pair (A,R) or its replacement is to be popped up as soon as all the descendants of A have been identified.

(3) The set R represents $\acute{R}$: k, where $\acute{R}$ is the set of all acceptable input sequences that could follow the descendants of the A. Thus, if (A_1,R_1) is the tape symbol below (A,R), then $R = (L_{A1}, R_1)$: k; and $R = \{\Lambda\}$ if (A,R) is the bottom tape symbol. The machine begins with the single symbol $(S, \{\Lambda\})$ on its pushdown tape.

We now describe the machine operations. Initially, the machine reads the first k inputs and stores them as the word ω in the finite control. The pushdown tape is initialised with the symbol $(S, \{\Lambda\})$. This initialised configuration satisfies 1, 2 and 3 above and we take it as self-evident that the operations described below preserve these properties. After r + k inputs have been

read, the operations are as follows:

Case 1. If the top tape symbol is (A,R) and A is a nonterminal, then $R = \{\omega_3 : k \mid S \Rightarrow^* \omega_1 A \omega_3^1\}$, where ω_1 consists of the first r inputs. Therefore, by the lemma, there is at most one production p that could be applied to A in order to be consistent with ω and R. Three subcases follow:

Case 1a. If there is no such p, the machine rejects the sequence.

Case 1b. If p is the production $A \Rightarrow \Lambda$ then the top tape symbol is popped off.

Case 1c. If p has the form $A \Rightarrow A_1 \ldots A_m$ for A_i in $V_N \cup V_T$, then the top symbol (A,R) is replaced by the sequence of symbols $(A_1, R_1) \ldots (A_n, R_n)$, where $R_n = R$ and $R_{i-1} = (L_{A_i}, R_i) : k$ for $1 < i \leq n$.

Case 2. If the top tape symbol is (A,R) and A is a terminal, there are two subcases:

Case 2a. If $\omega = A\,\omega'$ for some ω', then the top tape symbol is popped off, the next input x is read, and word $\omega' x$ replaces ω

in the finite control.

Case 2b. If ω does not begin with A, then the sequence is rejected.

Case 3. If there are no tape symbols on the pushdown tape, then if $\omega = \Lambda$, the sequence is accepted and otherwise it is rejected.

Although considerable information is encoded in the tape symbols (A,R), this is somewhat less information than is required for general LR(k) recognition. Furthermore, even this R information is needed only when the machine must choose among words in L_A which are shorter than k. To verify this, assume that (A,R) is the top tape symbol (i.e. the machine is looking for a production descendant from A) and that control word ω is the L_A (i.e. that the next k symbols after the start of A are all descendants of A). Then it follows from the definition of context-free grammars that the past can give no information as to which A production was used, and hence the decision is independent of R. This situation always occurs for LL(1) grammars if there are no Λ productions.

```
<X> = 0 1 ¬ 1
<Y> = 0 ¬ 1
<B> = 0 ¬N 0
<S> = 1 <X> <B> | 0 <Y> 0 <B>
%SOURCE
1 0 1 0
1 0 1 1 0
1 0 1 1 0
1 0 1 1 0
1 0 1 1 0 0
1 0 1 1 0 0 0
1 0 1 1 0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0 0
0 0 0 0 0 0
0 0 1 0 0
0 0 1 0 0 0
0 0 1 0 0 0 0
0 1 4 3 0
   £
*** THIS SYMBOL = 0 OR ITS EQUIVALENT
1 1 1 1 1
   £
*** THIS SYMBOL = 0 OR ITS EQUIVALENT
1 0 1 0 1 0 1 0
         £
*** ERROR FOUND
```

TABLE 3·1

ELL (0) version of the LL(3) grammar given in example 3.1

As an example where R information is necessary, consider the following grammar:

S => 1A1B

S => 0A0B

A => 0

A => 01

B => 0B

B => 0

EXAMPLE 3.1

This grammar is LL(3), but after 1 + 3 inputs have been read and ω = 010, one cannot determine which production to apply to A without consulting the corresponding R which will contain either {10} or {00} , depending on which production was applied to S.

3.4 GENERALISED LL(k) GRAMMAR:-

We define a generalised LL(k) grammar as follows:

A grammar is generalised LL(k) if, it can be written with the help of the metasyntactic language described in (3.5) and for all $\omega_1, \omega_2, \omega_2', \omega_3, \omega_3'$ in V_T^*, A in V_N and p , p' in G, $S \Rightarrow \omega_1 A \omega_3$ $S \Rightarrow \omega_1 A \omega_3'$,

$A \underset{p}{\Rightarrow} \omega_2$ $\quad$ $A \underset{p'}{\Rightarrow} \omega'_2$

and $(\omega_2\omega_3) : k = (\omega'_2\omega'_3) : k$ imply $p = p'$

or P and p' are left recursive as explained in § (3.8).

The definition of generalised LL(k) grammars is similar to the one given in (3.2) except that the if clause of the definition is further qualified by saying "if it can be written with the help of the metasyntactic language described in (3.5)" and "or P and P' are left recursive ".

It is clear that the definition of generalised LL(k) grammars is more powerful than that of LL(k) grammars. We therefore abbreviate the name to extended LL(k) written ELL(k). They include left recursive grammars and the rules of the metasyntactic language are more powerful than that of ordinary BNF.

3.5 ON THE PRACTICAL ALGORITHM OF GENERALISED LL(k) GRAMMARS:-

The practical algorithm for ELL(k) grammars is described in chapter 4. It is not difficult to see that in combination with the extension mechanism described in chapter (7) it performs

the task of the push down machine described in (3.3).

The word ω (steps 1 of the push down machine) is stored in the input buffer of the syntax analyser. R is stored by the extension program in SNOBOL4. Step 2 is performed by building a syntax graph and stacking in the node corresponding to the start symbol on the syntax analysis stack. Steps (a) to (m) of the syntax analyser correspond to the cases (1) to (3) in the push down machine.

Different facts about the LL(k) grammars are listed in appendix I. We will not prove these facts for ELL(k) grammars. We believe that, with the exception of the restrictions on left recursion , they can be proved by arguments similar to those used for LL(k) grammars.

3.6 METASYNTACTIC LANGUAGE:-

* Terminals stand for themselves.
* Nonterminals are enclosed in corner brackets "< " and " > " or in ampersands.
* The left hand side of a production is separated from its right hand side by "=".
* EMPTY is the system defined nonterminal representing Λ .

* ⟨BLANKS⟩is the system defined nonterminal representing zero or more blanks.
* Any terminal being followed by another symbol in a production requires a blank as a terminator.
* All members of the metasyntactic language except SNOBOL4 (Explained later) are normally reserved symbols.
* Any reserved symbol when preceded by an asterisk loses its special meaning.
* The grammar is written such that (a) A linking production (explained later) follows the production it links. (b) The production having the start symbol of the grammar on its L.H.S. may only be followed by linking productions. The existence of blanks in the source language statements can be specified explicitly on the right hand sides of MBNF productions. If a nonterminal is enclosed in corner brackets it is assumed that after the recognition of its rightmost descendant, at least one blank will follow in the source language statement. All blanks are ignored. On the other hand if a nonterminal is enclosed in ampersands, no assumption about the character to follow its rightmost descendant is made.

For example consider

3.6.1 < letter > = A|B|C|D .....| Z

3.6.2 < variable > = &variable& &letter& | &letter& G2

3.6.3 < list > = < list > < variable > | < variable >

Its language consists of character strings separated by blanks. Character strings are recognised by (1) and the blanks are introduced due to (3). The manner in which the left hand sides of these productions are specified is not important. SNOBOL4 can be considered as a subset of the metasyntactic language. During the syntax specification, one or more SNOBOL4 programs can be introduced. The code generated is the same as that generated by the SNOBOL4 compiler and no substantial loss of efficiency is incurred.

Undefined nonterminals of MBNF are considered to be the names of SNOBOL4 programmer defined functions and the user is assumed to have defined them in his SNOBOL4 programs. On execution, when any such nonterminal is encountered, linkage to the appropriate function is made automatically. Within a SNOBOL4 program, various key words are used to communicate with the parser. Facilities have been provided to introduce look ahead for

avoiding backup during parsing and to control lexical scanning.

In English-like programming languages [BARNETT 69] certain words are used which are essential for the naturalness of the language but have no significance for machine translation. These auxiliary words can usually be classified as obligatory or optional. For example,

3.6.4 DELETE THE 3-RD CHARACTER OF THE STRING.

3.6.5 DELETE THE 8-TH CHARACTER OF A.

3.6.6 DELETE A.

These examples are based on SNAP, described by Barnett.

It is obvious in (4) that "CHARACTER OF" and "THE" preceding the ordinal adjective are obligatory while "THE" preceding "STRING" is optional. One interesting property of such auxiliary words is that they can almost always be associated with the word to follow but not necessarily with their preceding word. Association of "THE" with "DELETE" will make (6) syntactically

incorrect while it can be safely associated with the ordinal adjective in (4) and (5). During parsing each auxiliary word τ is associated with its succeeding word and a single bit is used to record whether it is obligatory or optional. For instance the string $\beta_5\tau_4\tau_3\beta_2\tau_1\beta_0$ will be treated as $\beta_5\beta_{2\tau_4\tau_3}\beta_{0\tau_1}$ or $\beta_2'\,\beta_1'\,\beta_0'$ where $\beta_2' = \beta_5$ $\beta_1' = \beta_{2\tau_4\tau_3}$ and $\beta_0 = \beta_{0\tau_1}$

* In the system obligatory auxiliary words are enclosed in double quotes and optional auxiliary words are enclosed in single quotes.

3.7 EFFICIENCY CONSIDERATION.

Having outlined the system in the previous section, we are now in a position to discuss various techniques developed to increase its efficiency.

3.7.1 In $[(\alpha_0 = \delta) \rightarrow \delta, (\alpha_0 = \delta\,\xi) \rightarrow \delta\,\xi]$ if $|\xi| >_0$ then $k \geq 1$

The value of k is reduced to one if (1) can be handled as

3.7.2 $[(\alpha_1=\delta) \rightarrow \delta, (\alpha_0=\xi) \rightarrow \xi, (\alpha_0=\Lambda) \rightarrow \Lambda]$

This can be achieved by manipulating the grammar using the system-defined nonterminal "EMPTY" or one of the symbols "ㄱ" and "ㄱN" or simply re-writing the grammar.

* The syntactic entity on the right of "ㄱ" may occur zero or one times, that following "ㄱN" may have n occurrances where $n \geq o$. To illustrate our point we consider part of the grammar shown in table 3.1 (see also example 3.1 above).

3.7.3 $[(\alpha_2=\beta) \rightarrow \beta, (\alpha_1=\delta) \rightarrow \delta, (\alpha_1=\delta\xi) \rightarrow \delta\xi, (\alpha_0=\xi) \rightarrow \xi]$

and clearly (3) is a special case of (1) with $k > 1$. If in this simple case (3) is treated as follows k is reduced to 1.

3.7.4 $[(\alpha_3=\beta) \rightarrow \beta, (\alpha_2=\delta) \rightarrow \delta, (\alpha_1=\xi) \rightarrow \xi,$
$(\alpha_0=\xi) \rightarrow \xi, (\alpha_0=\Lambda) \rightarrow \Lambda]$

The whole grammar of table (3.1) may be similarly stated but the result is considerably more complex.

```
<S> = A ¬ B ¬ C ¬ D ¬ E ¬ F
<S>1,1 = <S>1,2
<S>1,2 = <S>1,3
<S>1,3 = <S>1,4
<S>1,4 = <S>1,5
<S>1,5 = <S>1,6
%SOURCE
A
A B
A B C
A B C D
A B C D E
A B C D E F
A C
A C D
A C E
A C F
C F
B D F
D A F
   £
*** ERROR FOUND
E D B
   £
*** ERROR FOUND
F A B C
   £
*** ERROR FOUND
```

TABLE 3.2

The notation < P > n, m is introduced to refer to the mth symbol of the nth alternative of the (unique) production whose left hand side is < P> , provided that this exists. The MBNF is extended to allow two symbols of a grammar to be linked by a production of the form

$$<P>\ n,m = <\acute{P}>\ \acute{n},\ \acute{m}$$

No new structure is created for this production but a pointer is created from < P> n, m to $<\acute{P}>\ \acute{n}$, $\acute{m}$. After recognising < P> n-1, m-1 if < P> n, m cannot be recognised, the processing continues with $<\acute{P}>\ \acute{n}$, $\acute{m}$. This rather simple idea is very helpful in reducing the value of k and the size of the grammar.

Consider a language with the vocabulary

3.7.5 a b c d e f

and sentences which are strings of arbitrary length but maintain the order specified in (5). Conventionally it will have a large context free grammar and consequently will require a large syntax graph. i.e.

```
<S> = a b c d e f | a c d e f | a d e f |
      a e f | a f | b c d e f | b d e f |
      b e f | b f | c d e f |   c e f |
      c f |   d e f   d f   |   e f | f
```

EXAMPLE 3.2

However, using the linkage scheme described above it can be written as a single basic production requiring an internal structure and several linking productions for setting pointers.

This is illustrated in table 3.2.

3.8 LEFT RECURSIVE GRAMMARS:-

Rosenkrantz and Lewis [ROSENKRANTZ 69] have proved that an LL(k) grammar can have no left recursive nonterminals. We do not dispute the validity of their proof for the LL(k) grammars they have defined but in the light of the definition of our extension it is not relevant.

In their push down machine, a left recursive grammar is of the following form

3.8.1 $[(\alpha_2=\Lambda) \rightarrow n=0, (\alpha_1=\beta') \rightarrow\rightarrow n=n+1, (\alpha_0=\beta) \rightarrow \beta(\beta')^n]$

They rightly argue that $k = |\hat{\beta}| * n$.

Since n is unknown k is also unknown. In our algorithm the left recursion has the form

3.8.2 $[(\alpha_1=\beta') \rightarrow \beta', (\alpha_0=\beta) \rightarrow\rightarrow \beta]$

Since for a nonnull α there is no choice at any level, a member of this class may be an ELL(1) grammar. However, if α_0 could have more than one different acceptable value the grammar would still be ELL(k) but k may be greater than 1. Informally speaking, for a left recursive grammar

<x> = <x> a a a | a b

EXAMPLE 3.3

in a conventional top-down parser, we start with < x > and replace it by < x > a a a. Then we replace <x> again and get <x> a a a a a a. The process continues and the loop never terminates. Lewis and Stearns' proof that the left recursive nonterminals can not be LL(k) follows from the above discussion. Since the language of the above grammar is of the form a b $(a^3)^n$ or a b

$(aaa)^n$, we must replace < x> by < x> aaa, the correct number of times and then start recognising the whole string. For this purpose, before replacing < x> at each stage, it is checked, by looking further ahead, whether the next three symbols are part of $(aaa)^n$ and another replacement is necessary or they are not, and loop must be terminated. In other words all the symbols in a b $(aaa)^n$ must be looked ahead. Since n is unknown, clearly k is also unknown.

In our algorithm, < x > will be replaced by < x > a a a only once. After that the system detects left recursion and recognises a b before trying a a a repeatedly. It will now be seen as in table 3.3 that our algorithm is capable of handling left recursive grammars. The table 3.3 shows four different blocks of information.

1. At the top is a left recursive grammar in which the nonterminal "P" is undefined.

2. "%SOURCE" is an indication to the system that no more production of the grammar is to follow. On meeting this command the system displays a warning message that "P"

```
<R> = <R> B | K
<L> = <L> T | T
<Z> = <Y> A | E F
<X> = <Z> A <L> | B C <R>
<Y> = <X> A A A | <P> C D
<S> = <Y>
%SOURCE
*** UNDEFINED NONTERMINALS
P
%SNOSOL
 Z = LEN(*FINPOS) SPAN(' ') 'C' SPAN(' ') 'D'
 DEFINE('P()','Z1') :(Z3)
Z1 CARD Z :F(Z2)
 CARD LEN(INPOS) 'B' :F(RETURN)
 MATCHED = 1 :(RETURN)
Z2 OBSTACLE = 1 :(RETURN)
Z3 &ANCHOR = 1
%FINISH
B C D
B C D
   £
*** THIS SYMBOL = C OR ITS EQUIVALENT
B C D A A T T A A A
B C D A A T T N
                £
*** ERROR FOUND
B C K B B B A A A
E F A T T T A A A
E F A T T T A C A
                £
*** THIS SYMBOL = A OR ITS EQUIVALENT
B C K A A A
B C K A A P
             £
*** THIS SYMBOL = A OR ITS EQUIVALENT
```

TABLE 3·3

has not been defined.

3. Inside the bracket "%SNOBOL" and "%FINISH" is a SNOBOL4 program. A user defined SNOBOL4 function is defined which performs two tasks:

   a) It performs lexical scanning to recognise B, since <P> = B.

   b) It introduces look ahead to decide whether <P> C D or B C <R> is to be followed.

4. The last part shows syntactically analysed source language statements.

The following grammar is a member of (3·8·2) but is not LL(k) since it is ambiguous

3.8.3 $[(\alpha_2=\beta'') \rightarrow \beta'', (\alpha_1=\beta') \rightarrow\rightarrow \beta', (\alpha_0=\beta) \rightarrow\rightarrow \beta]$

where $(\beta')^n = (\beta)^m$, n and m are positive integers.

Consider

< x > = < x > a a a | c

< S > = < S > a a | < x >

$\beta = a\ a$

$\beta' = a\ a\ a$

$\beta'' = c$

EXAMPLE 3.4

There are bound to be two positive integers n and m, such that:

$$(aa)^n = (aaa)^m$$

e.g. n = 3 and m = 2

satisfy this condition. They generate the following two parse trees

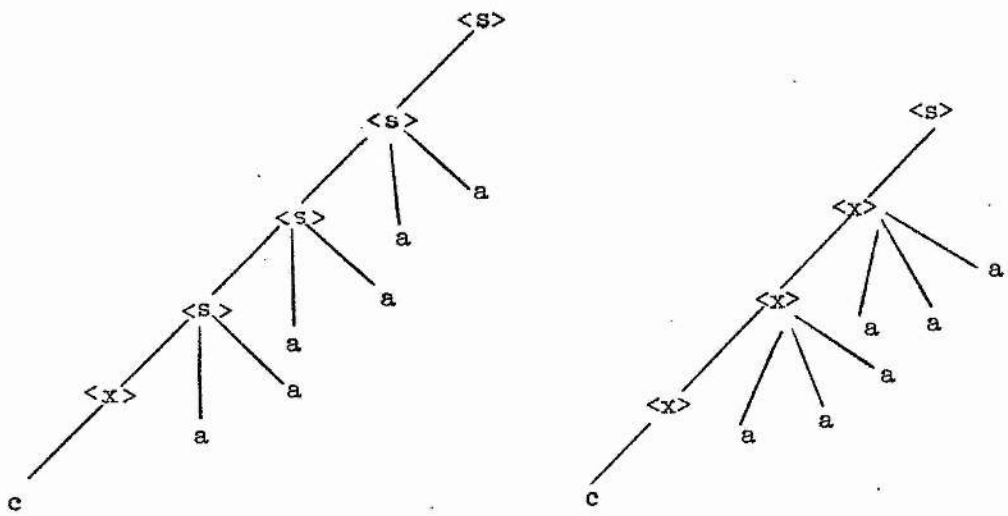


FIG. 3.1

This is also true with some members of the following set:

$[(\alpha_3=\beta^{'''}) \rightarrow \beta^{'''}, (\alpha_2=\beta^{''}) \rightarrow\rightarrow \beta^{''}, (\alpha_1=\beta^{'}) \rightarrow \beta^{'}, (\alpha_0=\beta) \rightarrow\rightarrow \beta\ ]$

For example if

$a^{r} = \beta$ $\qquad$ $a^{r''} = \beta^{''}$

$a^{r'} = \beta^{'}$ $\qquad$ $a^{r'''} = \beta^{'''}$

r, r', r", and r"' are positive integers.

then $|\beta^{'''}| + |\beta^{''}| * m + |\beta^{'}| + |\beta| * n =$

$|\beta^{'''}| + |\beta^{''}| * n + |\beta^{'}| + |\beta| * m$

for some n and m

In the grammar

$<x> = <x>\ a\ a\ a\ a \mid a\ a$

$<s> = <s>\ a\ a\ a \mid a\ <x>$

EXAMPLE 3.5

$\beta^{'''} = aaa$ $\qquad$ $\beta^{'} = a$

$\beta^{''} = a\ a\ a\ a$ $\qquad$ $\beta = a\ a$

and the condition

$a\ a + (aaaa) * n + a + (aaa) * m$

$= a\ a + (aaaa) * m + a + (aaa) * n$

is satisfied.

Since it can be written as

(a a a a) * m + (a a a) * n =

(a a a a) * n + (a a a) * m

it is ambiguous.

However, it is worth noting that these grammars are not ELL(k) because they are ambiguous and not because they are left recursive.

The following is an unambiguous left recursive grammar but is not ELL(k)

3.8.5 $[(\alpha_1=\beta'') \rightarrow \beta'', (\alpha_1=\beta') \rightarrow \beta', (\alpha_0=\beta) \rightarrow\rightarrow \beta\ ]$

where $|\beta'| > |\beta''|$

$\beta' : |\beta''| = \beta''$

$|\beta| > |\beta'|$

$\beta : |\beta'| = \beta'$

$(\beta')^n = (\beta)^m$, n and m are positive integers.

But it can easily be written as

3.8.6 $[(\alpha_1=\beta'') \rightarrow \beta'', (\alpha_0=\beta') \rightarrow\rightarrow \beta', (\alpha_0=\beta''') \rightarrow \beta'''\ ]$

$\beta''' = \beta' : (|\beta'| - |\beta''|)$

For example the grammar

< x > = a a a | a a | a

is left recursive and unambiguous, but it is not ELL(k). However if it is written as

< x > = < x > a ⌐ a | a

it is ELL(1).

CHAPTER IV

IMPLEMENTATION OF THE GENERALISED LL(k) PARSER

4.1 INTRODUCTION

The generalised LL(k) parser uses a syntax graph constructed from the grammar of the language being parsed. It applies a predictive algorithm to traverse through different nodes of the syntax graph, in order to recognise the source language statement. In this chapter, we will first describe the layout of the syntax graph and then discuss different aspects of the parsing algorithm.

4.11 SYNTAX GRAPH:-

The syntax graph has a start node and an arbitrary number of nodes accessible from it. Each node of the syntax graph represents a member of the vocabulary of the grammar being parsed and is linked with other nodes by one or more pointers. Each node consists of six fields as shown in 4.1. Each field either has an entry or has null string as its value.

DEF	QUAL	ALT	SUCC	MOD	AUX

format of a node

FIG. 4.1

Definition field DEF :-

This field either holds $\underline{Y} \in V$, if it is a terminal, or is a pointer to the node representing it if it is a non-terminal.

Qualification field (QUAL):-

This contains the following information represented by a unique code

a) Whether DEF represents
   (i) A terminal
   (ii) A nonterminal enclosed within corner brackets.
   (iii) A nonterminal enclosed within ampersands.

b) Whether "¬ " or "¬N " or neither of the two exist immediately to the right of the current symbol.

c) Whether the symbol represented by the node pointed at by AUX of the current node is obligatory or optional.

Alternative field (ALT):-

If X... is an alternative of Y... then ALT of the Y_{NODE} points at the X_{NODE}.

Successor field (SUCC):-

If the R.H.S. of a production is of the form ...Y X ... then SUCC of the Y_{NODE} points at the X_{NODE}.

Modification field:-

This field keeps the pointer (if any) to the node which must be tried in case the current path of the parse is to be modified.

Auxiliary field (AUX):-

If the right hand side of a production is of the form ...Ø Y where Ø is a non-empty string of auxiliary words, the AUX of $\underline{Y}_{NODE}$ points at the node representing the left most symbol of Ø. For all practical purposes the said production is considered to be of the form ...$\underline{Y}$... while Ø has an independent representation.

To construct the syntax graph, the parser uses a symbol table, each entry of which consists of two fields, the definition field and the pointer-field. The definition field accommodates a nonterminal X on the left hand side of a production P_i while the corresponding pointer-field keeps a pointer to the $\underline{Y}_{NODE}$ where X FIRST $\underline{Y}$.

A production is scanned from left to right to find X. If X is subscripted then $\underline{Y}$ must also be subscripted. Subscripted X and Y refer to $\underline{X}'$ and $\underline{Y}'$ where $X \Rightarrow \cdots X' \cdots$ and $Y \Rightarrow \cdots Y' \cdots$ while X', Y' are determined by the respective subscripts of X and Y. As a result of this production the MOD of X'_{NODE} is set to Y'_{NODE}. If X is not subscripted, it is entered in the symbol table, provided it has no entry already. A node called $\underline{Y}_{NODE}$ is created for $\underline{Y}$ and the pointer field of the most recent entry in the symbol table is set to the $\underline{Y}_{NODE}$. The production is then scanned further and the part of the syntax graph required for it created as follows.

If the next symbol is

a) t_i, the first of the consecutive auxiliary symbols $t_1\ t_2\ t_3 \ldots . t_n$ where $n \geq 1$ then t_{iNODE} is created. and AUXPOINT is turned on. Nodes are also created for $t_2\ t_3 \ldots t_n$. All the nodes of the consecutive auxiliary words are connected by their SUCC fields such that

$$SUCC(t_i) = t_{i+1} \qquad n > i > o$$

b) $\underline{Z} \in \underline{V}$ then a new node is created to accommodate it and SUCC of the previous node is set to the $\underline{Z}_{NODE}$. If the AUXPOINT is on, AUX of the $\underline{Z}_{NODE}$ is set to the node pointed at by it and the AUXPOINT is turned off.

c) "|", the symbol preceding it is considered as the last symbol of $Y^{[n]}$..., where $\underline{Y}^{[n]}$ is the nth alternative of $\underline{Y}^{[1]}$

The production is scanned further to find the next symbol which is expected to be $\underline{Y}^{[n+1]}$. A new node is then created for $\underline{Y}^{[n+1]}$ and ALT of $Y^{[n]}_{NODE}$ is set to the $\underline{Y}^{[n+1]}_{NODE}$.

d) " ↑ ". The ID of the most recently created node is modified to reflect the occurrence.

e) "↑N ", the ID of the previous node is modified as in (d) and further scanning continues to find $\underline{Z}$ such that $\underline{Z} \in \underline{V}$. Step (a) is performed and a pointer GRAPHPOINT is set to this node.

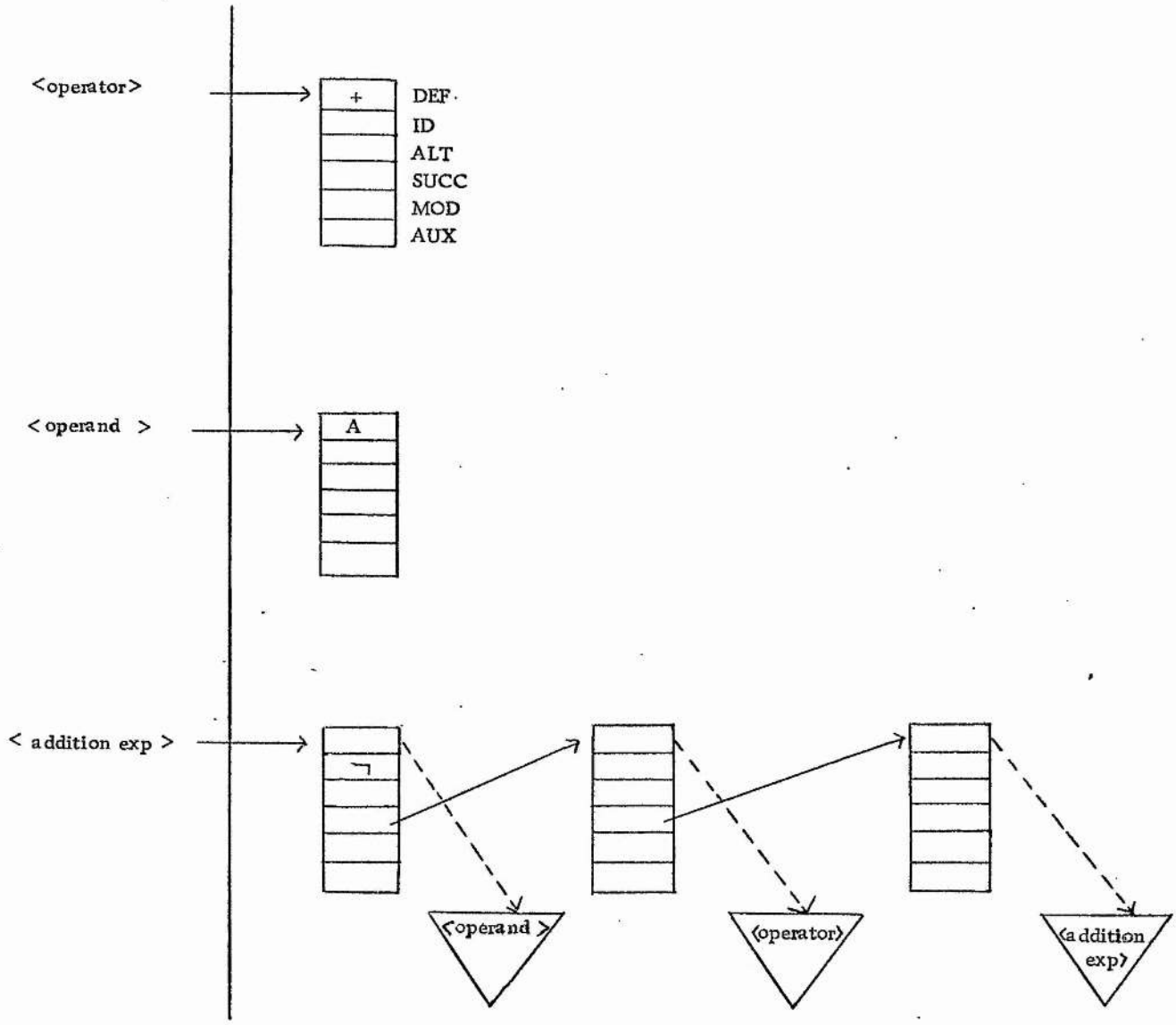
f) The last symbol of the production. It must be a member of V. A node is created for it and linked with the other nodes as usual. If the GRAPHPOINT is on, the SUCC of the current node is set to the node pointed at by GRAPHPOINT and GRAPHPOINT is turned off.

When reference is made to a nonterminal T which does not have an entry in the symbol table, the processing of the current production is suspended after saving any necessary information. An imaginary production <T>= Λ is then processed. It's symbol table entry is marked to show that it is an ad hoc one and then the processing of the actual production is resumed from the point at which it was suspended. At a later stage when the production<T>=<Y>... is processed, no new entry is made for T in the symbol table. It's previous entry is unmarked and the structure of the corresponding production is modified so as to accommodate the current production.

For this reason, before making a new entry, the symbol table is always searched for that entry.

The presence of any such unmarked entry is erroneous and the current production is ignored with a warning message. In all other circumstances the normal process continues.

At the end of the syntax specification, marked entries in the symbol table are displayed with an appropriate message. It is assumed that the user will have defined these nonterminals at some stage. The DEF of the node representing the undefined symbols are filled with the symbols themselves and corresponding SNOBOL4 code is generated and linked at the appropriate place of the processor. This enables the processor to call the corresponding user defined SNOBOL4 function when an undefined nonterminal is processed during Syntax analysis.



The above diagram shows the syntax graph of the following grammar.

$\langle operator \rangle$ = +

$\langle$ operand $\rangle$ = A

$\langle addition\ exp \rangle$ = $\langle$ operand $\rangle$ $\langle operator \rangle$ $\langle addition\ ex$

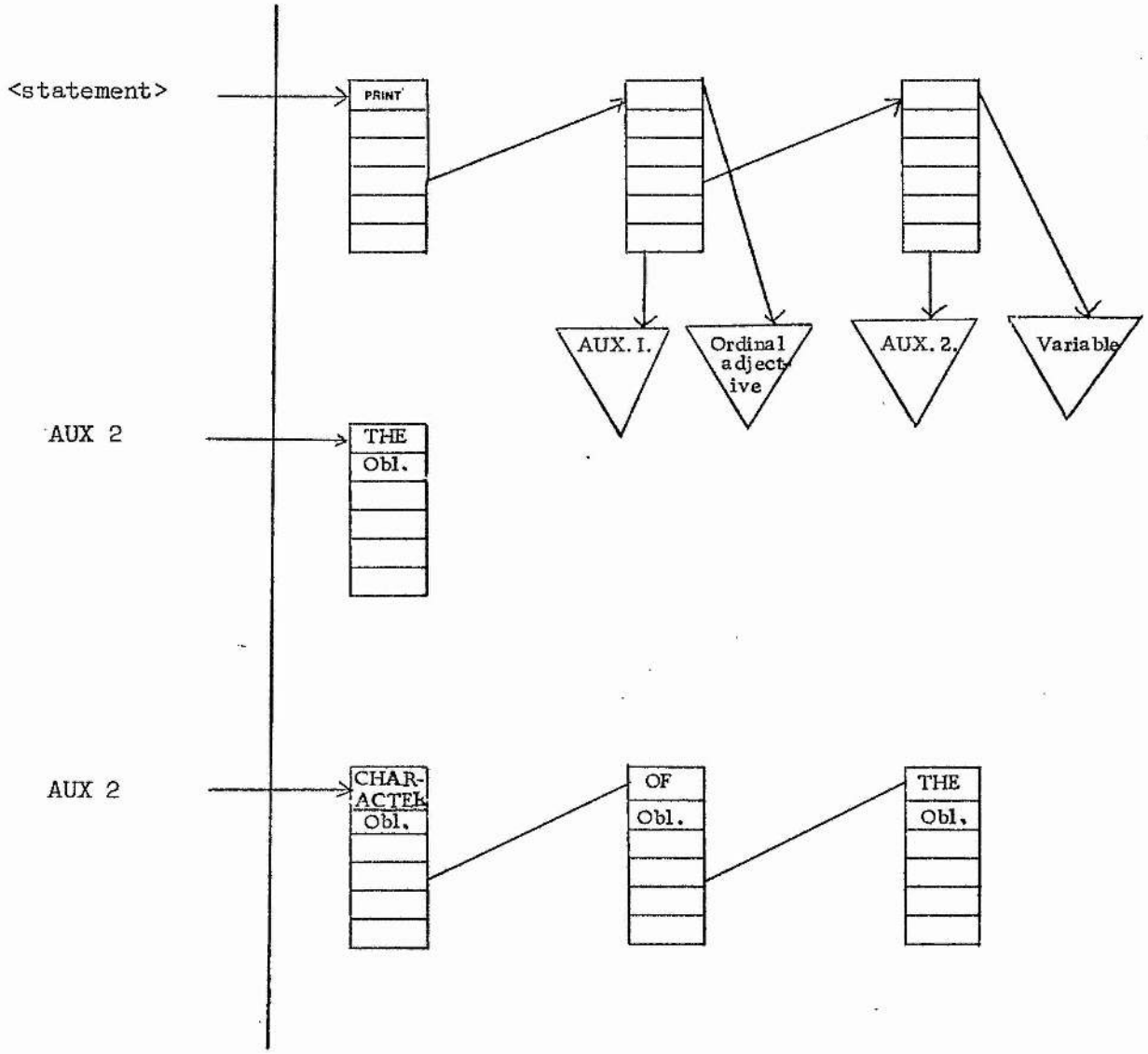
EXAMPLE 4.1

In the above diagram, symbols on the L.H.S. of the vertical line represent nonterminals to be entered in the symbol table and oblong boxes are nodes of the syntax graph. An arrow represents a pointer while a triangle is a pointer to a node which can be accessed with the help of symbol table entry specified in the triangle itself.

example:- 4.2

```
⟨statement⟩  =  PRINT "THE" ⟨ordinel adjective⟩
                            "CHARACTER OF"
                            'THE' ⟨variable⟩
```

The syntax graph of the above statement is given below.



Representation of auxiliary words.

FIG. (4.3)

4.12 PARSING ALGORITHM:-

We now describe the basic parsing algorithm. It's role is fundamental and rest of the predictive algorithm may be considered as its extension. A priority list of paths to be traversed in the syntax graph has been defined and is adhered to strictly. Necessary information is stored on the syntax - analysis stack (SAS). Each element of SAS has three fields: node, path and position. For the convenience of description it is considered as if this one stack is the "concatenation" of three stacks, each having elements consisting of single fields. When a source statement is to be parsed, the parser is initialised so that

(i) The left most symbol of the source statement is the current symbol.

(ii) The node representing the start symbol of the grammar is the current node.

The parser then goes through the following steps.

a) If the current node is a nonterminal, stack it in the node stack, stack the current position in the position stack and stack an element in the path stack marking it as daughter. Make the

daughter of the current node as the "new" current node and repeat this step.

b) If the terminal is a function, obey it, otherwise match it with the current symbol.

c) If the match is a success, check to see if there is another symbol in the source statement immediately to the right of the current symbol. If "yes" pick the new symbol and go to (i). If no "new" symbol can be picked, run the algorithm until successful recognition of the source statement is confirmed by exhausting the stack or an error condition is sensed. In either case the algorithm is terminated.

d) In (c) if the match is not successful, check to see whether the current node is a successor of some other node. If "yes" try the MOD field otherwise give an error message and terminate the algorithm.

e) If the current node is $EMPTY_{NODE}$, the position fields of the elements of SAS are marked -ive and the control is transferred to (i).

f) If MOD field is null and the current node is mandatory (not optional due to ⌉ or ⌉N), the current state is an error state. If the current node is not mandatory, ignore it. If the MOD field is not null, stack the current node, entering the position and the path fields. If it is already at top of SAS, mark the path top as MOD. Make the modification node, the current node and go to (a).

g) If the current node is the left most symbol on the R.H.S. of a production, check to see if it has an alternative. If yes make the alternative, the new current node and go to(a),otherwise to to (h).

h) (i) If the SAS has been exhausted, check how the algorithm is to be terminated, with or without an error.

(ii) If the current position does not match the POSITION-top, mark the PATH-top as successor and go to (i).

(iii) If the PATH-top is marked as MOD go to (hV) and if it is marked as ALT or DAU go to (hVI)

(iv) If the MOD field is null, mark the PATH-top as MOD, make the modification node, the new current node and go to (a).

(v) Check to see whether the right hand side of the current production can end at the current point. If "yes" delete the SAS-top and repeat (h). If "no" print an error message and terminate the algorithm.

(vi) If the current production has an alternative, make it the new current node and go to (a), otherwise delete the SAS-top and repeat (h).

i) If the current node has a successor, stack the current node, make its successor the new current node and go to (a),otherwise go to (h).

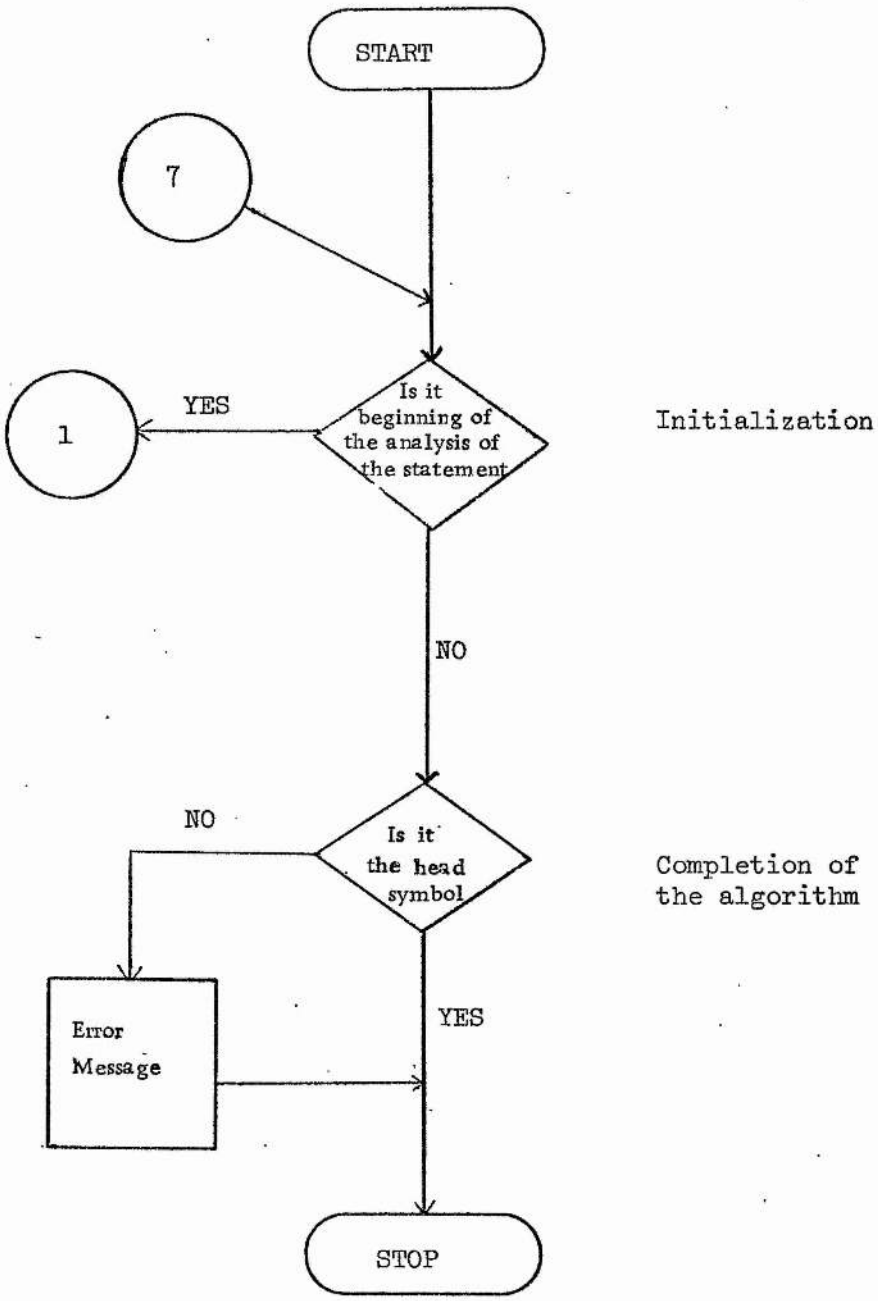
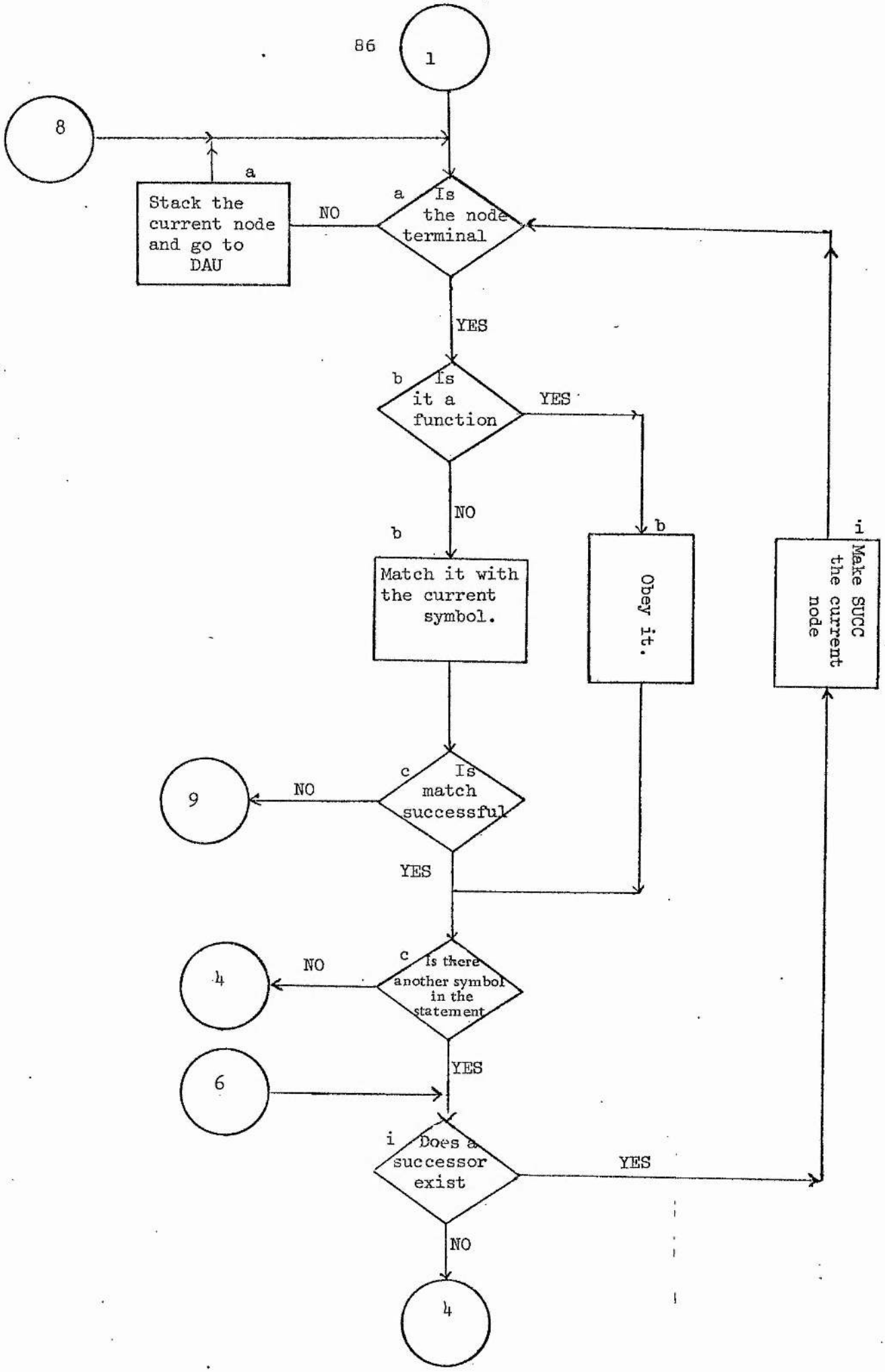
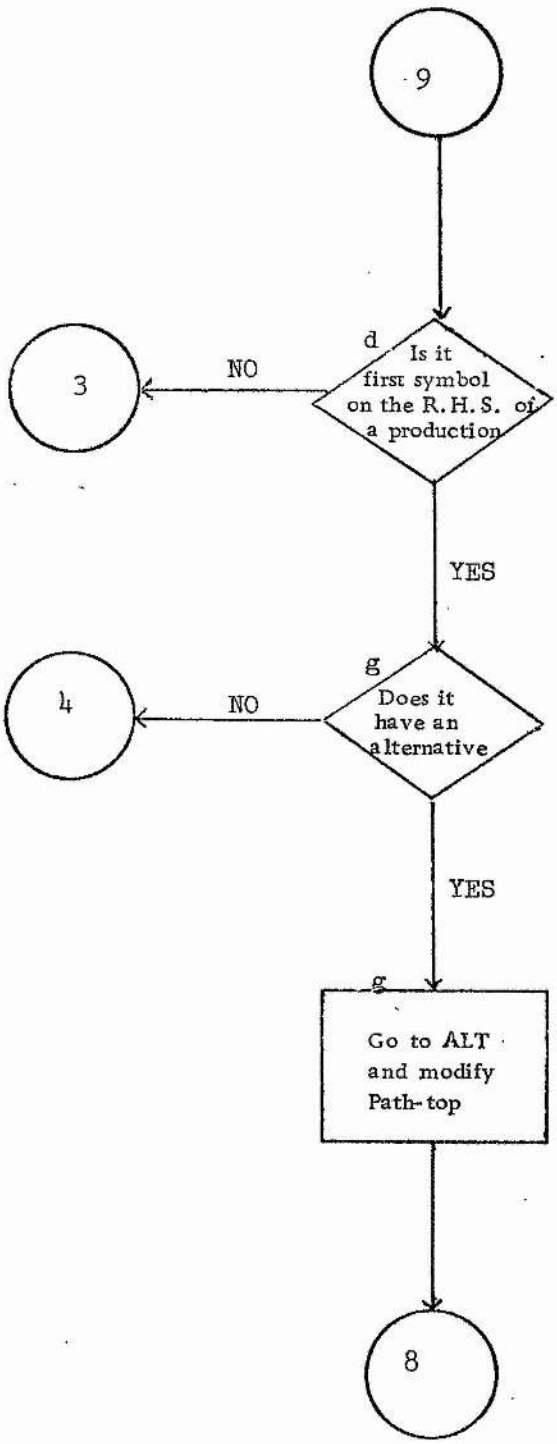
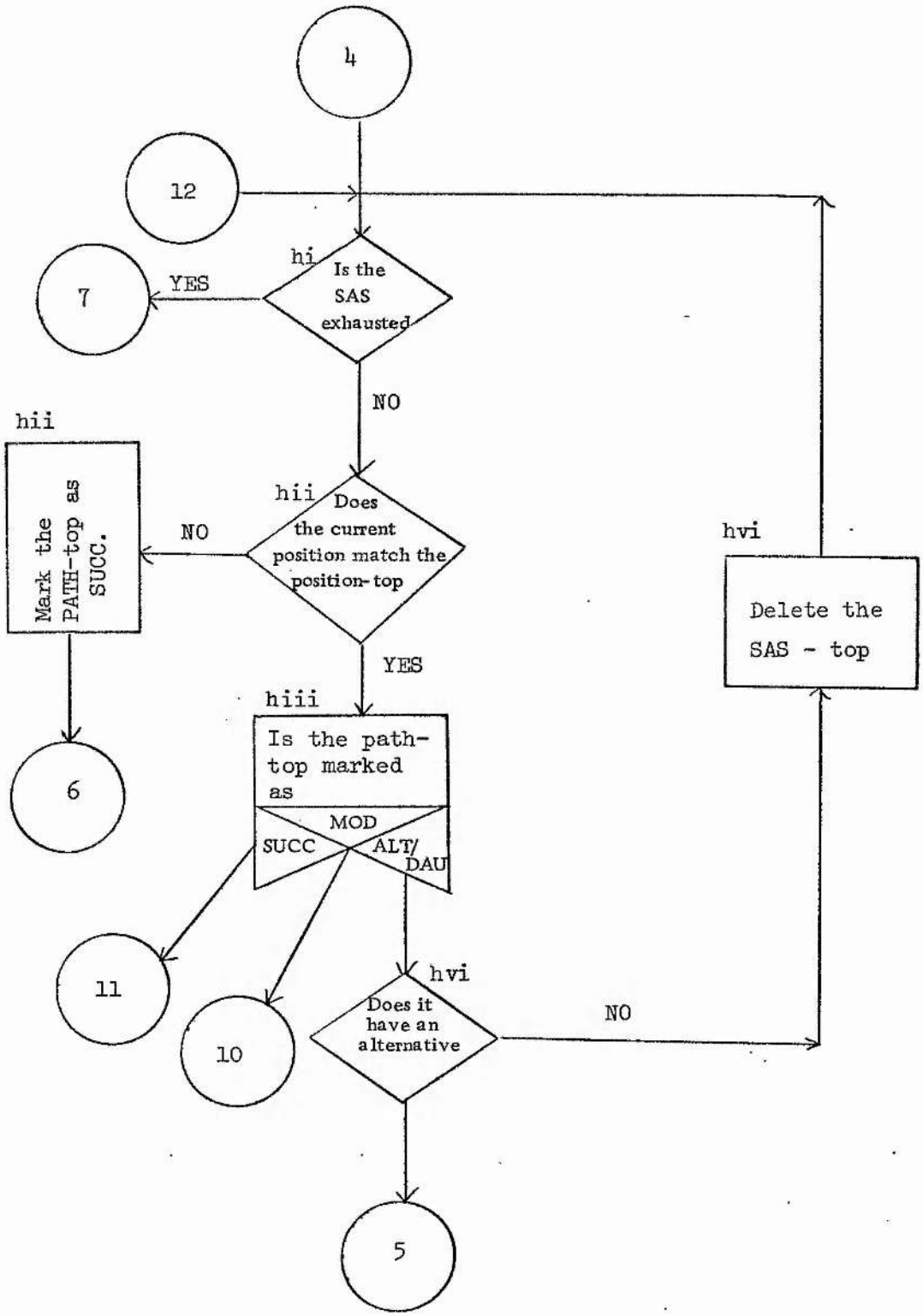
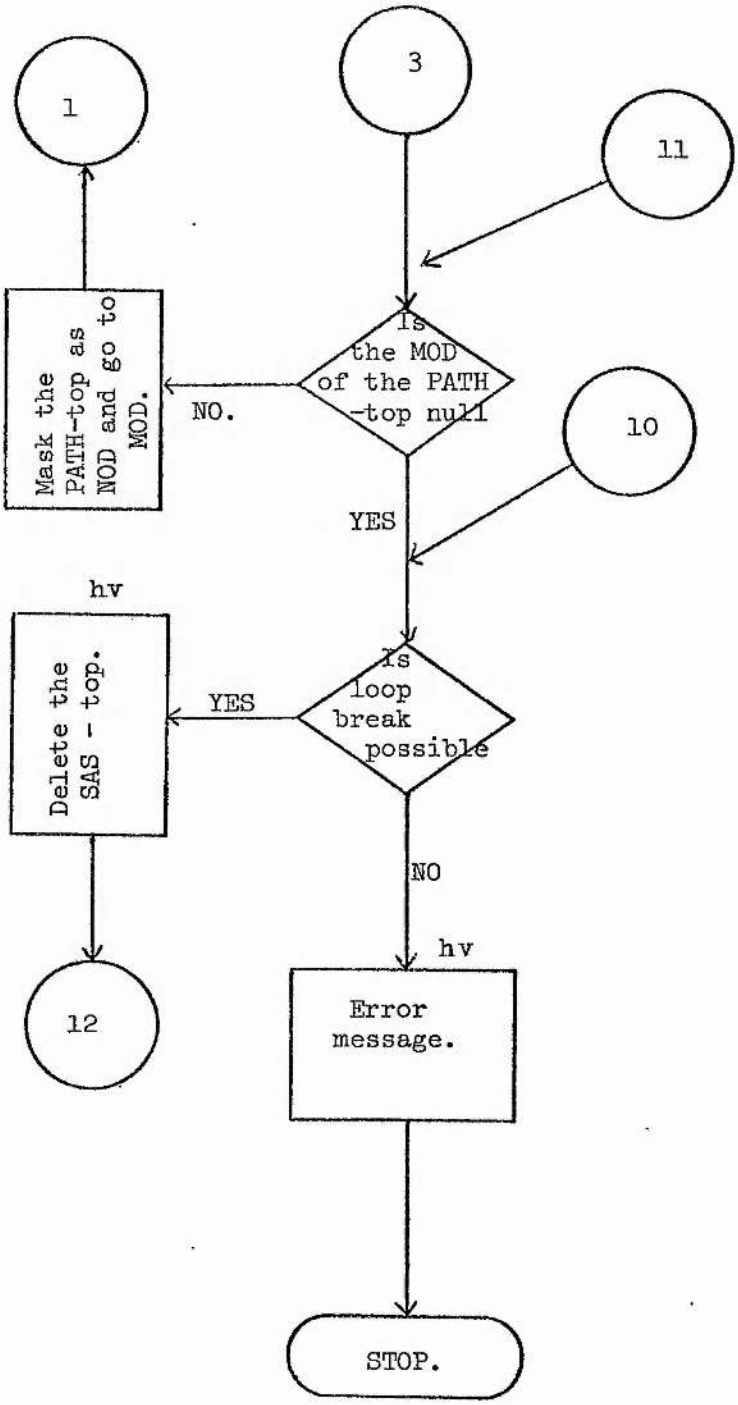

START
7
1
YES
Is it beginning of the analysis of the statement
Initialization
NO
NO
Is it the head symbol
Completion of the algorithm
Error Message
YES
STOP

FIG 4.4


1
8
a
Stack the current node and go to DAU
NO
a Is the node terminal
YES
b Is it a function
YES
NO
b
Match it with the current symbol.
b
Obey it.
i
Make SUCC the current node
9
NO
c Is match successful
YES
4
NO
c Is there another symbol in the statement
6
YES
i Does a successor exist
YES
NO
4


9
d
Is it first symbol on the R.H.S. of a production
NO
3
YES
g
Does it have an alternative
NO
4
YES
g
Go to ALT and modify Path-top
8


4
12
hi
Is the SAS exhausted
7
YES
NO
hii
Mark the PATH-top as SUCC.
hii
Does the current position match the position-top
NO
YES
hvi
Delete the SAS - top
hiii
Is the path-top marked as
MOD
SUCC
ALT/DAU
6
11
10
hvi
Does it have an alternative
NO
5


1
3
11
Mask the PATH-top as NOD and go to MOD.
Is the MOD of the PATH -top null
NO.
10
YES
hv
Delete the SAS - top.
Is loop break possible
YES
NO
hv
12
Error message.
STOP.

EXAMPLE 4.3

With the grammar given in example 4.1, the expression "A + A" will be recognised in the following steps.

1) Make "A" as the current symbol, stack ⟨addition exp⟩ $_{NODE}$ in the node field of SAS and go to its daughter node.

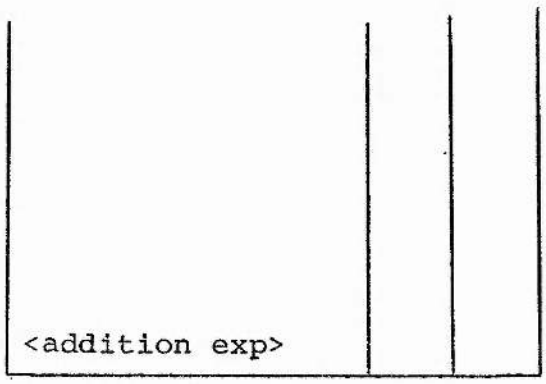


FIG. 4.5

2) Stack ⟨addition exp⟩ 1,1 $_{NODE}$ in the node field of SAS, the current position in the position field and DAU in the path field. (Step a)

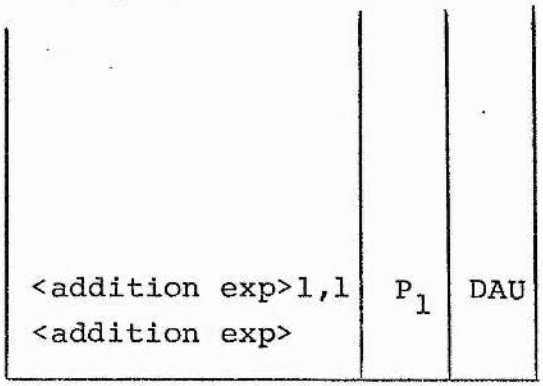


FIG. 4.6

3) Recognise "A", pick "+" as the current node (Step c).

4) There is no successor of the current node (Step i).

5) Mark the path-top as SUCC (Step hii) and go to the successor of the node-top (Step i).

<addition exp>1,1 <addition exp>	P_1	SUCC

FIG. 4.7

6) Stack the current node (Step a) and go to its DAU.

<addition exp>1,2	P_2	DAU
<addition exp>1,1	P_1	SUCC
<addition exp>		

FIG. 4.8

7) Recognise "+" and pick "A" as the new current symbol (Step c).

8) Go through (4), (5) and (6).

<addition exp>1,3	P_3	DAU
<addition exp>1,2	P_2	SUCC
<addition exp>1,1	P_1	SUCC
<addition exp>		

FIG. 4.9

The SAS is now as given in the diagram 4.9

9) Since the current node represents a nonterminal, stack it and go to its daughter (Step a)

<addition exp>1,1	P_3	DAU
<addition exp>1,3	P_3	DAU
<addition exp>1,2	P_2	SUCC
<addition exp>1,1	P_1	SUCC
<addition exp>		

FIG. 4.10

10) Recognise "A". This is the end of the statement since there is no symbol on its right. Neither the current node nor any one of the nodes on the SAS have a mandatory successor node. Hence this is a legal statement. (Step c).

4.2 LEFT RECURSION:-

4.21 GENERAL CONSIDERATION:-

We consider a grammar G' (V_N', V_T', P', S'), constructed from a grammar G (V_N, V_T, P, S) in satisfying the following restrictions.

$$V_T' \subset V_T$$
$$V_N' \subset V_N$$
$$P' \subset P$$

and P' containing a mutually left recursive subset of productions P_1', P_2', P_3' ----------P_n'.

The sentences of $L(G')$ are thus sentences of L(G), or parts of them.

EXAMPLE 4.4

$$\left.\begin{array}{l} \left.\begin{array}{l} <S> = \quad <X> <S'> \quad \ldots \\ <S'> = \quad <X> \; a \mid c \end{array}\right] \; G' \\ <X> = \quad <S'> \; b \mid d \end{array}\right] \; G$$

It is not difficult to see that

$L(G) = \{c[ba]_o^m\} \cup \{d\,a[b\,a]_o^n\}$ m, n $\geq$ o

If P_i' is of the form

$$< X > = < Y > \; < Z > | \ldots$$

We define γ_i to be such that $Z \Rightarrow^* \gamma_i$

and $\gamma = \gamma_1 \gamma_2 \gamma_3 \ldots\ldots \gamma_n$ $n \geq o$ and

γ is a part of a sentence of $L(G')$.

We shall also consider the "tail" ξ of γ as

$\xi = \gamma_{m+1} \gamma_{m+2} \gamma_{m+3} \ldots\ldots \gamma_{m+n}$ $o < m < n$

For production P_i at which the left recurrsion may terminate, the terminating alternative will be referred to as η_i. The notation can be generalised if there are several such alternatives. β is simply defined as $\eta\xi$.

Informally stating, in the example 4.2 L(G) has three different types of substring:

a) which can have n occurrences (such as "ba" in $\{d\,a\,(b\,a)^n\}$

(γ in the notation.)

b) which can have only one occurrence but are parts of substrings in (a) (such as the first "a" in $da[ba]_o^n$).

(ξ in the formal notation)

c) which can occur once only but are not parts of substrings in (a) (such as d in da $(ba)^n$)

(η in the formal notation)

In the example 4.4 , for G'

$\beta_1 = \eta_1\xi$ $\quad\quad$ $\beta_2 = \eta_2\xi_2$

$\eta_1 = c$ $\quad\quad$ $\eta_2 = d$

$\xi_1 = \Lambda$ $\quad\quad$ $\xi_2 = a$

4.22 PRACTICAL CONSIDERATION:-

It is possible that there are several nonterminals $S'_1, S'_2 \ldots\ldots\ldots S'_n$ in G' that are initial symbols of corresponding grammars $G'_i , G'_2 \ldots\ldots\ldots G'_n$ having the same set of productions $P'_1, P'_2 \ldots\ldots\ldots P'_n$. G_i is essentially a set of grammars consisting of one set of productions but in different order in each case. So an algorithm is required which could detect S'_i and its corresponding order $P'_1, P'_2 \ldots\ldots\ldots P'_n$ for G'_i.

For example in example 4.4, X and S' are in both G and G'. Hence we consider S' as S'_1, and X as S'_2.

EXAMPLE 4·5

G'_1 is the same as G' in 4.4 but G_2 will be as follows

$$\langle X \rangle = \langle S' \rangle \quad b \mid d$$
$$\langle S' \rangle = \langle X' \rangle \quad a \mid c$$

By the definition of context free grammar (example 4.4)

$$G'_1 \neq G'_2$$

Also $L(G'_1) \neq L(G'_2)$

It is therefore necessary to find the right order of productions $P'_1, P'_2, P'_3 \ldots\ldots P'_n$ in G'_2 and the corresponding start symbol S'_2

4.3 IMPLEMENTATION OF LEFT RECURSION :-

4.3.1 BASIC PHILOSOPHY:-

Let S_i' be on the left hand side of P_i' which is in G_i', then by the definition of left recursion

$$S_i' \text{ FIRST* } S_i' .$$

With this in mind we explain our algorithm by the following example.

$$\begin{aligned} &< X > = < y > \ a \mid d \\ &< Y > = < Z > \ b \mid c \\ &< Z > = < X > \ c \mid f \end{aligned}$$

EXAMPLE 4.6

From example 4.6 using the parsing algorithm already developed, the SAS will be in the form shown in diagram 4.11 for any legal grammar , and then the three elements will be stacked again and again in the same order. The following extension to the parsing algorithm prevents this repetition.

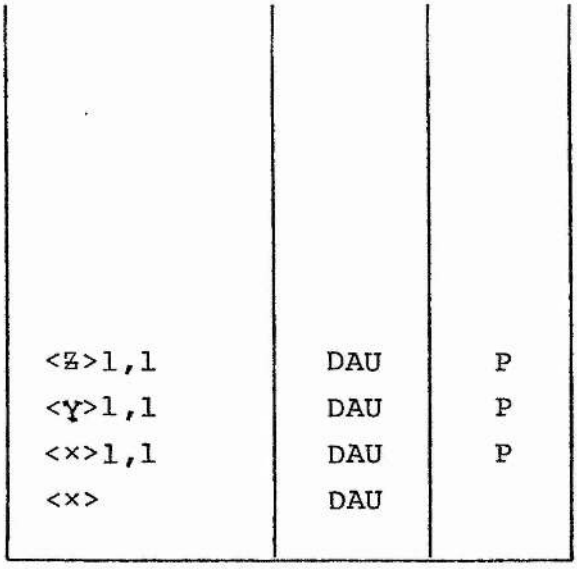


FIG. 4.11

Before making a new entry in SAS, it is processed from top towards the bottom. The current position is matched with elements of the position stack and the current node is matched with the corresponding elements of the node stack. The search is terminated when, either a match is found or the stack is exhausted. If the match was found between the current node and an element of the node stack, mark that element of SAS as S'_i. If the element immediately below is also marked, all the elements of SAS above, including the current element, are marked as left recursive and the element immediately below S'_i is unmarked. After that η_i, ξ_i and γ are determined and $\eta_i \xi_i \gamma^n$ with $n > o$ represents a sentence of $L(G')$. Algorithms which recognise η, ξ and γ are now described.

<x>	L.R.	P
<ξ>	L.R.	P
<γ>	L.R.	P
<x>	DAU	P

Stack with the above mentioned modifications

FIG. 4.12

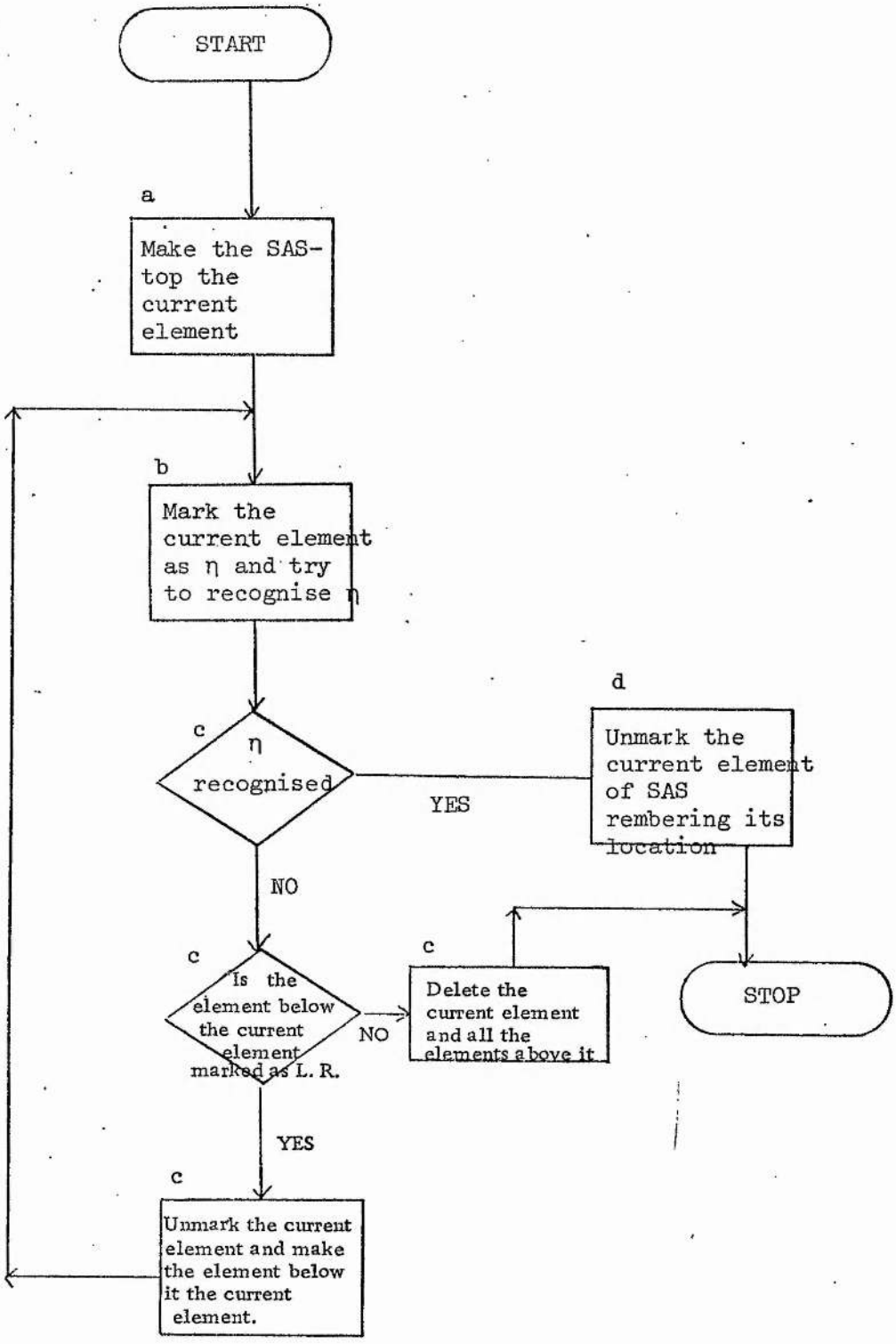
4.32 η - ALGORITHM:-

a) Make the SAS-top the current element.

b) Mark the current elements as η and try to recognise η by looking k symbols ahead.

c) If the recognition is not successful, unmark the current element and check to see whether the element immediately below the current element (if any) is marked as left recursive. If so, unmark the current element and make the element below it, the current element and go to (b), otherwise delete the elements above and including the current element.

d) In (c) if the appropriate η is determined and recognised, that means η_i has been completely evaluated. The current element of SAS is unmarked ξ-algorithm is called in.


START
a
Make the SAS-top the current element
b
Mark the current element as η and try to recognise η
c
η recognised
YES
d
Unmark the current element of SAS rembering its location
NO
c
Is the element below the current element marked as L. R.
NO
c
Delete the current element and all the elements above it
STOP
YES
c
Unmark the current element and make the element below it the current element.

4.33 SOME PROBLEMS WITH η - ALGORITHM:-

If X is a nonterminal or the descendant of a nonterminal in the sentential form representing η_i, there are three possibilities.

a) X FIRST* X is not true.

b) X FIRST* X is true but
   S_i' FIRST* X is not true

c) Both X FIRST* X
   and S_i' FIRST* X are true

In other words

S_i' FIRST* X is true.

In the first two cases, no explanation is necessary, since the parser will work as usual, the only difference being that when an element of SAS is uncovered which is marked as left recursive, it must be determined whether η_i has been evaluated completely. Condition (c) has more serious implications since the parser goes into a loop. This however does not pose any serious problems since it is detected at the time of stacking X and hence the sentential form containing X is not tried. Incidently grammar satisfying this condition is ambiguous.

4.34 ξ - ALGORITHM:-

Since ξ is a part of γ, the same algorithm can be used to evaluate both. One exception is that the element of SAS (if any) immediately above the element marked as η by the η-algorithm is considered as the current element at the start of the ξ-Algorithm. Another difference of course is that ξ is compulsory while γ is optional.

4.35 γ - ALGORITHM:-

Mark the element of SAS reached in the ξ-algorithm as γ and try to evaluate $\gamma_{CURRENT}$. If the evaluation is successful, unmark the current element of SAS and make the element immediately above it, the current element and repeat from start of the γ-algorithm. The process is interrupted when the top of the stack is reached. At this stage, starting from the top of the stack, the lowest of the "consecutive" elements marked as left recursive is determined, starting from this element and going upwards a search is made for an element such that $|\gamma_{CURRENT}| \geq 1$. It is marked γ and an attempt is made to evaluate $\gamma_{CURRENT}$. If no symbol is matched

for $\gamma_{CURRENT}$, the algorithm is terminated, otherwise the normal process is continued.

CHAPTER 5

METASEMANTIC LANGUAGE

5.1 INTRODUCTION:-

In this chapter we shall describe the semantic synthesiser part of the automatic translator. While the syntax of context free languages has been thoroughly formalised, no satisfactory formalisation of language semantics exists. A practical general technique is also difficult to imagine, since the semantics of different programming languages can be so different. The same is true with the machines on which they are to be implemented. However the metasemantic language (MSEAL) has been so designed that a great deal of formalism has been achieved without imposing too many constraints on its power. MSEAL can be considered as a problem oriented computer language. The problem involved is the representation of the meaning of high level programming language statements.

5.2 SURVEY:-

Before we go into the details of MSEAL, a brief survey of different techniques currently

being employed for specifying the semantics of programming languages is in order.

Probably the oldest and definitely informal method of defining semantics is by using a natural language as a metasemantic language. Most programming language manuals have adopted this method. Various objections to such a definitional method arise. The strongest of these is that natural language itself incorporates a huge and unanalysed body of tools which we are still far from being able to handle. This difficulty arises most strongly in connection with the semantic properties of natural language itself. Thus we have no mechanical way of processing natural language definitions and even if given what purports to be a complete definition D of a programming language L, we have no programmable way of verifying the completeness of D, mechanically transforming D into a compiler or interpreter for L, or mechanically determining whether any given compiler for L does realise the object defined in D.

The second semantic definition method commonly encountered and suitable for either informal use is that which may be called the method of devolution and is as follows: Within a language L, to be defined semantically, we determine a sublanguage λ, which is as restricted as possible; then we treat the full language L as an extension of λ. That is, specifying some formal mechanism by which programs written in L can be written in the more restricted language, we reduce the semantic definition problem of L to that for λ. Such reduction may clear away a fair amount of "superficial mess" associated with L but not present in λ. For example if we apply this method to FORTRAN we can eliminate the DO-statement by an explicitly programmed iterative loop.

This method restricts the structure of the languages which may be defined too strongly for use in an automatic translator of the kind we have constructed.

The third type of models we will consider is abstract semantics models. The objects being represented are assumed to have an existence independently of any representation. It is

the purpose of the semantic definition to characterise the "essence" of such independently existing objects in a representation-independent way. This approach leads to attempts to reduce computational notations to methematical notations, since mathematical models are assumed to capture the representation-independent essence of computational phenomena. For example Scotts model [Scott 70] of Computable functions in terms of a class of mathematical lettics is an abstract semantics model.

While at some time in the future it might have some practical importance at present its significance is mainly theoretical.

Input-Output models are another interesting way of investigating programming language semantics. In these models, the functions we wish to compute are characterised in terms of the relation between inputs and outputs which they determine. This approach to the assigning of meaning to programmes was considered by Floyd [FLOYD] and developed by Manna [MANNA 69] , Hoare [HOARE 71] and Manna and Waldinger [MANNA 71].

Although many of the computations that we wish to specify in practice are conveniently specified by a relation between inputs and outputs, there are some computations which cannot be specified in this way. For example programming languages generally have an undecidable halting problem. We can not use input-output semantics to uniformly specify the semantics of an interpreter for a programming language in terms of a relation between inputs and outputs. Moreover input-output semantics regard all programs which realise the same function as equivalent. However the language designer is interested in differences of representation of a function in different programming languages and the language implementer is interested in differences of implementation of a given program in a given programming language. An operational model of semantics is the last model we will discuss and our semantic synthesizer falls in to this category. In this model we are concerned not only with the relations between inputs and outputs, but also with the path by which we get from the input to the output and the information structure generated along this path. A general class of models for the operational specification of programming languages in terms of information structure transformations which

are performed by the language translator can be called information structure model and defined as follows. An information structure model is a triple $M = (I, I^o, F)$ where I is a countable set of information structures (structured states), $I^o \subseteq I$ is a set of initial representations and F is a finitely representable set of unary operations. whose domain and range is a subset of I. A deterministic (sequential) information structure model is one which, for all $I_j \in I$, has at most one element $f \in F$ applicable to I_j. From now on we will only deal with the deterministic information structure model. A computation in a (deterministic) information structure model $M = (I, I^o, F)$ is a sequence I_o, I_1----- of elements of I such that $I_o \in I^o$ and for J = 0, 1, 2 -----, $I_j = f_j(I_j)_1$ for some $f \in F$. If I^o is the syntax of a programming language, then I_2, I_2 ----I_n is a sequences of steps which must be carried out in some part of a translation. $f(I_j)$ is the action which is to be carried out at step I_j. If $f_c(I_j)$ denotes the generation of code then $f_c(I_j) \subseteq \Sigma f(I_j)$ for any I^o. I^o is said to be completely evaluated when for some integer n, an I_n is reached to which no element of $f \in F$ is applicable.

5.3 METASEMANTIC LANGUAGE (MSEAL):-

The basic approach we have adopted is that three different mechanisms are provided for handling each one of I^o, I_1------I_{n-1} and I_n. The input to the semantic synthesizer consists of semantic productions. Each semantic production has three fields ordered from left to right: the environment field, the action field and the code field. The symbol " -> " separates the environment field from the action field and ":" separates the action field from the code field. If M is the set of all the semantic productions, then I^o is the set of all the environment fields in M.

A formal mechanism has been provided to specify I_o where $I_o \in I^o$, I_o comprises of different environment relations. These are combinations of various static objects such as nonterminals, terminals and identifiers and the so called relation operators. Statements requiring different actions have been provided to specify F in terms of I, in the action field. High level data structure oriented commands have been provided to manipulate various data objects. I_n can be either specified in the action field

or more explicitly in the code field. For a semantic action, using the information obtained from the syntax analyser, I_o is selected where $I_o \in I^o$. I_o is then automatically transferred successively into I_1, I_2, ---- I_n. In the rest of this chapter we shall give details of the metasemantic language (MSEAL). For clarity, it has been divided into various sections. Each section contains the syntax and the semantics of a subset of the MSEAL. MSYL described in chapter 3 has been adopted as the metasyntactic language for MSEAL, with the exception that the right most descendant of a nonterminal enclosed in corner brackets can be followed by zero or more blanks. The following are assumed throughout this chapter.

(i) Only strings can be concatenated.

(ii) Arithmetic operations can be performed on integers only.

(iii) A cell of a data object can not have another cell, element or data object as its value.

(iv) In the action field of a semantic production, statements are separated by ampersands.

5.4 DATA OBJECTS, IDENTIFIERS AND SELECTORS:-

⟨Brac op⟩ = (

⟨Brac cl⟩ =)

⟨letter⟩ = A|B|C ----------|Z

⟨digit⟩ = 0|1|2|3 ----------|9

⟨integer⟩ = &integer& &digit&| &digit&

⟨alphanumeric⟩ = ⟨letter⟩ |⟨digit⟩

⟨identifier⟩ = &identifier& &alphanumeric&|&letter&

⟨que limit⟩ = BACK| FRONT

⟨stack limit⟩= TOP | BOTTOM

⟨quote⟩ = "

⟨string⟩ = any string of characters including null string

⟨quoted string⟩ = &quote& &string& &quote&

⟨cell index⟩ = ⟨integer⟩|⟨identifier⟩

⟨index⟩ = ⟨integer⟩|⟨identifier⟩ | ⟨quoted string⟩| ⟨indexed nonterminal⟩

⟨indexed stack id⟩ = ⟨identifier⟩ ⟨brac.op.⟩ ⟨index⟩ ⟨brac.cl.⟩ |

⟨identifier⟩ ⟨brac.op.⟩ ⟨stack limit⟩ ⟨brac.cl.⟩ |

⟨identifier⟩ ⟨brac.op.⟩ ⟨index⟩ ⟨brac.op.⟩⟨cell index⟩ ⟨brac.cl.⟩ ⟨brac.cl.⟩ |

⟨identifier⟩ ⟨brac.op.⟩ ⟨stack limit⟩ ⟨brac.op.⟩ ⟨cell index⟩ ⟨brac.cl.⟩ ⟨brac.cl.⟩

⟨stack object⟩ = ⟨indexed stack id⟩ | NEXTUP

```
                <indexed stack id>|NEXTDOWN
<indexed stack id>
<stack id> = <stack object>| <identifier>
<indexed que id> = <identifier> <brac.op.>
                   <index> <brac. cl.> |
<identifier> <brac.op.> <que limit> <brac.cl.> |
<identifier> <brac.op.> <index> <brac.op.>
<cell index> <brac.cl.> <brac.cl.>|<identifier>
<brac.op.> <que limit> <brac. op.> <cell
 index> <brac.cl.> <brac.cl.>
<que object> = <indexed que id>|NEXTFRONT
               <indexed que id>|NEXTBACK
               <indexed que id>
<que id> = <que object>| <identifier>
<table index> = <identifier> |<quoted string>
<indexed table id> = <identifier> <brac.op.>
                     <table index> <brac.cl.>|
<identifier> <brac.op.> <table index> <brac.op.>
<cell index> <brac.cl.> <brac.cl.>
<identifier> = <stack id>| <que id>| <table id> |
               <indexed nonterminal>†
```

† As described in Chapter 3.

The data structures available in the MSEAL are stacks, ques and tables. Each element of these objects and the simple variables can have a quoted string, or an integer as a value. There is no limit to the size of strings. A method is provided to index different elements of stacks, ques and tables. Stack elements are indexed from top towards bottom and que elements from back towards front. If an indexing identifier has two references, the first one refers to the element while the second to the particular cell in it. On evaluating the first index, if it is an integer, the element is indexed by counting the elements. On the other hand if it is a string, the element is determined by matching the string with the first cell of different elements. If there is only one index in an identifier, it is considered as the first one.

An identifier alone refers to a whole object. No two objects may have the same identifier.

EXAMPLE 5.1

a) STAC (TOP)

STAC is the name of the data object.

TOP is the only index in this case.

The whole identifier therefore refers to the

top most element of the stack STAC.

b) STAC (TOP (2))

STAC is the name of the data object (a stack in this case), TOP is the first index which refers to the top element of the stack. "2" is the second index and is considered as the cell index. The whole identifier refers to the 2nd cell of the top most element of the stack STAC.

c) ABC ("STRING" (3))

ABC is the name of the data object, STRING is the contents of the first cell of the required element.

'3' indicates that after finding the required element, its 3rd cell is to be referred.

Depending upon whether the data object is a stack, que or a table, the element is searched in the usual manner. The first element found is assumed to be the desired one.

5.5 THE ENVIRONMENT AND THE CODE FIELDS:-

The description of the environment and the code fields is short and informal and appears first. We shall devote the rest of this chapter to describing the action field.

The environment field determines the context in which the semantic production in hand is to be activated. Its entries will be called environment expressions. An environment expression can be either a part or whole of the right hand side of a syntactic production or can be formed by using environment symbols and relation operators.

When any symbol belonging to the vocabulary of the grammar of a language is used in specifying its semantics, it is always indexed. In line with this strategy, the whole or part of a production used as an environment expression is represented by indexing its constituents. The consecutive symbols of a production can be represented by indexing the leftmost symbol among them as described in the 3rd chapter and introducing the numbers of the subsequent symbols preceded by semicolons. For example in the production.

5.5.1 < X > = < Y > < Z > < T >

< Z > < T > can be represented by indexing <Z> and then introducing ;3 after it.

<Z> is indexed as < X > 1, 2 i.e. the second symbol of the first alternative of a production whose left hand side is < X > . Hence < Z > < T > will be referred to as < X > 1, 2; 3. If only some right hand symbols of a production form an environment expression, they are preceded by the system variable DUMMY.

The above method of forming environment expressions is very useful if the user wants to specify the semantics in terms of the syntax. If however it is desirable to perform semantic synthesis independently of the syntax of the language, the environment expression can be performed independently. Any identifier, nonterminal, indexed nonterminal, quoted string (terminals are treated as strings) or a MTL system variable can be treated as an environment expression. Alternatively they can be combined with the following symbols.

:: The relation is satisfied if the values of both sides are the same.

¬:: The relation is satisfied if values of its sides are not the same.

_ The symbol on its left hand side should be on the top of the system defined stack STACK.

@ The symbol on its left hand side should be the current symbol of the source statement.

The right hand sides of "-" and "@" can be members of the vocabulary of the grammar. This symbol by itself or any one of its descendants should match the current symbol. The logical OR operation is represented by "|" and the logical AND operation by "&". The latter has priority over the former but this can be overridden by bracketing. Nesting of brackets is not allowed.

If (1) is to be considered as an environment expression, it will be written as

5.5.2 DUMMY <X> 1, 2; 3

EXAMPLE 5.2

a) 'ABE' | 'ABD' - 'ABC'

This is true either

if the current symbol is ABE
or if the current symbol is ABC and
the symbol at top of the STACK is ABD.

EXAMPLE 5.3

b) ('ABE' | 'ABD' - 'ABC') & X :: 5

This is true if (a) is true and
X is equal to 5.

There is a great deal of freedom to a user in the code field. This field can have any number of identifiers and quoted strings. If the SEAS is being used for semantic synthesis, any indexed symbol of the current syntactic production can also be used. The code generated by the execution of a code field is the concatenation of all the strings appearing in it. Where appropriate, the format is controlled by

/ generate end of line

£ generate end of page

5.6 ACTION FIELD:-

It is in this field that most of the semantic action takes place. A user can define an arbitrary number of stacks, ques and tables. Various facilities have been provided for searching, deleting and transferring data from one data object to another. All statements either succeed or fail. If a semantic incompatibility is detected in any statement, it fails otherwise it succeeds. Conditional statements have been provided to make various checks on different data objects. Depending

upon whether a statement fails or succeeds, transfer of control can be directed by branch statements. Assignment statements and MTL system variables provide extra power for data manipulation. Any two statements in the action field are separated by an ampersand. There must be at least one blank between an ampersand and the statement which follows. A label starts immediately after an ampersand or " ->" as appropriate.

5.61 STACK STATEMENTS:-

⟨coma⟩ = ,

⟨cell⟩ = ⟨identifier⟩ | ⟨integer⟩ | ⟨string⟩
SYMBNOW | Λ

⟨element⟩ = &element& &coma& &cell& | &cell&

⟨push statement⟩ = PUSH ⟨element⟩ IN† ⟨variable⟩

⟨stack delete command⟩ = DELETEUP | DELETEDOWN

⟨stack search command⟩ = SEARCHUP | SEARCHDOWN

⟨stack command qualifier⟩ = ⟨indexed stack id⟩ |
⟨indexed stack id⟩ UNTIL† EXHAUSTED |
⟨indexed stack id⟩ UNTIL† ⟨indexed stack id⟩

⟨stack delete statement⟩ = ⟨stack delete command⟩
⟨stack command qualifier⟩

⟨stack search statement⟩ = ⟨stack search command⟩
⟨stack command qualifier⟩

† Blanks must appear on each side.

<stack statement> = <push statement> |
<stack delete statement> |
<stack search statement>

The PUSH statement enters an element of the stack at its top. If there is no such stack in the system, a new stack is created, and then the new element pushed in it. The SEARCHUP and the DELETEUP commands initiate their respective search and delete operations from a particular point upwards. The SEARCHDOWN and the DELETEDOWN commands initiate their respective operations from a particular point downwards. The values of elements and that of cells are assigned to VALELEMENT and VALCELL respectively. Their indexes are assigned to INDELEMENT and INDCELL respectively. If the search fails the previous values remain unaltered.

There are two system defined stacks:

SEAS and STACK. The behaviour and use of SEAS will be discussed at length separately.

The STACK has only one cell in each element.

Among other things, it can be used to communicate between MSEAS and SNOBOL4 extension programs. In the latter case, it appears as an array named STACK having 80 elements.

5.62 QUE STATEMENTS:-

⟨register statement⟩ = REGISTER ⟨element⟩ IN† ⟨identifier⟩

⟨que delete command⟩ = DELETEON | DELETEBACK

⟨que search command⟩ = SEARCHON | SEARCHBACK

⟨que command qualifier⟩ = ⟨indexed que id⟩ | ⟨indexed que id⟩ UNTIL EXHAUSTED | ⟨indexed que id⟩ UNTIL† ⟨indexed que id⟩

⟨que delete statement⟩ = ⟨que delete command⟩ ⟨que command qualifier⟩

⟨que search statement⟩ = ⟨que search command⟩ ⟨que command qualifier⟩

⟨que statement⟩ = ⟨register statement⟩ | ⟨que delete statement⟩ | ⟨que search statement⟩

The allowed operations on ques are quite similar to those of stacks. The REGISTER statement is used to make a new entry in a que. If the que already exists the entry is made at the back of the que; otherwise a new one is created. The

† Blanks must appear on each side.

SEARCHON and the DELETEON commands initiate their respective operations from a particular point towards the front of the que. On the other hand the SEARCHBACK and the DELETEBACK commands initiate their respective actions from a particular point in the que backwards. There is no system defined que.

5.63 TABLE STATEMENTS:-

⟨enter statement⟩ = ENTER ⟨element⟩ IN† ⟨identifier⟩

⟨table delete statement⟩ = DELETE ⟨indexed table id⟩

⟨table search statement⟩ = SEARCH ⟨indexed table id⟩

⟨table statement⟩ = ⟨enter statement⟩ | ⟨table delete statement⟩ | ⟨table search statement

Unlike stacks and ques, a user can not specify the direction of a table operation. The ENTER statement makes a new entry in a table, the delete statement deletes an already existing entry and the SEARCH statement searches an entry.

† Blanks must appear on each side.

An element of a data object consists of any numbers of cells separated by comas. There is no explicit declaration statement but data objects are automatically declared at the time of making the first entry. The number of cells in each element of a data object is the same. The statement

PUSH "AB", "C", "D" IN AA.

declares a stack AA if it does not exist already, with each element having three cells. The first element of the stack is initialised to have "AB", "C", and "D" in its cells in the same order. If AA exists already the element "AB", "C", "D" is pushed in it. A stack command operates on its argument. A DELETEDOWN command requires the deletion of the whole or part of a stack starting from the specified point downwards. The case is opposite for the command DELETEUP. The information about the name of the stack and the particular starting point for deletion is acquired from the first argument. For instance the statement

DELETEDOWN STACK (TOP)

will delete the top of the STACK and the statement

DELETEDOWN STACK (TOP) UNTIL STACK (3)

deletes top three elements of the STACK. If some middle part of a data object, is to be deleted

the skeleton remains but the deleted cells are initialised to null string. SEARCHDOWN and SEARCHUP commands search a stack from a particular point towards the bottom and top respectively. The specification of the name, starting and finishing points of the search is similar to that of the delete operations. On successful search, different values are assigned to the following as appropriate

VALELEMENT	Value of the whole element searched
VALCELL	Value of the searched cell
INDELEMENT	Index of the element searched
INDCELL	Index of the cell searched.

If only some of the above mentioned MTL system variables get new values in a successful search, rest of them are assigned null string. If however a search fails the value of the above mentioned MTL system variable remains unaltered. If due to some semantic reason an operation on any one of the data objects is not possible, the statement fails.

5.64 ASSIGNMENT STATEMENT:-

<operator> = +|-|*|/

<unary operator> = +|-

```
<unsigned operand> = <integer> | <identifier>
<operand> =  &unary operator& <unsigned operand> |
                                <unsigned operand>
<arithmetic exp> = <arithmetic exp>  <operator>
                   <operand> | <operand>
<search value> =  VAL | INDELEMENT | INDCELL
<search statement> = <stack search statement> |
<que search statement> | <table search statement>
<assignment body> = <arithmetic exp> |
   <search value> | <search statement> | NULL
<system variable> =  DELIMITER|VAR|NEWVAR|SYMBNOW
<boolean value> =  0|1
<string value> = <quoted string> | <identifier>
<string exp> = <string exp>  <string value> |
               <string value>|<system variable> |
                CODE
<assignment variable> =  RETURN|STOP|SEAS
<device> =  PRINT|PUNCH|TAPE|DISC
<assignment id> = <identifier> |CODE| <indexed
 stack id> |<device>| <indexed que id> |
<indexed table id>
<assignment sign> =  :=
<assignment statement> = <assignment id><assignment sign>
                                <string exp> |
<assignment id> <assignment sign> <assignment body> |
                        <assignment sign> <string exp> |
<assignment variable> <assignment sign> <boolean value>
```

The evaluation of an arithmetic expression takes place from left to right with no operator precedence. The value of a string expression is the concatenation of all the strings appearing in it. Every time the system variable "NEWVAR" is executed, a new variable is generated. Its value is automatically assigned to "VAR" and can be used later. The system variable "DELIMITER" by default has a blank as its value. But this value can be changed by an assignment statement. When the code for a full source statement, is generated, the delimiter is inserted at its end. Execution of the statement := CODE has the effect of handing the current value of CODE to the system which in turn preserves it at the appropriate place (as described later). On completing the analysis of a source statement code is generated on the output device.

5.65 MISCELLANEOUS STATEMENTS:-

⟨segment command⟩ = SEGMENT-

⟨segment statement⟩ = &segment command& ⟨integer⟩ |
&segment command& ⟨variable⟩

⟨eliminate statement⟩ = ELIMINATE | ELIMINATE
&segment statement& &coma& ⟨integer⟩ | ELIMINATE
&segment statement& &coma& ⟨variable⟩

<reinstate statement> = REINSTATE | REINSTATE
&segment statement& &coma&<integer> |
REINSTATE <segment statement> &coma&
<variable>

<code statement> = CODE | CODE <string exp>

<jump> = CONTINUE <alphanumeric>

<miscellaneous statement> = <segment statement> |
<eliminate statement> | <reinstate statement> |
<code statement> |
CLEAR | CURSOR | IGNORE †

A segment is a part of semantic specification which can be called for action as an independent piece of specification. A segment is named by a segment statement.

An eliminate statement temporarily eliminates a semantic production. On any subsequent occasion, it is considered non existent until it is reinstated by a reinstate statement. If the command ELIMINATE has no argument, the current statement is assumed. Otherwise the segment is specified explicitly and the production number is separated from it by a coma. A statement CODE with no arguments generates code as specified in the code field

† it can be followed by a string of valid characters.

of the current semantic production, otherwise that specified by its arguments is generated. The IGNORE statement allows comments to be introduced. The whole statement is ignored. On executing the CURSOR statement, "£" is printed under the current cursor position.

5.66 TRANSFER OF CONTROL STATEMENTS:-

⟨affirmative predicate⟩ = EQ|LT|GT|LE|GE

⟨negation⟩ = N

⟨negative predicate⟩ = &negation& ⟨affirmative predicate⟩

⟨predicate⟩ = ⟨affirmative predicate⟩ | ⟨negative predicate⟩

⟨argument⟩ = ⟨operand⟩ | ⟨quoted string⟩

⟨test statement⟩ = ⟨predicate⟩ ⟨op.brac.⟩ ⟨argument⟩ ⟨coma⟩ ⟨argument⟩ ⟨cl.brac.⟩

⟨conditional statement⟩ = ? ⟨statement⟩

⟨transfer of control statement⟩ = ⟨test statement⟩ | ⟨conditional statement⟩

⟨statement⟩ = ⟨stack statement⟩ | ⟨que statement⟩ | ⟨table statement⟩ | ⟨assignment statement⟩ ⟨jump⟩

The significance of each of the affirmative predicates is as suggested by the mnemonic. When prefixed by N, the predicate is negated. The result of evaluating a test statement is to assign the appropriate value, TRUE or FALSE, to the system variable TEST. This value remains accessible in the succeeding statement.

To execute a conditional statement, the value of the variable TEST is checked (its value would have been effected by the previous statement). If it is true, the part of the current statement following "?" is executed, otherwise the control is passed to the next statement.

5.67 SEAS:-

This is a system defined stack with two cells in each element. It is controlled by the system and developes and collapses automatically. In the first cell of each element is kept a particular indexed nonterminal of the grammar. The second cell holds the source string produced from it. When semantic action is to be taken on recognising part or whole of a MBNF production,

the values of its constituents appear in the top elements of the SEAS, the rightmost constituent of a production being at the top of the SEAS. For example in the grammar

5.67.1 < y > = a b c

5.67.2 < x > = (< y >)

the SEAS is shown in the diagram

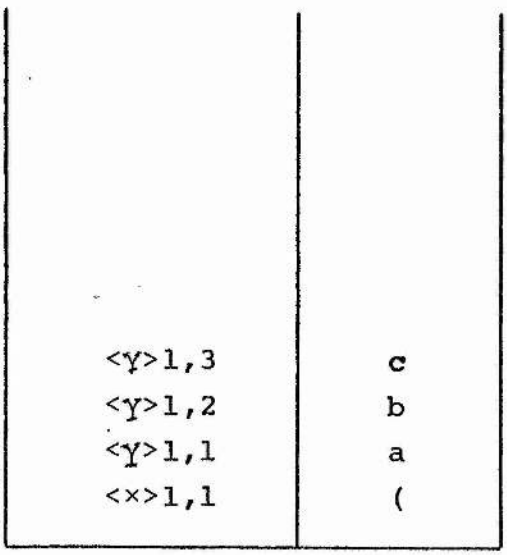


FIG. 5.1

On recognising a semantic production, parts of the string covered by the production are accessible. These strings appear as values of the nodes of the syntax graph at the nearest possible state of the production. For example if the environment field recognised is <y>1, 1; 2; 3, the values available will be

<y>1, 1 = a

<y>1, 2 = b

<y>1, 3 = c

On the other hand if the environment field recognised is <x>1, 1; 2; 3 the available values will be

< x > 1, 1 = (

< x > 1, 2 = a b c

< x > 1, 3 =)

EXAMPLE 5.3

The semantic production to be executed is

< x > 1,1;2;3 -> CODE < y > 1,1 & CODE < y > 1,2 &
CODE < y > 1,3 & := CODE :

The first three statements generate a b c as code, and then this code is handed over to the system by the last statement. The system keeps the SEAS up to date. On completing the execution of the action, all cells of SEAS above the one representing the left most symbol in the environment field (<x> 1,1 in the current case) are deleted.

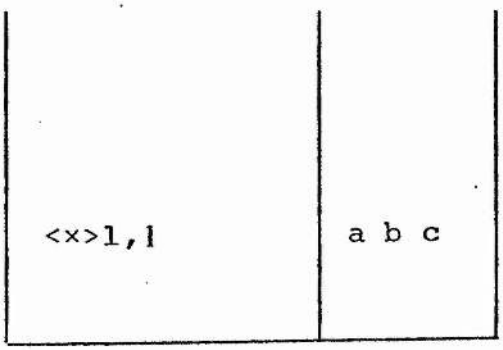


FIG. 5.2

5.7 OVERALL STRATEGY:-

The semantic definition of a programming language follows its syntactic definition separated by the command %SEMANTICS. This definition may be in anyone of the two modes of specification i.e. the production mode and the relation mode. To select a mode, the %SEMANTICS command may be followed by PRODUCTION or RELATION as appropriate.

In the production mode, all the environment relations are whole or parts of MBNF productions. In this mode SEAS is available and user is advised to use it. In the relation mode SEAS is not accessible.

In either case the semantic definition may consist of one or more segments, the top most being "SEGMENT-0". Each segment of the semantic definition is an ordered set of three field productions. Productions with empty environment field are executed only once i.e. at the start of execution. Recognition of the environment field takes place from top towards bottom, the first match being considered the valid one.

Eliminated statements are ignored. On recognising each non-auxiliary word, control is passed to the semantic synthesiser which processes the SEGMENT-O. When the environment field of a semantic statement is matched, its action field is executed. Execution of the action field is complete when its last statement is executed or on branching to label "STOP" The execution of a segment is complete if no environment field matches in the whole segment or if at least one semantic statement is executed and the value of "STOP" is "1". If the value of "STOP" is "O", on completing the execution of a semantic statement, the same segment is processed again starting from the top most semantic statement of that segment. On branching to "RETURN", if the value of "RETURN" is "O", the execution of the current segment is considered to be complete and if the value of "RETURN" is "1", the current semantic process is considered to be complete. In this, control is passed back to the syntax analyser.

All symbols of the MTL are reserved symbols. However this specification can be overridden by preceding any such symbol by an asterisk. Two consecutive asterisks give the effect of a single asterisk as if it is not a symbol of MTL. The

generated code is kept internally in the form of a string as value of "CODE". It is generated on the output stationary only after the whole of a source language statement has been recognised. The output device can be selected by the device assignment statement. Execution of "CURSOR" prints "£" under the current cursor position.

TABLE 5-1

In this example, the MTL specification of a subset of the programming language PATE is given. Its syntax and the semantics are separated by the control card "%SEMANTICS RELATION". This means that its semantic synthesis is to be performed in the relation mode.

At this stage we will not elaborate the syntax any more since it is explained in chapter 8. The nonterminal "NUMBER" is not defined by a MBNF production but it is defined by an MBNF function. The source language statements are processed and polish notation generated for them. Different stages of this process are also shown.

Since the semantic synthesis is performed in the relation mode, the manner in which the syntax is written has no effect on it. The parser is used to detect errors and give diagnostics. The system defined stack "STACK" is used for semantic synthesis and basically the tasks shown in the following flow chart are performed.

```
1   <ADDITION OPERATOR> = PLUS | ADDED TO
2   <ADDITION FORMULA1> = <NUMBER> ¬ <ADDITION OPERATOR> <NUMBER> ¬N -
3   <ADDITION OPERATOR> <NUMBER>
4   <ADDITION FORMULA2> = <FUNCTION> ¬ ; <ADDITION OPERATOR> -
5   <FUNCTION> ¬N ; <ADDITION OPERATOR> <FUNCTION>
6   <MULTIPLICATION OPERATOR> = TIMES | MULTIPLIED BY
7   <MULTIPLICATION FORMULA1> = <NUMBER> ¬ <MULTIPLICATION OPERATOR> -
8   <NUMBER> ¬N <MULTIPLICATION OPERATOR> <NUMBER>
9   <MULTIPLICATION FORMULA2> = <FUNCTION> ¬ ; <MULTIPLICATION OPERATOR> -
10  <FUNCTION> ¬N ; <MULTIPLICATION OPERATOR> <FUNCTION>
11  <ADDITION FORMULA> = <ADDITION FORMULA1> | <ADDITION FORMULA2>
12  <MULTIPLICATION FORMULA> = <MULTIPLICATION FORMULA1> -
13   | <MULTIPLICATION FORMULA2>
14  <MULTIPLICATION FORMULA1>1,1 = <MULTIPLICATION FORMULA2>1,1
15  <ADDITION FORMULA1>1,3 = <ADDITION FORMULA2>1,4
16  <ADDITION FORMULA2>1,4 = <ADDITION FORMULA1>1,3
17  <MULTIPLICATION FORMULA1>1,3 = <MULTIPLICATION FORMULA2>1,4
18  <MULTIPLICATION FORMULA2>1,4 = <MULTIPLICATION FORMULA1>1,3
19  <ADDITION FORMULA1>1,5 = <ADDITION FORMULA2>1,7
20  <ADDITION FORMULA2>1.7 = <ADDITION FORMULA1>1,5
21  <MULTIPLICATION FORMULA1>1,5 = <MULTIPLICATION FORMULA2>1,7
22  <MULTIPLICATION FORMULA2>1,7 = <MULTIPLICATION FORMULA1>1,5
23  <ADDITION FORMULA2>1,3 = <MULTIPLICATION FORMULA2>1,3
24  <ADDITION FORMULA1>1,1 = <ADDITION FORMULA2>1,1
25  <ADDITION FORMULA1>1,3 = <ADDITION FORMULA2>1,4
26  <ADDITION FORMULA2>1,4 = <ADDITION FORMULA1>1,3
27  <ADDITION FORMULA1>1,2 = <MULTIPLICATION FORMULA1>1,2
28  <MULTIPLE COMMAND> = SUM | MULTIPLY | PRODUCT | ADD
29  <SEPARATOR> = , | AND
30  <DIVISION OPERATOR> = OVER | DIVIDED
31  <SUBTRACTION OPERATOR> = MINUS
32  <DIADIC OPERATOR> = <SUBTRACTION OPERATOR> | <DIVISION OPERATOR>
33  <DIADIC FORMULA1> = <NUMBER> ¬ <DIADIC OPERATOR> <NUMBER>
34  <DIADIC FORMULA2> = <FUNCTION> ¬ ; <DIADIC OPERATOR> <FUNCTION>
35  <DIADIC FORMULA> = <DIADIC FORMULA1> | <DIADIC FORMULA2>
36  <MULTIPLICATION FORMULA2>1,3 = <DIADIC FORMULA2>1,3
37  <FORMULA> = <ADDITION FORMULA> | <MULTIPLICATION FORMULA> -
38   | <DIADIC FORMULA>
39  <DIADIC BODY> = <FORMULA> <SEPARATOR> <FORMULA>
40  <DIADIC COMMAND> = DIFFERENCE | QUOTIENT
41  <DIADIC FUNCTION> = <DIADIC COMMAND> <DIADIC BODY>
42  <DIADIC BODY>1,3 = <ADDITION FORMULA2>1,3
43  <MULTIPLICATION FORMULA2>1,3 = <DIADIC FORMULA1>1,2
44  <MULTIPLICATION FORMULA1>1,2 = <DIADIC FORMULA1>1,2
```

TABLE 5·1

```
45          <BODY> = <FORMULA> <SEPARATOR> <FORMULA> ¬N <SEPARATOR> <FORMULA>
46          <MULTIPLE FUNCTION> = <MULTIPLE COMMAND> <BODY>
47          <FUNCTION> = <MULTIPLE FUNCTION> | <DIADIC FUNCTION>
48          <EXPRESSION> = <FORMULA> | <FUNCTION>
49          <STATEMENT> = <EXPRESSION> .
50          %SEMANTICS RELATION
51          *** UNDEFINED NONTERMINALS
52          NUMBER
53          ************
54          ';' * 'MINUS' -> DELETE STACK(TOP) & PUSH '-' IN STACK                :
55
56          ';' * 'OVER' | ';' * 'DIVIDED' -> DELETE STACK(TOP) &                  -
57          PUSH '/' IN STACK                                                     :
58
59          ';' * <MULTIPLICATION OPERATOR> -> DELETE STACK(TOP) &                 -
60          PUSH '*' IN STACK                                                     :
61
62          ';' * <ADDITION OPERATOR> ->  DELETE STACK(TOP) & PUSH '+' IN STACK   :
63
64          '+' - ',' | '/' - ',' | '-' - ',' | '*' - ',' -> DELETE STACK(TOP)    :
65
66          '+' - <SEPARATOR> | '/' - <SEPARATOR> | '-' - <SEPARATOR> |           -
67          '*' - <SEPARATOR> -> DELETE STACK(TOP)                                :
68
69          '+' - <ADDITION OPERATOR> ->                                          :
70
71          <ADDITION OPERATOR>       -> PUSH '+' IN STACK                        :
72
73          '*' - <MULTIPLICATION OPERATOR> ->                                    :
74
75          <MULTIPLICATION OPERATOR>  -> PUSH '*' IN STACK                       :
76
77          '-' - <SUBTRACTION OPERATOR> ->                                       :
78
79          <SUBTRACTION OPERATOR> -> PUSH '-' IN STACK                           :
80
81          '/' - <DIVISION OPERATOR> ->                                          :
82
83          <DIVISION OPERATOR>     -> PUSH '/' IN STACK                          :
84
85          'ADD' | 'SUM' -> PUSH '/+' IN STACK                                   :
86
87          'MULTIPLY' | 'PRODUCT'  -> PUSH '/*' IN STACK                         :
88
```

```
'QUOTIENT' -> PUSH '//' IN STACK                                        :

'DIFFERENCE' -> PUSH '/-' IN STACK                                      :

'ADD' * <NUMBER> | 'SUM' * <NUMBER> | 'MULTIPLY' * <NUMBER> |           -
'DIFFERENCE' * <NUMBER> | 'QUOTIENT' * <NUMBER> |                       -
'PRODUCT' * <NUMBER> -> CODE SYMBNOW                                    :

'/+' - <NUMBER>        -> CODE SYMBNOW & CODE '+'                       :

'/*' - <NUMBER>        -> CODE SYMBNOW & CODE '*'                       :

'/-' - <NUMBER>        -> CODE SYMBNOW & CODE '-'                       :

'//' - <NUMBER>        -> CODE SYMBNOW & CODE '/'                       :

'+' - <NUMBER> | '*' - <NUMBER> | '-' - <NUMBER> |                      -
'/' - <NUMBER> -> CODE SYMBNOW  & CODE STACK(TOP)                       :

<NUMBER>  -> CODE SYMBNOW                                               :

%GENERATE
%SNOBOL
 DEFINE('NUMBER()','Z7') :(Z8)
Z7 CARD LEN(*INPOS) BREAK(' ') . ZVAL
 CONVERT(ZVAL,'REAL') :S(Z1)
 OBSTACLE = 1
 MATCHED = 0 :(RETURN)
Z1 MATCHED = 1
 OBSTACLE = 0 :(RETURN)
Z8 &ANCHOR = 1
%FINISH
ADD 5 , 6 .
5 6 +
MULTIPLY 6 AND 9 .
6 9 *
   ADD 5 , 456 .
5 456 +
ADD 4 2
      £
*** THIS SYMBOL = AND OR ITS EQUIVALENT
ADD 5 , 6 , 7 , 6 .
5 6 + 7 + 6 +
ADD 5 PLUS 6 PLUS 9 , 9 PLUS 2 .
```

```
133             5 6 + 9 + 9 + 2 +
134             MULTIPLY 5 PLUS 8 , 6 , 3 .
135             5 8 + 6 * 3 *
136             ADD 4 , 4 .+
137                       £
138             *** THIS SYMBOL = . OR ITS EQUIVALENT
139             5 PLUS 6 .
140             5 6 +
141             5 PLUS 6 PLUS 7 PLUS 8 .
142             5 6 + 7 + 8 +
143             MULTIPLY SUM 5 , 6 , 7 ; AND 8 .
144                                        £
145             *** ERROR FOUND
146             PRODUCT 5 , 6 , 7 , PLUS 8 .
147                                  £
148             *** THIS SYMBOL = NUMBER OR ITS EQUIVALENT
149             PRODUCT 5 , 6 , 7 ; PLUS 8 .
150             5 6 * 7 *
```

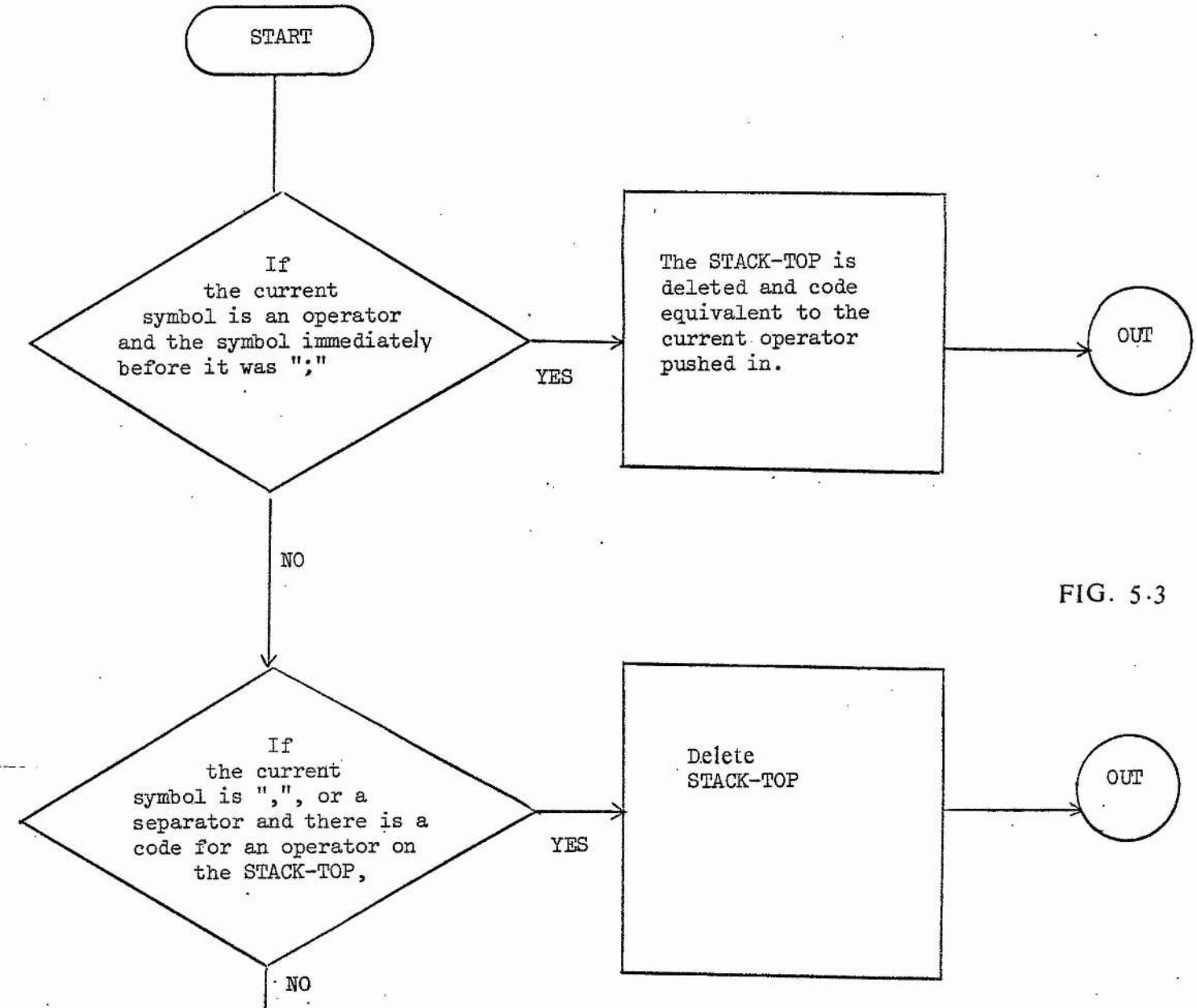
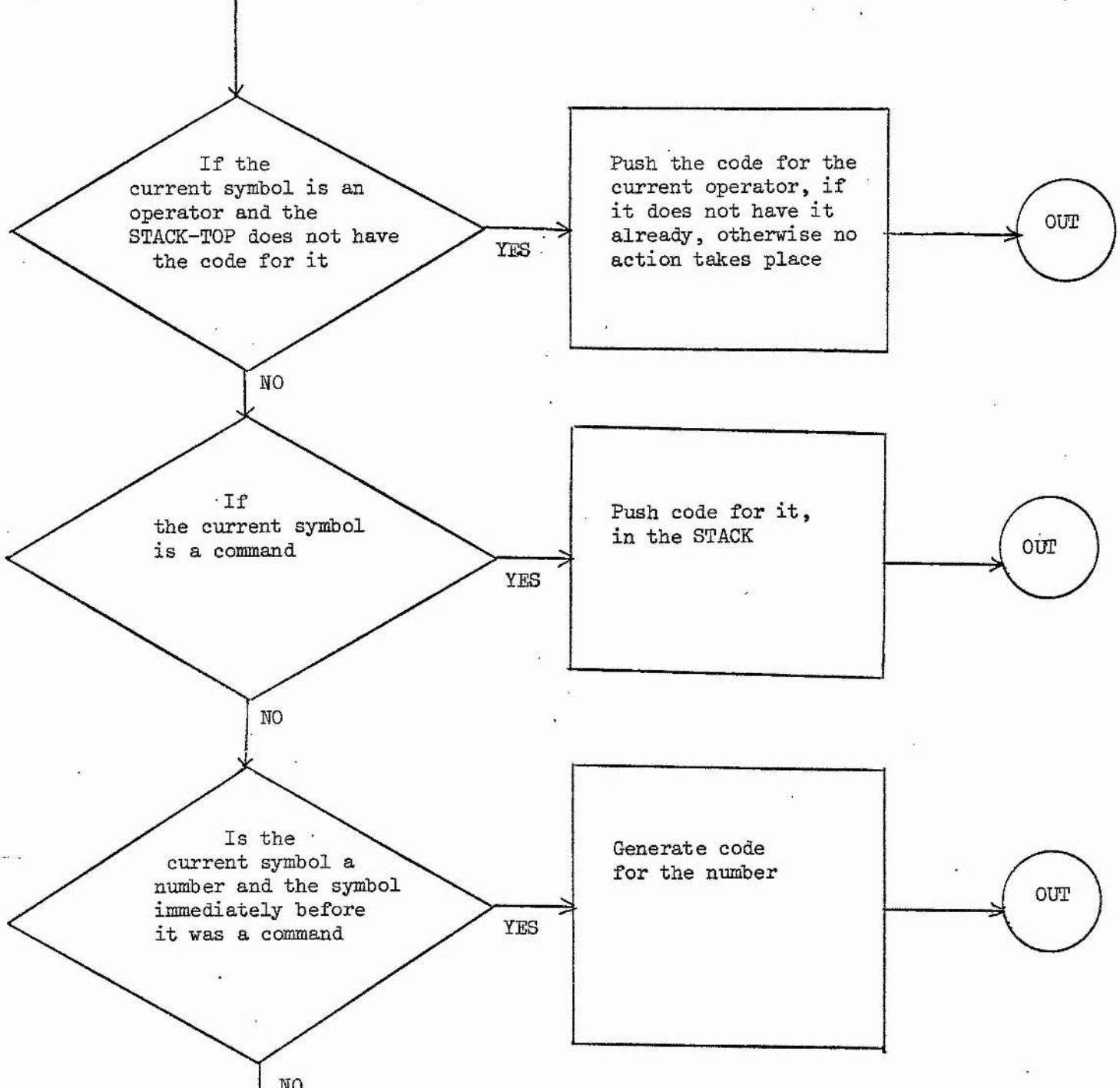

START
If the current symbol is an operator and the symbol immediately before it was ";"
YES
The STACK-TOP is deleted and code equivalent to the current operator pushed in.
OUT
NO
If the current symbol is ",", or a separator and there is a code for an operator on the STACK-TOP,
YES
Delete STACK-TOP
OUT
NO

FIG. 5·3


If the current symbol is an operator and the STACK-TOP does not have the code for it
YES
Push the code for the current operator, if it does not have it already, otherwise no action takes place
OUT
NO
If the current symbol is a command
YES
Push code for it, in the STACK
OUT
NO
Is the current symbol a number and the symbol immediately before it was a command
YES
Generate code for the number
OUT
NO

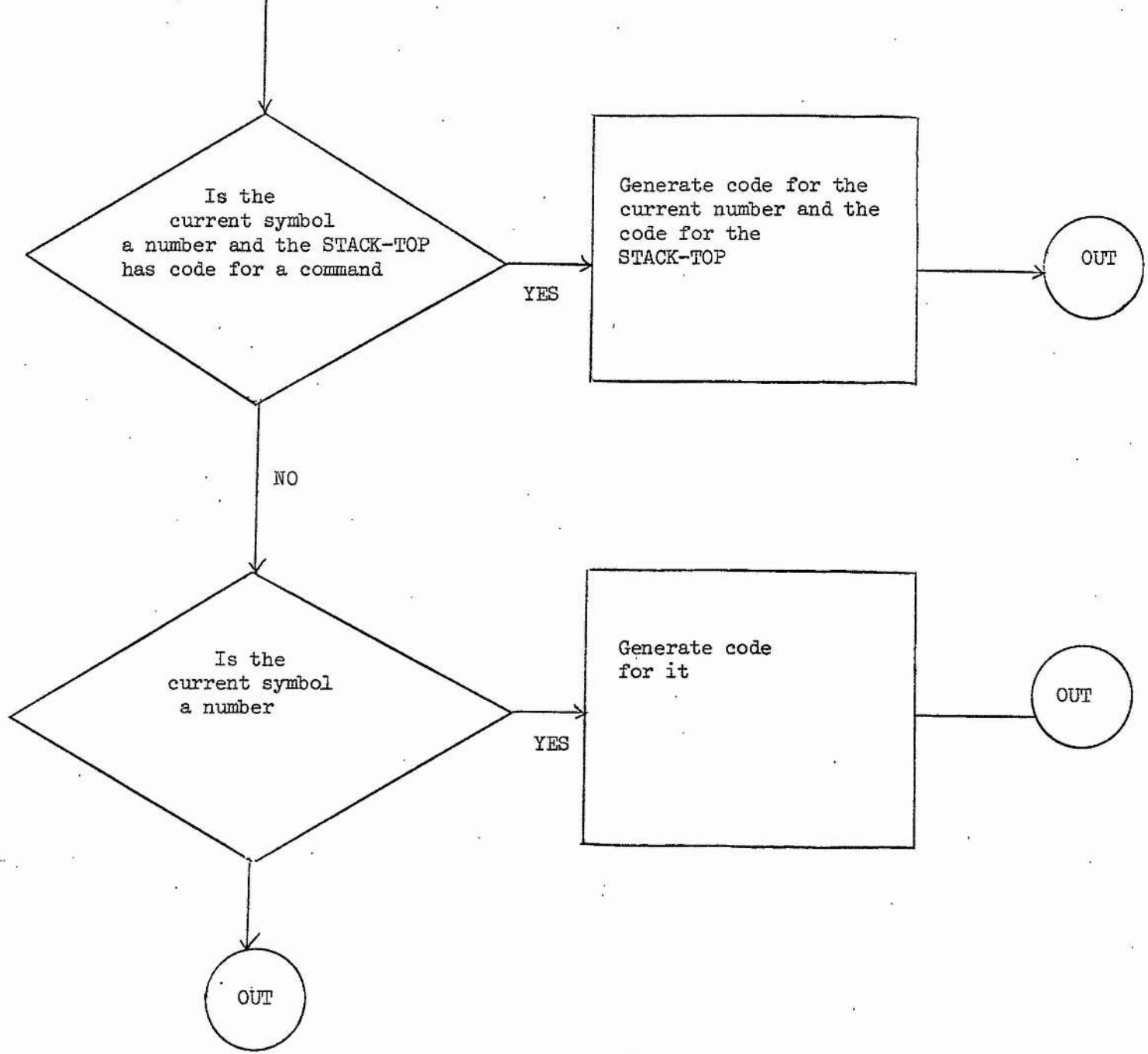

Is the
current symbol
a number and the STACK-TOP
has code for a command
YES
Generate code for the
current number and the
code for the
STACK-TOP
OUT
NO
Is the
current symbol
a number
YES
Generate code
for it
OUT
OUT

Table 5.2 describes the processing of an assignment statement. The table is formed of three blocks of information. The topmost block specifies the syntax of the assignment statements. The terminal alphanumeric is defined by SNOBOL4 which has been added to the MTL processor. The semantics is separated from the syntax by the control card %SEMANTICS PRODUCTION. It specifies that the semantic synthesis is to be done in the production mode. The processed assignment statements are displayed following the %GENERATE command.

The object of this exercise is to generate polish notation for assignment statements. The syntax analysis is performed by the syntax analyser which has been discussed in detail in chapters 3 and 4.

In this example, there are three semantic productions which perform the following task:

a) In an arithmetic expressions without brackets
   i) as soon as a multiplying operator and its trailing operand are recognised, their order is changed.
   ii) when the whole expression is recognised, the SEAS is scanned and complete polish

notation generated.

b) Bracketed expressions are evaluated and then their values treated as ordinary operands.

c) The expressions inside brackets are considered as independent expressions and are treated as described in (a) and (b).

```
<VARIABLE> = ALPHANUMERIC
<ADDING OPERATOR> = + | -
<MULTIPLYING OPERATOR> = * | /
<OPERAND> = ( <ARITHMETIC EXP> ) | <VARIABLE>
<ADDING EXP> = <OPERAND> ¬N <ADDING OPERATOR> <OPERAND>
<MULTIPLYING EXP> = <OPERAND> ¬N <MULTIPLYING OPERATOR> <OPERAND>
<ARITHMETIC EXP> = <ADDING EXP> | <MULTIPLYING EXP>
<ASSIGNMENT STATEMENT> = <VARIABLE> := <ARITHMETIC EXP>
<MULTIPLYING EXP>1,2 = <ADDING EXP>1,2
<ADDING EXP>1,2 = <MULTIPLYING EXP>1,2
$SEMANTICS PRODUCTION
<OPERAND>1,1;2;3 -> Z := NULL &                                          -
Z1 := IND SEARCH SEAS(<OPERAND>1,1) & Z1 := IND NEXTUP SEAS(Z1) &        -
CODE SEAS(Z1(2)) &LABEL1 Z1 := IND NEXTUP SEAS(Z1) &                     -
?S (LABEL3) &LABEL4 CODE Z & := CODE & CODE := NULL & (RETURN)           -
&LABEL3    EQ ( SEAS(Z1(1)) , <ADDING EXP>1,2 ) & ?S (LABEL2)            -
& EQ ( SEAS(Z1(1)) , <OPERAND>1,3 ) & ?S (LABEL4)                        -
& CODE SEAS(Z1(2)) & (LABEL1) &LABEL2 CODE Z &                           -
Z := SEAS(Z1(2)) & (LABEL1)  :

DUMMY <MULTIPLYING EXP>1,2;3 -> CODE <MULTIPLYING EXP>1,3                -
& CODE <MULTIPLYING EXP>1,2  & := CODE & CODE := NULL  :

<ASSIGNMENT STATEMENT>1,1;2;3 -> Z := NULL &                             -
CODE <ASSIGNMENT STATEMENT>1,3 & ?F (LABEL5) &LABEL6 CODE Z &            -
 CODE <ASSIGNMENT STATEMENT>1,1 & CODE <ASSIGNMENT STATEMENT>1,2         -
& := CODE & CODE := NULL & (RETURN)                                      -
&LABEL5 Z := NULL & Z1 := IND SEARCH SEAS(<ADDING EXP>1,1) &             -
CODE SEAS(Z1(2)) &LABEL1 Z1 := IND NEXTUP SEAS(Z1) &                     -
?S (LABEL3) & (LABEL6)                                                   -
&LABEL3    EQ ( SEAS(Z1(1)) , <ADDING EXP>1,2 ) & ?S (LABEL2)            -
& CODE SEAS(Z1(2)) & (LABEL1) &LABEL2  CODE Z &                          -
Z := SEAS(Z1(2)) & (LABEL1)    :
```

TABLE 5.2

```
%GENERATE

K := ( H + P * D )
H P D * + K :=
N := ( M * ( B * M + L ) )
M B M * L + * N :=
N := ( ( H * E ) + L )
H E * L + N :=
L := ( M * ( J * W + P ) - L )
M J W * P + * L - L :=
B := M + N + K
M N + K + B :=
N := G + M * L
G M L * + N :=
L := B * N * D
B N * D * L :=
N := N + ( M * L )
N M L * + N :=
N := N * ( L + M * D )
N L M D * + * N :=
P := ( B * ( K + L ) )
B K L + * P :=
P := ( * ( M + K ) + U )
          £
*** THIS SYMBOL = ( OR ITS EQUIVALENT
P
L := ( L * ( K + D ) + P )
L K D + * P + L :=
L := ( M * E ) + ( H - U )
M E * H U - + L :=
L := N * ( M * ( J - K ) + D )
N M J K - * D + * L :=
X := Y + C * D + ( A + B - C ) / ( C - D ) + L
Y C D * + A B + C - C D - / + L + X :=
```

CHAPTER 6

IMPLEMENTATION OF THE SEMANTIC SYNTHESISER

6.1 SEMANTIC TABLES:-

On reading the semantics of a programming language, tables of its intermediate language are constructed such that starting from the first semantic production read, the environment fields (E F) of all the statements can be constructed sequentially. The action fields (A F) and the code fields (C F) are accessible through the environment fields of the semantic production in which they appear. Segment names are treated in a manner similar to the environment fields.

← SEGMENT-0→ ←SEGMENT-1→ ←— SEGMENT-2 —→		
E.F.11	A.F.11	C.F.11
E.F.10	A.F.10	C.F.10
E.F.9	A.F.9	C.F.9
E.F.8	A.F.8	C.F.8
E.F.7	A.F.7	C.F.7
SEGMENT - 2		
E.F.6	A.F.6	C.F.6
E.F.5	A.F.5	C.F.5
SEGMENT - 1		
E.F.4	A.F.4	C.F.4
E.F.3	A.F.3	C.F.3
E.F.2	A.F.2	C.F.2
E.F.1	A.F.1	C.F.1

Semantic table

FIG. 6.1

The diagram 6.1 shows the semantic tables for a semantic specification. Three fields of each semantic production are represented by three columns. The bottom production is the first production read. The bottom production in each segment is called the start production of the segment. As is clear from the diagram, starting from EF1, it is possible to access EF2, EF3 ---- EFn

sequentially and from EF.m, AF.mand CF.mare accessible; m and n being positive integers.

On the completion of the syntax and the semantic specification, when a "%GENERATE" command is met, the system goes through all the semantic statements and executes all those which do not have an environment field. After that they are made unaccessible to the system and processing of the source language starts.

6.2 SEMANTIC SYNTHESIS:-

During the syntax analysis of a source language statement, a "history" of the recognition is kept on the SAS. When a node is successfully traversed, it is recorded. In essence it is deleted when the string covered by it is recognised. When semantics of the language is specified in the relation mode, each time a new symbol is recognised, the semantic synthesizer is called for action. In the production mode it is called each time the part of a production enough to form a valid environment expression is recognised.

When the semantic synthesizer is called, an attempt is made to recognise a production in SEGMENT-O. If the match is found, its action field is executed, otherwise the control is passed back to the syntax analyser. In the action field, execution takes place sequentially from left to right unless an explicit transfer of control takes place. The code field is invoked by the execution of a statement in the action field. The string resulting from the execution of a code field is generated as the code.

At the time of constructing the semantic tables, internal code is generated for each semantic production as it is read. The order of the internal code therefore is reversed from the one specified by the user. To ensure top-down recognition for the user specification, the recognition of the semantic productions in any segment always proceeds from bottom upwards.

To recognise a semantic production in any segment, starting from the start production of the segment, an attempt is made to match the environment fields of any one of the productions, the first match considered as the desired one.

To recognise different relations in the environment field, various checks are made on the relation symbols under consideration and on various data types and previously kept symbols. For example to recognise '+' - ',' it is checked that '+' is at the top of the STACK and "," is the symbol currently under consideration. In the relation mode the recognition process is fairly straightforward, but in the production mode it is more sophisticated and needs some explanation and is discussed below.

6.3 RECOGNITION OF ENVIRONMENT EXPRESSIONS:-

Consider that at any stage of parsing N_1, N_2, N_3 ------ N_n are the nodes in SAS such that N_1 is at the top while N_n is at the bottom. N_1 is the immediate descendant of N_2, N_2 of N_3 and so on, N_O is the immediate descendant of N_1. N* is a node in the syntax graph representing the nonterminal under consideration in the environment field. To recognise N*, it is matched with N_i, where $i = 0, 1, 2$ ------ n. i is assumed to be such that N_i does not have a successor node.

If the whole of a production appears as a relation in the environment field, SAS is searched from top towards bottom for a node N_P such that

N_P is descendant of N_{P+1}

and N_{P-1} is the successor of N_P

N_{P-2} is the successor of N_{P-1}

and so on.

N_P, N_{P-1}, N_{P-2} ------ N_O are then matched with the given production in the environment field. If only the r rightmost symbols of a production appear as a relation, then the given environment field is matched with

$N_{P-(P-r)}$, $N_{P-(P-r)-1}$, $N_{P-(P-r)-2}$

For example consider the example 4.3, on recognising the expression A + A, the SAS has the form shown in Fig. 6.2.

If N* is

"A", it matches the most recently recognised symbol.

⟨operand⟩ , it matches the node N_0, which is not on SAS.

N_1	<addition exp>1,1	DAU
N_2	<addition exp>1,3	DAU
N_3	<addition exp>1,2	SUCC
N_4	<addition exp>1,1	SUCC
N_5	<addition exp>	

FIG. 6.2

It is worth noting at this stage, that the ⟨addition exp⟩ can have two different values i.e. ⟨operand⟩ and ⟨operand⟩ ⟨operator⟩ ⟨addition exp⟩. Hence in the semantic specification ambiguity can be caused. It is the users responsibility to avoid it. This however is not a big handicap since in BNF a nonterminal can have more than one different alteratives any way.

On deleting the top cell of the SAS shown in 6.2, it will have the following form.

In this case $\rho = 3$ and ⟨addition exp⟩ 1; 2; 3 matches N_3, N_2 and N_1 while DUMMY ⟨addition exp⟩ 1, 2; 3 matches N_2 and N_1 since $r \triangleq 2$.

N_1	<addition exp>1,3	DAU
N_2	<addition exp>1,2	SUCC
N_3	<addition exp>1.1	SUCC
N_4	<addition exp>	

FIG. 6.3

6.4 THE OVERALL STRUCTURE:-

The semantics of a programming language is described in terms of its syntax. If V_G is the vocabulary of a grammar G and $v_G \in V_G$. Then v_G is treated as an

identifier which always has as its value, the correct code corresponding to the part of the source statement recognised by its descendants at any instant of time.

Consider the following specification

6.4.1 ⟨operand⟩ = A

6.4.2 ⟨operator⟩ = +

6.4.3 ⟨addition exp⟩ = ⟨operand⟩ ⟨operator⟩ ⟨addition exp⟩

6.4.4 %SEMANTICS PRODUCTION

6.4.5 ⟨addition exp⟩ 1, 1; 2; 3 -> CODE ⟨addition exp⟩ 1, 1 & CODE ⟨addition exp⟩ 1, 2 & := CODE & CODE = NULL:

In (5) we may refer to members of V_G (In processing different texts and at different instances in processing the same source statement).

To deal with this, the system manipulates different nodes of the syntax graph, assigns code to them and updates it as the processing proceeds. For every occurrence of V_G in a semantic statement, reference to the corresponding node is assumed. For this purpose, the system has a semantic stack (SEAS) with two cells in each element: definition cell and the value cell. Nodes are stored in the definition field and their respective codes in the value field.

When a symbol of the source statement is recognised, the node being traversed by the syntax analyser along with the most recently recognised symbol is stacked in the SEAS.

Consider that the syntax analyser traverses a path to "go" from the current node to the one which covers it. In this case if the current node N_m is at top of both SAS and SEAS

a) the SAS-top is deleted.

b) N_{m+1} replaces N_m at the SEAS-top.

Suppose that N_m, N_{m-1}, N_{m-2} ------ N_1 (examples 4.1, 4.3) are the nodes in SAS which match the environment field. In the action and code fields of these semantic statements reference

may be made only to v_Gs corresponding to N_m, N_{m-1}, N_{m-2} ..... N_1. On successfully completing the execution of the action field, the elements of SEAS representing N_{m-1}, N_{m-2}, N_{m-3} ..... N_1 are deleted. Control is then handed back to the syntax analyser. Processing continues until the syntax analyser reaches the start symbol of the grammar. At this stage there may be only one element in the SEAS with the start symbol of the syntax graph in its definition field. The value of this element is generated as code on the output stationary. However, if the definition field of the element in SEAS is other than the start node of the syntax graph, no code is generated from the SEAS, since it is assumed that the user has a separate algorithm for doing so.

A string is formed by concatenating the code generated by executing successive CODE statements. When the statement ":= CODE" is executed, the value of N_m in SEAS is set to the CODE string. If the MTL variable SEAS is set to "0", the semantic stack SEAS does not develop and the user must declare and manipulate his own semantic stacks.

In the example 4.3, when the SAS has the form shown in FIG. 4.10, the SEAS will be as follows:

N_1	<addition exp>1,3	A
N_2	<addition exp>1,2	+
N_3	<addition exp>1,1	A

FIG. 6.4

First of all "A" was recognised and N_3 was the only element stacked in SEAS. It had ⟨addition exp⟩ 1,1 in the definition cell and A in the value cell.

On recognising "+", a new element was created in SEAS, with definition field as ⟨addition exp⟩ 1,2 and the value field as "+". As "A" was recognised, the third element in SEAS was created having ⟨addition exp⟩ 1,3 and "A" in its definition and

the value fields respectively. At this stage (5) is executed. On executing its action field, a string of code A A + is generated. This code is set in the value field of N_3 and the string itself destroyed. On completing the execution, the elements of SEAS representing N1 and N2 are deleted.

6.5 HIERARCHY WITHIN ENVIRONMENT FIELD:-

The environment field of any particular statement is tried from left to right. First of all the symbol "|" outside the scope of brackets "(" and ")" is searched, for successful recognition of its left hand side known as a master alternative results in a successful match of the environment field. In the case of failure, the master alternative on the right of the current one is tried. This master alternative essentially is the environment expression between the above mentioned "|" and the next one on its right, which is out-side the scope of "(" and ")". If however there is no such symbol, the end of the environment field is assumed to have been reached. In any one of the above mentioned cases if there is no "|" outside the scope of "(" and ")", the end of the

environment field is assumed instead. On testing all the master alternatives, if no match is found, failure is reported to the semantic synthesiser in the recognition of the current environment field.

To recognise any one of the master alternatives, an attempt is made to find a relation operator (if any) outside the scope of "(" and ")". Its left hand side is evaluated before the right hand side. On either side of the relation operator all the alternatives are tried, ignoring brackets. However if no alternative matches on any one side of the relation operator, the recognition of the current master alternative is considered to have failed. While evaluating expressions inside brackets "(" and ")" all the alternatives are tried from left to right and the same rules apply as that of master alternatives, except that no further bracketing is expected. This would have been detected at compile time as an error.

example:- 6.1

Consider the environment field

('C' | 'B' | 'A') -⟨b⟩ | (⟨a⟩ | ⟨b⟩ | ⟨c⟩)

which has two master alternatives:

('C' | 'B' | 'A') -⟨b⟩ and (⟨a⟩ | ⟨b⟩ | ⟨c⟩).

The former is tried first and in the event of failure the latter is attempted. To recognise the former master alternative, the position of "-" is determined and then the bracketed alternatives are matched against the source text. If any one of these alternatives matches, the right hand side of "-" is tried, otherwise the second master alternative is attempted.

6.6 PROCESSING OF SOURCE STATEMENTS:-

During the processing of a source statement, every time the control is transferred from the syntax alanyser to the semantic synthesizer, the SEAS is adjusted. The nature of the adjustment depends upon whether this action was taken due to the recognition of a new source language symbol or purning of some part of the parse. A search is then made for a statement in SEGMENT-O, the environment field of which matches the current

environment. The system routine whose job it is to recognise the environment fields of the semantic statements is divided into two parts:

i) which recognises parts (or whole) of the MBNF productions (Production mode).
ii) which recognises all other types of master alternatives in the environment expressions (relation mode)

The semantic synthesizer determines whether or not (i) is applicable, (i) is applicable only if at least the top or the second top element of SAS has the SUCC in its path field.

The environment field of the semantic statements are considered one by one. If the semantics is specified in the production mode (i) is applied otherwise (ii) is considered. If a match is found, its corresponding action field is executed, otherwise the control is returned back to the syntax analyser. During the execution of the action field, if the control is transferred to another segment, the current position is stacked in a system stack. The new segment is then executed exactly in the same manner as that of the SEGMENT-0.

Flow of control for most of the statements used in the action field is straightforward. All the statements are executed sequentially from left to right, except when the control is transferred by a branching statement.

When an ELIMINATE statement is executed, the environment field of the corresponding semantic statement is marked. Marked environment fields play no part in the recognition of environment fields. REINSTATE statements unmark the environment fields.

On completing the execution of a semantic production if the value of STOP is "0", control is handed back to the syntax analyser. For non-zero values of STOP the whole segment is tried again. If the execution of an action field is terminated by branching to the label RETURN, the value of STOP is assumed to be "0".

In MSEAL, there are symbols with more than one meaning depending upon the context in which they are used. These cases are treated separately. For example CODE can either be used as a variable having a string value or as a MSEAL command. In the former case, a SNOBOL4 variable is made

equivalent to it and is referred to at all subsequent occasions. In the latter case, it is treated like any other MSEAL command and is kept in the MSEAL symbol table.

6.7 DATA OBJECTS:-

All data objects, ques, tables and stacks are dynamic and develop in the form of doubly linked lists. The system has a linked list, called "START-LIST", of pointers to a variable number of linked lists which represent the data objects. An element of the START-LIST is known as a descriptor. Its format is as follows.

Definition	Type	Size	Start Pointer	End Pointer	Forward Pointer

FIG. 6.5

DEFINITION:- holds the name of the data object in character form.

TYPE:- coded to indicate whether the data object is a table, que or a stack.

SIZE:- Specifies the number of cells of an element of a data object as it appears to the user. (The two cells required for linking purposes are not included here).

START-POINTER:- pointer to the start of the data object.

END-POINTER:- pointer to the end of the data object.

FORWARD POINTER:-Pointer to the next element in the START-LIST. The last element in the START-LIST has a null string in this cell.

Each element of the START-LIST has pointers to both ends of its data object. Each element of a data object except the end elements, points to and is pointed at by its adjacent elements. The end elements are marked by null string in the pointer field.

When a fresh data object is to be constructed, a new descriptor is created and linked in the START-LIST with appropriately initialised fields. An element of the data object is then created with the required number of cells. These also are initialised appropriately. When a new element is entered in any data object, the start list is first searched for the name of the data object and then after checking its type and the element size, the new element is created at the end of the data object. If however, the name of the data object is not found in the START-LIST, a new data object is created. If the name is found in the START-LIST but its size or type do not match an error message is output.

To search for an element of a data object, the START-LIST is first checked for, the name, type and the element size of the data object. Then using the links of the data object, the particular element is sought. Since all the data objects are created using doubly linked lists and their "starters" in the START-LIST point to both ends, it is possible to make a search starting from either end. The first cell of each element of a data object is considered as the definition cell. If the element is referred to by name rather than

the index, this cell is checked.

EXAMPLE 6.2

Consider the statement

PUSH "ABC", "CD" IN STACK.

The START-LIST is searched to find an element with

DEFINITION = STACK

TYPE = stack

SIZE = 2

If result of the search is "yes", a new element of the stack with "ABC" in the first cell and 'CD' in the second is created. The last element of the stack is linked with it and it is pointed at by the END POINTER of the stack. If search of the START-LIST for an element with "STACK" in the definition field fails, a new starter is created with the following specification:

DEFINITION = STACK

TYPE = stack

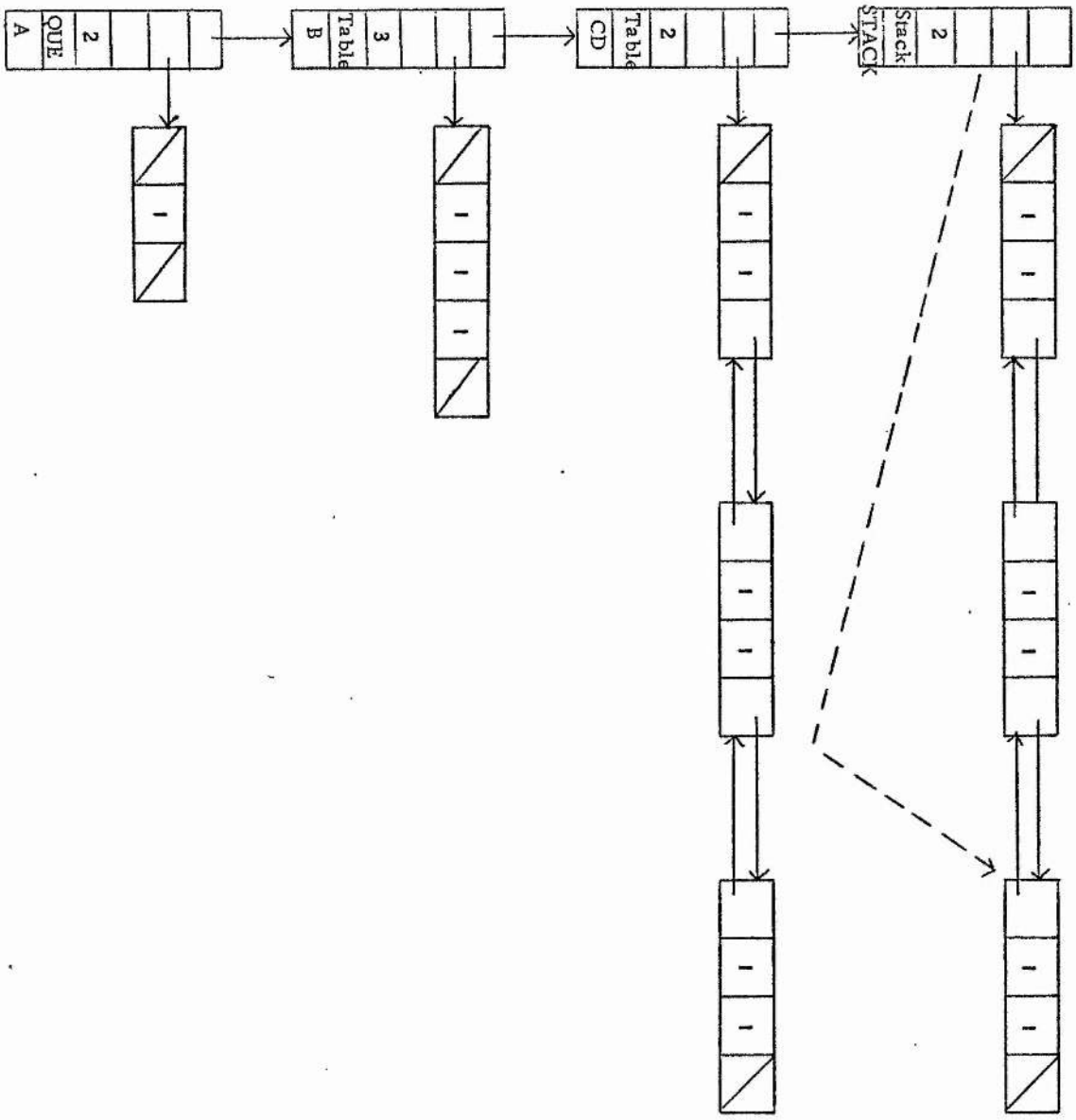
SIZE = 2

The first element of the stack with the above starter has "ABC" and "cd" in its two cells. To retrieve information from one of the data objects, a statement of the following form is executed.

6.7.1 SEARCHDOWN STACK ('ABC'(2))

The system searches the START-LIST for the specifications given in example (6.2), using the END POINTER of this starter, the stack is then searched from top downwards so that it has "ABC" in its first cell (internally 2nd cell). The value of the 2nd cell (internally the 3rd cell) is then returned using VAL.

If the statement (1) is of the form SEARCHDOWN STACK (C(X))
the process of execution is exactly the same except that the values of C and X are assumed.



Internal structure of different data objects.

FIG. 6.6

Table 6.1 is the same as table 5.1 except that in this table systematic conversion of arithmetic expressions to the reverse polish form is displayed. After every call of the semantic synthesiser, the state of the code is displayed.

```
<ADDITION OPERATOR> = PLUS | ADDED TO
<ADDITION FORMULA1> = <NUMBER> ¬ <ADDITION OPERATOR> <NUMBER> ¬N
<ADDITION OPERATOR> <NUMBER>
<ADDITION FORMULA2> = <FUNCTION> ¬ ; <ADDITION OPERATOR>
<FUNCTION> ¬N ; <ADDITION OPERATOR> <FUNCTION>
<MULTIPLICATION OPERATOR> = TIMES | MULTIPLIED BY
<MULTIPLICATION FORMULA1> = <NUMBER> ¬ <MULTIPLICATION OPERATOR>
<NUMBER> ¬N <MULTIPLICATION OPERATOR> <NUMBER>
<MULTIPLICATION FORMULA2> = <FUNCTION> ¬ ; <MULTIPLICATION OPERATOR>
<FUNCTION> ¬N ; <MULTIPLICATION OPERATOR> <FUNCTION>
<ADDITION FORMULA> = <ADDITION FORMULA1> | <ADDITION FORMULA2>
<MULTIPLICATION FORMULA> = <MULTIPLICATION FORMULA1>
 | <MULTIPLICATION FORMULA2>
<MULTIPLICATION FORMULA1>1,1 = <MULTIPLICATION FORMULA2>1,1
<ADDITION FORMULA1>1,3 = <ADDITION FORMULA2>1,4
<ADDITION FORMULA2>1,4 = <ADDITION FORMULA1>1,3
<MULTIPLICATION FORMULA1>1,3 = <MULTIPLICATION FORMULA2>1,4
<MULTIPLICATION FORMULA2>1,4 = <MULTIPLICATION FORMULA1>1,3
<ADDITION FORMULA1>1,5 = <ADDITION FORMULA2>1,7
<ADDITION FORMULA2>1,7 = <ADDITION FORMULA1>1,5
<MULTIPLICATION FORMULA1>1,5 = <MULTIPLICATION FORMULA2>1,7
<MULTIPLICATION FORMULA2>1,7 = <MULTIPLICATION FORMULA1>1,5
<ADDITION FORMULA2>1,3 = <MULTIPLICATION FORMULA2>1,3
<ADDITION FORMULA1>1,1 = <ADDITION FORMULA2>1,1
<ADDITION FORMULA1>1,3 = <ADDITION FORMULA2>1,4
<ADDITION FORMULA2>1,4 = <ADDITION FORMULA1>1,3
<ADDITION FORMULA1>1,2 = <MULTIPLICATION FORMULA1>1,2
<MULTIPLE COMMAND> = SUM | MULTIPLY | PRODUCT | ADD
<SEPARATOR> = , | AND
<DIVISION OPERATOR> = OVER | DIVIDED
<SUBTRACTION OPERATOR> = MINUS
<DIADIC OPERATOR> = <SUBTRACTION OPERATOR> | <DIVISION OPERATOR>
<DIADIC FORMULA1> = <NUMBER> ¬ <DIADIC OPERATOR> <NUMBER>
<DIADIC FORMULA2> = <FUNCTION> ¬ ; <DIADIC OPERATOR> <FUNCTION>
<DIADIC FORMULA> = <DIADIC FORMULA1> | <DIADIC FORMULA2>
<MULTIPLICATION FORMULA2>1,3 = <DIADIC FORMULA2>1,3
<FORMULA> = <ADDITION FORMULA> | <MULTIPLICATION FORMULA>
 | <DIADIC FORMULA>
```

TABLE 6·1

```
<DIADIC BODY> = <FORMULA> <SEPARATOR> <FORMULA>
<DIADIC COMMAND> = DIFFERENCE | QUOTIENT
<DIADIC FUNCTION> = <DIADIC COMMAND> <DIADIC BODY>
<DIADIC BODY>1,3 = <ADDITION FORMULA2>1,3
<MULTIPLICATION FORMULA2>1,3 = <DIADIC FORMULA1>1,2
<MULTIPLICATION FORMULA1>1,2 = <DIADIC FORMULA1>1,2
<BODY> = <FORMULA> <SEPARATOR> <FORMULA> -N <SEPARATOR> <FORMULA>
<MULTIPLE FUNCTION> = <MULTIPLE COMMAND> <BODY>
<FUNCTION> = <MULTIPLE FUNCTION> | <DIADIC FUNCTION> .
<EXPRESSION> = <FORMULA> | <FUNCTION>
<STATEMENT> = <EXPRESSION> .
%SEMANTICS RELATION
*** UNDEFINED NONTERMINALS
NUMBER
';' * 'MINUS' -> DELETE STACK(TOP) & PUSH '-' IN STACK                    :
';' * 'OVER' | ';' * 'DIVIDED' -> DELETE STACK(TOP) &                     -
PUSH '/' IN STACK                                                         :
';' * <MULTIPLICATION OPERATOR> -> DELETE STACK(TOP) &                    -
PUSH '*' IN STACK                                                         :
';' * <ADDITION OPERATOR> ->  DELETE STACK(TOP) & PUSH '+' IN STACK       :
'+' - ',' | '/' - ',' | '-' - ',' | '*' - ',' -> DELETE STACK(TOP)        :
'+' - <SEPARATOR> | '/' - <SEPARATOR> | '-' - <SEPARATOR> |               -
'*' - <SEPARATOR> -> DELETE STACK(TOP)                                    :
'+' - <ADDITION OPERATOR> ->                                              :
<ADDITION OPERATOR>       -> PUSH '+' IN STACK                            :
'*' - <MULTIPLICATION OPERATOR> ->                                        :
<MULTIPLICATION OPERATOR>  -> PUSH '*' IN STACK                           :
'-' - <SUBTRACTION OPERATOR> ->                                           :
<SUBTRACTION OPERATOR> -> PUSH '-' IN STACK                               :
                                                                          '
```

```
'/' - <DIVISION OPERATOR> ->                                                    :
<DIVISION OPERATOR>      -> PUSH '/' IN STACK                                   :
'ADD' | 'SUM' -> PUSH '/+' IN STACK                                             :
'MULTIPLY' | 'PRODUCT'  -> PUSH '/*' IN STACK                                   :
'QUOTIENT' -> PUSH '//' IN STACK                                                :
'DIFFERENCE' -> PUSH '/-' IN STACK                                              :
'ADD' * <NUMBER> | 'SUM' * <NUMBER> | 'MULTIPLY' * <NUMBER> |                    -
'DIFFERENCE' * <NUMBER> | 'QUOTIENT' * <NUMBER> |                                -
'PRODUCT' * <NUMBER> -> CODE SYMBNOW                                            :
'/+' - <NUMBER>          -> CODE SYMBNOW & CODE '+'                             :
'/*' - <NUMBER>          -> CODE SYMBNOW & CODE '*'                             :
'/-' - <NUMBER>          -> CODE SYMBNOW & CODE '-'                             :
'//' - <NUMBER>          -> CODE SYMBNOW & CODE '/'                             :
'+' - <NUMBER> | '*' - <NUMBER> | '-' - <NUMBER> |                               -
'/' - <NUMBER> -> CODE SYMBNOW  & CODE STACK(TOP)                               :
<NUMBER>  -> CODE SYMBNOW                                                       :
%GENERATE
%SNOBOL
 DEFINE('NUMBER()','Z7') :(Z8)
Z7 CARD LEN(*INPOS) BREAK(' ') . ZVAL
 CONVERT(ZVAL,'REAL') :S(Z1)
 OBSTACLE = 1
 MATCHED = 0 :(RETURN)
Z1 MATCHED = 1
 OBSTACLE = 0 :(RETURN)
Z8 &ANCHOR = 1
%FINISH
ADD 5 , 6 .

5
5
5 6 +
5 6 +
MULTIPLY 6 AND 9 .

6
6
6 9 *
6 9 *
   ADD 5 , 456 .

5
5
5 456 +
5 456 +
ADD 4 2

4
```

```
          £
*** THIS SYMBOL = AND OR ITS EQUIVALENT
ADD 5 , 6 , 7 , 6 .

5
5
5 6 +
5 6 +
5 6 + 7 +
5 6 + 7 +
5 6 + 7 + 6 +
5 6 + 7 + 6 +
ADD 5 PLUS 6 PLUS 9 , 9 PLUS 2 .

5
5
5 6 +
5 6 +
5 6 + 9 +
5 6 + 9 +
5 6 + 9 + 9 +
5 6 + 9 + 9 +
5 6 + 9 + 9 + 2 +
5 6 + 9 + 9 + 2 +
MULTIPLY 5 PLUS 8 , 6 , 3 .

5
5
5 8 +
5 8 +
5 8 + 6 *
5 8 + 6 *
5 8 + 6 * 3 *
5 8 + 6 * 3 *
ADD 4 , 4 .+

4
4
4 4 +
          £
*** THIS SYMBOL = . OR ITS EQUIVALENT
5 PLUS 6 .
5
5
5 6 +
5 6 +
5 PLUS 6 PLUS 7 PLUS 8 .
5
5
5 6 +
5 6 +
5 6 + 7 +
5 6 + 7 +
5 6 + 7 + 8 +
5 6 + 7 + 8 +
MULTIPLY SUM 5 , 6 , 7 ; AND 8 .
```

```
5
5
5 6 +
5 6 +
5 6 + 7 +
5 6 + 7 +
                                £
*** ERROR FOUND
PRODUCT 5 , 6 , 7 . PLUS 8 .

5
5
5 6 *
5 6 *
5 6 * 7 *
5 6 * 7 *
                              £
*** THIS SYMBOL = NUMBER OR ITS EQUIVALENT
PRODUCT 5 , 6 , 7 ; PLUS 8 .

5
5
5 6 *
5 6 *
5 6 * 7 *
5 6 * 7 *
5 6 * 7 *
```

Table 6.2 shows different stages of processing assignment statements with the specification of table 5.2. At different stages, when the semantic synthesiser is called, a table of information stating the states of the SAS and the SEAS is displayed along with the current cursor position on the source language statement. The node fields show which node is stacked in the SAS or the SEAS. POS is the current cursor position and PATH is the path traversed after stacking the said node. In SEAS, the CODE field gives the current value of the node in terms of the code.

As it can be seen from different tables, it is not necessary that every time the semantic synthesiser is called, some semantic production must be executed. If no semantic production is recognised the control is returned back after updating the SEAS. For example, although no semantic production was recognised in table 6.21, ALPHANUMERIC NODE is stacked in the node.

and X in the code field.

table 6.22, := NODE is stacked in the node field

and := in the code fields of SEAS.

In 6.23 however two actions took place. The node field of the topmost element was changed from ALPHANUMERIC $_{NODE}$ to $\langle$variable$\rangle_{NODE}$ Then a new element was stacked with $+_{NOE}$ and +.

Similar is the case at other stages.

In table 6.24 N_{11} to N_1 form the parse tree given on page 188.

```
<VARIABLE> = ALPHANUMERIC
<ADDING OPERATOR> = + | -
<MULTIPLYING OPERATOR> = * | /
<OPERAND> = ( <ARITHMETIC EXP> ) | <VARIABLE>
<ADDING EXP> = <OPERAND> ¬N <ADDING OPERATOR> <OPERAND>
<MULTIPLYING EXP> = <OPERAND> ¬N <MULTIPLYING OPERATOR> <OPERAND>
<ARITHMETIC EXP> = <ADDING EXP> | <MULTIPLYING EXP>
<ASSIGNMENT STATEMENT> = <VARIABLE> := <ARITHMETIC EXP>
<MULTIPLYING EXP>1,2 = <ADDING EXP>1,2
<ADDING EXP>1,2 = <MULTIPLYING EXP>1,2
%SEMANTICS PRODUCTION
<OPERAND>1,1;2;3 -> Z := NULL &                                          -
Z1 := IND SEARCH SEAS(<OPERAND>1,1) & Z1 := IND NEXTUP SEAS(Z1) &        -
CODE SEAS(Z1(2)) &LABEL1 Z1 := IND NEXTUP SEAS(Z1) &                     -
?S (LABEL3) &LABEL4 CODE Z & := CODE & CODE := NULL & (RETURN)           -
&LABEL3    EQ ( SEAS(Z1(1)) , <ADDING EXP>1,2 ) & ?S (LABEL2)            -
& EQ ( SEAS(Z1(1)) , <OPERAND>1,3 ) & ?S (LABEL4)                        -
& CODE SEAS(Z1(2)) & (LABEL1) &LABEL2 CODE Z &                           -
Z := SEAS(Z1(2)) & (LABEL1)  :

DUMMY <MULTIPLYING EXP>1,2;3 -> CODE <MULTIPLYING EXP>1,3                -
& CODE <MULTIPLYING EXP>1,2  & := CODE & CODE := NULL  :

<ASSIGNMENT STATEMENT>1,1;2;3 -> Z := NULL &                             -
CODE <ASSIGNMENT STATEMENT>1,3 & ?F (LABEL5) &LABEL6 CODE Z &            -
 CODE <ASSIGNMENT STATEMENT>1,1 & CODE <ASSIGNMENT STATEMENT>1,2         -
& := CODE & CODE := NULL & (RETURN)                                      -
&LABEL5 Z := NULL & Z1 := IND SEARCH SEAS(<ADDING EXP>1,1) &             -
CODE SEAS(Z1(2)) &LABEL1 Z1 := IND NEXTUP SEAS(Z1) &                     -
?S (LABEL3) & (LABEL6)                                                   -
&LABEL3    EQ ( SEAS(Z1(1)) , <ADDING EXP>1,2 ) & ?S (LABEL2)            -
& CODE SEAS(Z1(2)) & (LABEL1) &LABEL2  CODE Z &                          -
Z := SEAS(Z1(2)) & (LABEL1)    :


%GENERATE

X := A + B * C
```

TABLE 6.2

```
**********************************************************

                SAS                SEAS

**********************************************************

      NODE          POS PATH              NODE          CODE

<VARIABLE>            1  DAU  |     ALPHANUMERIC          X
<ASSIGNMENT STATEMENT>


************************

CURRENT CURSOR POSITION


X := ( A + B ) * C
 |
```

TABLE 6·21

```
**************************************************************

                    SAS              SEAS

**************************************************************

      NODE          POS PATH              NODE          CODE


                          |               :=              :=
<VARIABLE>            I   SUCC    |       <VARIABLE>      X
<ASSIGNMENT STATEMENT>


************************

CURRENT CURSOR POSITION


X := ( A + B ) * C
     |
```

TABL 6.22

```
**********************************************************************

                    SAS                 SEAS

**********************************************************************

      NODE          POS PATH                NODE          CODE


<VARIABLE>           6  DAU   |
<OPERAND>            6  DAU   |
<ADDING EXP>         6  DAU   |
<ARITHMETIC EXP>     6  DAU   |     ALPHANUMERIC          A
:=                   2  SUCC  |     :=                    :=
<VARIABLE>           1  SUCC  |     <VARIABLE>            X
<ASSIGNMENT STATEMENT>


**********************

CURRENT CURSOR POSITION

.

X := A + B * C
     |
```

TABLE 6·23(a)

TABLE 6.23(b)

```
**********************************************************************

                    SAS                 SEAS

**********************************************************************

      NODE          POS PATH                    NODE          CODE


<ADDING OPERATOR>     8  DAU   |
<OPERAND>             6  SUCC  |
<ADDING EXP>          6  DAU   |      +                        +
<ARITHMETIC EXP>      6  DAU   |      <OPERAND>                A
:=                    2  SUCC  |      :=                       :=
<VARIABLE>            1  SUCC  |      <VARIABLE>               X
<ASSIGNMENT STATENENT>


************************

CURRENT CURSOR POSITION


X := A + B * C
     ^
     1
```

SAS SEAS

N	NODE	POS	PATH		NODE	CODE
1	<OPERAND>	14	DAU	|		
2	<MULTIPLYING OPERATOR>	12	SUCC	|		
3	<ADDING OPERATOR>	12	MOD	|		
4	<OPERAND>	13	SUCC	|	ALPHANUMERIC	C
5	<ADDING OPERATOR>	8	SUCC	|	<MULTIPLYING OPERATOR>	*
6	<OPERAND>	6	SUCC	|	<OPERAND>	B
7	<ADDING EXP>	6	DAU	|	<ADDING OPERATOR>	+
8	<ARITHMETIC EXP>	6	DAU	|	<OPERAND>	A
9	:=	2	SUCC	|	:=	:=
10	<VARIABLE>	1	SUCC	|	<VARIABLE>	X
11	<ASSIGNMENT STATEMENT>					

****

CURRENT CURSOR POSITION

X := A + B * C
THERE IS NO MORE SYMBOL IN THE SOURCE STATEMENT

TABLE 6.4

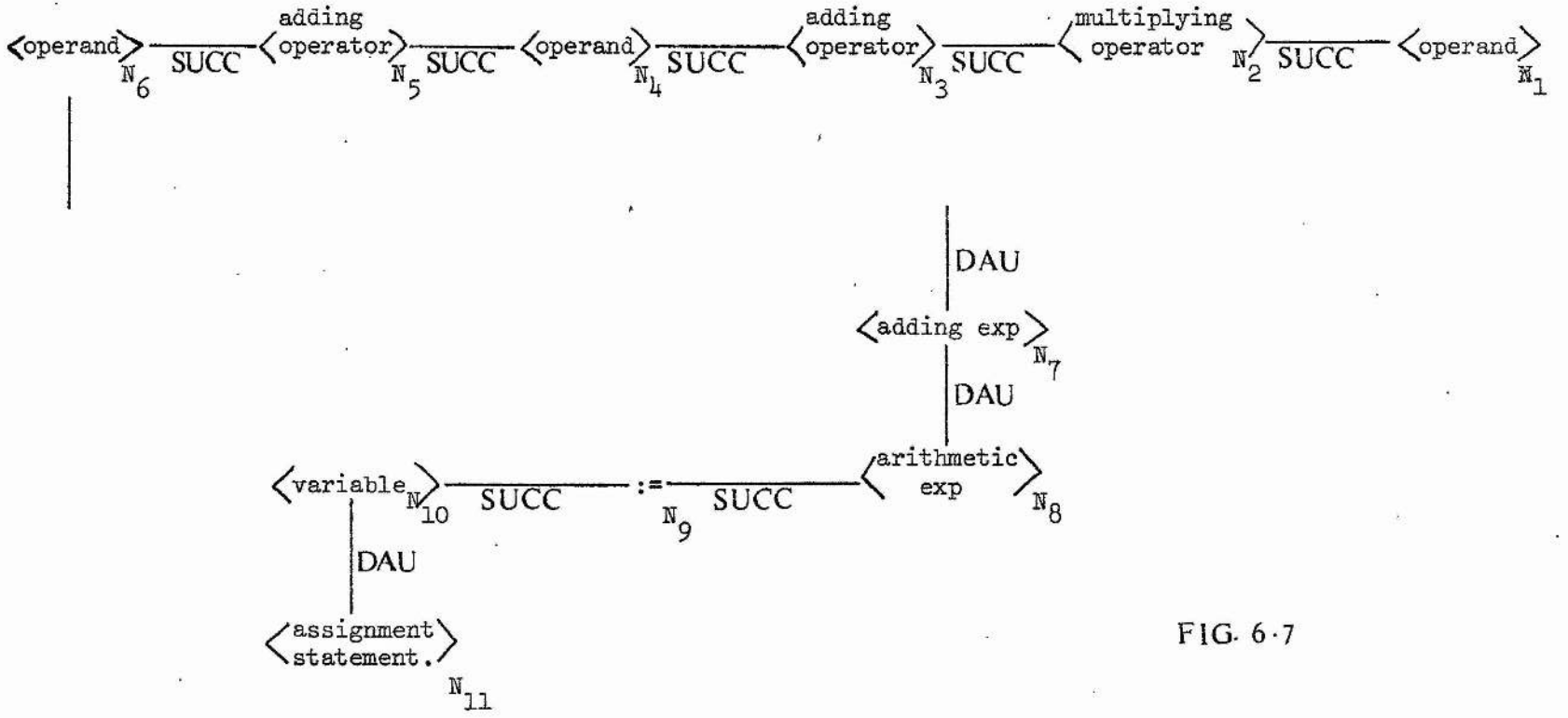

<operand> N6
SUCC
adding operator N5
SUCC
<operand> N4
SUCC
adding operator N3
SUCC
multiplying operator N2
SUCC
<operand> N1
DAU
<adding exp> N7
DAU
<variable> N10
SUCC
:= N9
SUCC
arithmetic exp N8
DAU
<assignment statement.> N11

FIG. 6·7

To recognise a semantic production at the current state, we are only concerned with N_6 to N_1. As a matter of fact N_3 is redundant since it plays no part in parsing except to modify the path. Hence we concern ourselves with the following tree structure.

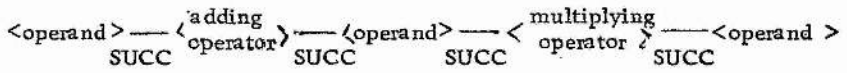


FIG. 6.8

This matches the seoond semantic production in table 6.2, Since

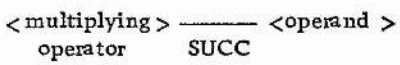


FIG. 6.9

is the same as ⟨multiplying exp⟩ 1, 2; 3 and the DUMMY by definition matches all the nodes on their left.

CHAPTER 7

EXTENSIBILITY IN MTL.

7.1 EXTENSIBILITY:-

A programming language must provide the user with adequate means of expressing an algorithm in a manner which matches the problem he wants to solve. There are two approaches to achieve this end: either to have a large universal language or a large number of problem oriented languages. For a large universal language, features must be provided for many diverse areas e.g. numerical analysis, compiler writing, list processing etc. The translator for a universal language will unavoidably be large and hence may not be usable on small machines. Because of its size the translator is difficult to write, maintain and perhaps relatively inefficient. Another difficulty with the design of a universal language is that the continuous need for revision of already existing computer languages, has proved that it is impossible to visualise all the needs of the prospective users and also of the usefulness and shortcommings of different features of the language.

Once the language is designed, one might be so committed to it that any modification requires a major change in the design or else inefficiency and inconsistancy is the result.

The idea of having special-purpose languages has a slight advantage over the above mentioned approach although most problems remain or are merely replaced by similar ones. The need for the maintenance of their translators means that a large part of the systems-programming effort at a given establishment may eventually be taken up in ensuring that a large number of languages are available. Different implementations will have different designs and hence pose a greater problem for maintenance of software and advisory support for its users. Introduction of new features in a problem oriented language is no simpler. However the languages are more likely to be suited to the user's requirements and probably more suitable for his problem.

Another alternative approach is the design of an extensible language which starts off with a few features, but which can be extended by the user. The extensibility can be of two forms : syntactic and semantic. The term syntactic extensibility

is used to indicate the possibility of extending a language by means of a program written in the same language. The concept is similar to that introducing procedures in ALGOL 60, except that the syntactic form of call of a procedure is fixed whereas in some syntactically extensible languages like IMP [Irons.E.T. 1970] it is separately specified in each declaration. The possibility of introducing new concepts in the language (by modifying its basic implementation) may be called semantic extensibility.

Extensible languages (specially semantically extensible languages) can be seen to have the same effects as that of several problem oriented languages since different extended versions of an extensible language can be considered as different problem oriented languages. Maintenance is relatively simpler since the design approach remains unaltered. It is however wrong to say that an extensible language is a perfect solution to the system-software problems. Most existing extensible languages put some constraints on the type of features which may be introduced as an extension. The size of some such languages is fairly big. In the case of semantically extensible languages, a high degree of programming skill and knowledge concerning the language implementation is required

to make extensions. Moreover a survey of extensible languages [SOLNTSEFF & YEZERSKI 74] shows that at present there is a great diversity of approaches taken by different workers in the field and there is no apparent agreement as to what constitutes an extensible language. Since very little material is available on the experience with extensible languages, it is rather difficult to comment confidently on their different aspects to a user. As Irons' [IRONS 70] experience shows, the problem of diagnostics and ambiguities due to extensions should be taken seriously. In our case, the arguments in favour of semantic extensibility can also be derived from chapters 3 and 5. We shall rely on this feature for introducing the desired no-backup parsing, to control lexical scanning and to provide power for MSEAL.

7.2 MTL AND EXTENSIBILITY:-

The MTL is semantically extensible. New features can be introduced in it and even the whole of SNOBOL4 can be considered as a subset of MTL, since different SNOBOL4 programs can be used to perform tasks which are not conveniently performed

by the MTL. As many SNOBOL4 programs can be introduced as the user wishes and at any place in the MTL definition. Each program is enclosed in bracket "%SNOBOL" and "%FINISH". The SNOBOL4 "END" statement is required only if a transfer to it is to be made (in which case the execution stops completely and the job is terminated). All programs are handled automatically by the processor and are linked at the appropriate place. To facilitate communication between the MTL and its extension routines in SNOBOL4, different system variables are provided. Since the processor is implemented in SPITBOL, the version of SNOBOL4 available for extension is as described in the SPITBOL manual. Some restrictions however have been put on its use. Slightly different extension mechanisms have been provided in MSYL and MSEAL, the reason for which will become clear in the latter part of this chapter.

Before we go any further we shall take an overview of the SNOBOL4 facilities available for extensibility. Each extension of MTL must be a valid SNOBOL4 program rather than a mere user defined function. No global variables, user defined function names or labels may start with the letter "A". Internally

all SNOBOL4 programs forming extensions are linked together and treated as a single entity. It is therefore essential that conflicts of labels, variables and function names within various extensions be avoided. It is also possible to define a data object in one extension and access it in another. Since the extension programs are handled by the MTL processor, no facility concerning system data sets and JCL is available or necessary.

Optional SPITBOL control cards -NOCODE, -CODE, -OPTIMISE and -NOOPTIMISE are also not available. They may be used but are ignored by the system. The system variable EDITOUT can have a system unit as its value. The edited version of the SNOBOL4 program statements appearing after it is listed. This listing includes the above mentioned ignored control cards. When the extension programs are completely debugged, the edited code is placed at the beginning of the MTL processor source which is then recompiled. During this compilation all control cards specified in the extension programs are in effect within their scope. No control card is ignored. The new code is now the extended version of the MTL processor. The aim of extensibility in MSYL are two fold:

a) to allow lexical scanning;

b) to introduce look ahead for avoiding backup.

The former is important in automatic syntax analysers to allow for exceptions to the general rules in MBNF, for instance to distinguish key words from identifiers etc. Moreover it is faster than a complete top down recognition process. The second reason is important since ours is an ELL(k) parser and look ahead needs to be introduced. Since the rest of the syntax analysis is done automatically, the only type of extensibility required is to be able to invoke some SNOBOL4 extension program on meeting a certain symbol of MBNF. All undefined MBNF nonterminals are considered as names of the SNOBOL4 user defined functions (with no parameters). The user is assumed to have defined them in his SNOBOL4 programs. When such a nonterminal is processed by the system, its corresponding SNOBOL4 function is called. On returning from the function the processing proceeds as usual. The source language string appears in the SNOBOL4 extension program as the value of the MTL system variable CARD. The user is allowed to read more cards and control the listing by himself or alternatively to concatenate them at the end of the already

existing value of CARD. In this case the listing is controlled by the processor, INPOS and FINPOS are the initial and final positions of the symbol under consideration in the string CARD. It is the users responsibility to set appropriate values of these system variables when returning control back to the system. System variables OBSTACLE and MATCHED are used to inform the system about the result of lexical scanning or look ahead performed in any SNOBOL4 extension. On entering a SNOBOL4 extension, their values by default are "0" and "1" respectively. On returning to the system if the value of OBSTACLE is "1", it indicates that the current production is not to be followed any longer. If the value of MATCHED is "0", it means that, the attempt to recognise the current symbol has failed. STACK is a system defined stack which is accessible both in MSYL and SNOBOL4. In SNOBOL4 it appears as a one dimensional array STACK with 80 cells. The aim of extensibility in MSEAL is different from that in MSYL. In the current version of MSEAL, the emphasis is towards generating the intermediate language and it is not geared towards providing full facilities of a systems language. The extensibility provided allows the use of SNOBOL4 for the implementation of such features as

are required. The new facilities introduced will then form part of MSEAL and can be used for later purposes.

There are two methods of introducing extensions in MSEAL, both appearing only in the action field of a semantic statement. In both cases an extension must be an independent statement.

The first of the two methods is similar to the method of extending MSYL. When an undefined nonterminal appears as an independent statement, the corresponding SNOBOL4 user defined function is called and on returning from the function, processing continues as usual.

The alternative method of extending MSEAL is to include SNOBOL4 programs in successive brackets of %EXTENSION and %PAUSE; the end of this sequence of brackets being marked by %FINISH. The name of each program follows its corresponding %EXTENSION command.

During the execution of an action field in MSEAL, when a program name is executed as a statement, its corresponding bracketed program is activated. When a SNOBOL4 extension is activated, the MTL

system variable CARD refers to the whole of the action field under consideration. INPOS and FINPOS are pointers to the initial and final positions of the symbol under consideration. Using these pointers, the later part of the statement (if any) can be scanned. The MTL system variable TEST by default has value "S". Its value can be changed to "F" to inform the system that the current statement has failed. The value of ACTION can be a string of the form SNOVAR = ⟨identifier⟩ or ⟨identifier⟩ = SNOVAR. In the former case value of the ⟨identifier⟩ in MSEAL is assigned to SNOVAR, while in the latter case the opposite is true.

7.3 IMPLEMENTATION OF EXTENSION PROGRAMS:-

We start by describing some of the important concepts of SNOBOL4 and then in steps we shall explain the whole process of extension.

During the execution of a SNOBOL4 program, it is possible, by using the primitive function CODE (STRING) to convert a string of characters into object code. We shall refer to this process as run time compilation. The effect of the function

CODE is to convert its argument STRING in_to the object code. The string must represent a valid SNOBOL4 program complete with labels and using ";" to separate statements. Blanks are as important in strings to be converted to code as they are in program itself. A statement without a label must begin with a blank. For example the variable COMPILE can have a string value assigned by the following statement.

```
7.3.1 COMPILE  =  'START   &TRIM  =  1  ;'
    +             '            N  =  10  ;'
    +             '         LINE  =      ;'
    +             'LOOP        N  =  GT(N,0)  N-1  :F(LAB) ;'
    +             '         LINE  =  LINE  INPUT     :(LOOP);'
    +             'LAB    OUTPUT  =  "ACTION COMPLETE" '
    +             '                  :(LABEL)'
```

This string can be compiled at run time by executing the following statement.

7.3.2 CODE (COMPILE)

After compilation, one way of executing this program is by transferring control to START i.e. by executing the statement

7.3.3 :(START)

The following statements will perform all the three tasks described in (1), (2) and (3).

```
7.3.4     COMPILE   =  'START    &TRIM  =  1  ;'
    +                 '              N  =  10 ;'
    +                 '           LINE  =     ;'
    +                 'LOOP          N  =  GT(N,0) N - 1 :F(LAB);'
    +                 '           LINE  =  LINE INPUT  :(LOOP);'
    +                 'LAB       OUTPUT =  "ACTION COMPLETE"
    +                 '                                :(LABEL)'
                      CODE (COMPILE)                   :(START)
```

It is necessary to transfer control to the first statement of the newly compiled program, in order to make sure that it runs. It is also essential to jump out of it, otherwise the control is automatically transferred to the label END. It is due to this reason that :(LABEL) has been introduced in the above program. In order to make this statement semantically correct, LABEL must appear somewhere in the main program. We shall denote (4) by

7.3.5 START Program :(LABEL)

The philosophy behind the extensibility in MTL is that at execution time the MTL processor

a) given a SNOBOL4 program, converts it into a valid string and compiles it into the form (5)

b) when required for extension purposes, it branches to START and after completing the execution branches back to LABEL.

The method described above is adequate, provided, all the SNOBOL4 statements are syntactically correct, but how to detect errors if there are any? The answer lies in the fact that the SNOBOL4 function CODE fails if there is any syntactic error. Using this fail condition, the key word &ERRLIMIT and the function SETEXIT, diagnostics can be realised. However if the whole SNOBOL4 program is treated as a single string, only one failure will occur and statement by statement diagnostics will not be possible. To overcome this problem, each SNOBOL4 statement which need be separated from the other by a semicolon is treated as a separate program, edited as such and compiled in the form (5). Each statement of (1) will therefore be edited into the following form and compiled individually.

```
7.3.6 COMPILE  =  'START    &TRIM = 1     :(A501)'
      COMPILE  =  'A501        N = 10    :(A502)'
      COMPILE  =  'A502       LINE =     :(LOOP)'
      COMPILE  =  'LOOP          N = GT(N,0) N - 1
                                 :F(LAB)S(A503)'
      COMPILE  =  'A503       LINE = LINE INPUT  :(LOOP)'
      COMPILE  =  'LAB      OUTPUT = "ACTION COMPLETE"  :(LABEL)
```

It is not difficult to see that the compiled version of (1) and (6) will have similar semantic effects. However (6) can be developed on the same lines as that of (3), (4) and (5). Now we shall modify (5) to be written as

```
7.3.7   START     Σ program     :(LABEL)
```

In MTL, SNOBOL4 extension programs need be executed immediately after compilation. It is therefore necessary to compile and execute the following statement immediately after (7).

```
    COMPILE  =  '    :(START)'
```

Since the extension programs are to be linked with the MTL processor which itself is written in SNOBOL4, various checks need be made and actions taken to avoid conflicts. We will discuss them briefly since we assume that the reasoning behind them is relatively easy to follow.

Different effects can be had in a SNOBOL4 program by setting the values of &ANCHOR, &TRIM and &FULLSCAN. As far as the linking process is concerned, the MTL processor works with the following specifications.

7.3.8

```
&ANCHOR  =  1
&TRIM    =  0
&FULLSCAN =  0
```

It is thus necessary that (8) should be executed every time control is passed back to the MTL processor from an extension program, in case these specifications have been changed. This is partly embedded in the system and partly achieved at the time of generating code. While developing (7) for an extension program:

a) comment cards are ignored,

b) The SNOBOL4 control cards -CODE, -NCODE, -COPY, -FAIL, -NOFAIL, -OPTIMISE or -NOOPTIMISE are ignored.

c) For all SNOBOL4 control cards other than described in (b) no code is compiled but appropriate action is taken by the MTL processor.

d) Only the leftmost 72 columns of a card are considered.

e) Before compilation, statements continuing on more than one card are concatenated so as to make a single string. In this case

the continuation symbol '+' is removed.

f) All SNOBOL4 statements including the original processor and internally generated statements are counted for listing. This is to make sure that the error messages are given with correct statement numbers.

g) Duplication of labels is checked.

h) It is checked that no label, identifier or a function name starts with letter "A".

The name following the %EXTENSION command is considered as the name of a user defined function. A statement is generated and compiled for defining a function with this name, having no formal parameters or local variables. The next SNOBOL4 statement is considered as the first statement of the function and flow of control is arranged for this. When a %PAUSE or %FINISH command corresponding to a %EXTENSION command is met, the end of a particular function is assumed and a go to field (RETURN) is generated and compiled such that after execution of the function, control is always transferred to the label RETURN. Considering that the %EXTENSION PRINT appears immediately before a SNOBOL4 program which is edited into (6). Then the edited version will be as follows

```
7.3.9 COMPILE = 'START  DEFINE ("PRINT()" , "A501" ) :(A505)'
      COMPILE = 'A501   &TRIM = 1                    :(A502)'
      COMPILE = 'A502   N     = 10                   :(A503)'
      COMPILE = 'A503   LINE  =                      :(LOOP)'
      COMPILE = 'LOOP   N     = GT(N,0) N - 1        :F(LABEL)S
                                                      (A504)'
      COMPILE = 'A504   LINE  = LINE INPUT           :(LOOP)'
      COMPILE = 'LABEL  OUTPUT = "ACTION COMPLETE"   :(RETURN)'
      COMPILE = 'A505                                :(LABEL)'
```

On encountering an ACTION statement, statements are generated and compiled to call an appropriate system defined function to perform the action specified. When the command %EDITOUT is specified in any SNOBOL4 extension program the effect is as follows.

Actions (a) to (h) specified above are not carried out. Moreover the statements

```
'      :(LABEL)'
'      :(START)'
```

are not generated at the end of the program, instead the statements specified in (8) are generated. SNOBOL4 statements are generated to initialise tables of labels, function names and to account for the total number of statements generated. All these SNOBOL4 statements are displayed on the output stationary.

When the user adds these newly generated statements to the processor, it looks as follows:

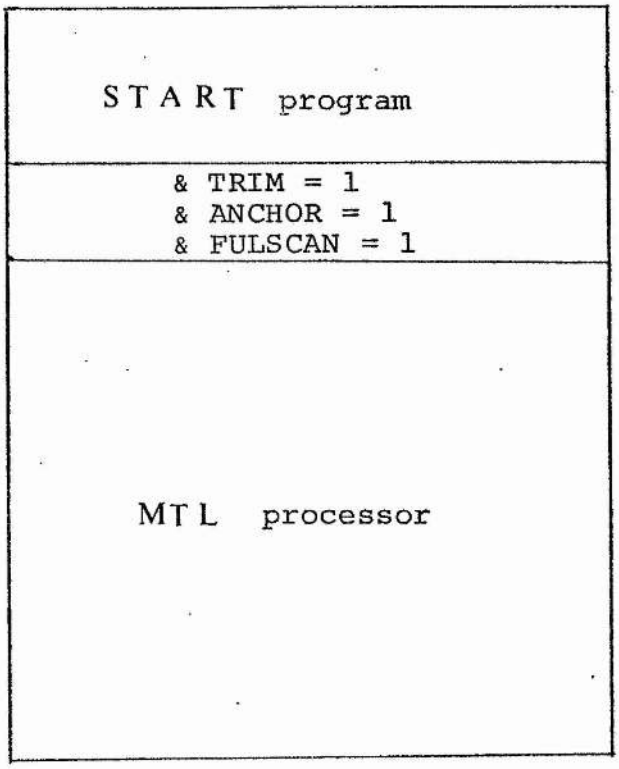


FIG. 7.1

The extension statements now form part of the MTL processor and the initialisation of the key words described above helps to maintain the correct mode of the MTL processor.

7.4 LINKING EXTENSION PROGRAMS WITH THE MTL PROCESSOR:-

During the construction of the syntax graph, the system uses a symbol table. All nonterminals are entered in this table. The entries for undefined nonterminals are marked. At the completion of the syntax graph, the system scans through the whole of the symbol table to look for the marked entries and generates appropriate code for linking user defined SNOBOL4 functions. Moreover the syntax graph is modified so that the marked nonterminals are treated as terminals and it is possible to link them with the user defined SNOBOL4 functions.

During the normal process of syntax analysis, when a terminal is processed, control is passed from the MSYL SCHEDULER to the MSYL MATCH-BLOCK of the processor. In the MATCH-BLOCK, it is attempted to match the current terminal with the current symbol of the source statement. The control is then passed back to the SCHEDULER with the appropriate signal for whether or not the match was successful.

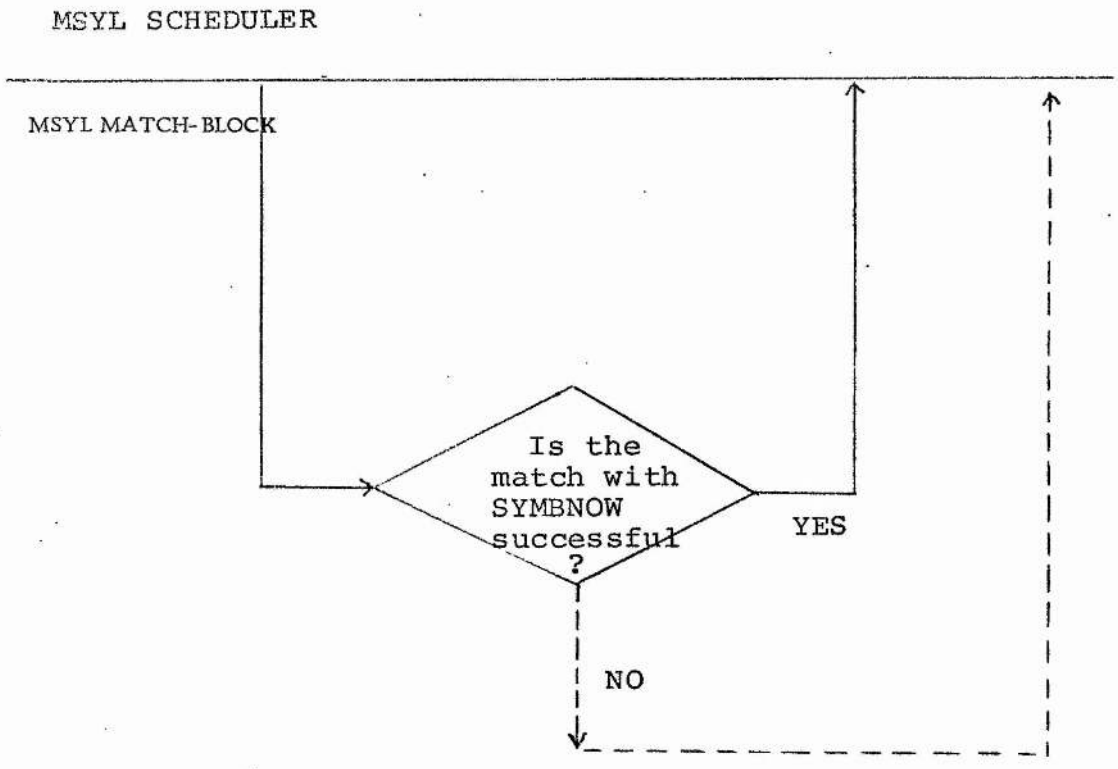


FIG.7.2

The failure path is kept open at the time of writing the processor. On completing the syntax graph if there is no undefined nonterminal in the symbol table this path is completed as shown in the above diagram. For this purpose, SNOBOL4 code is generated internally in the form of a string and is compiled using the SNOBOL4 function CODE. On the other hand, on scanning the symbol table if the system detects some undefined nonterminals, a warning message is displayed on the output stationary. The symbol table entries are unmarked. The syntax graph is modified such that the node under consideration is treated as a terminal rather than a nonterminal.

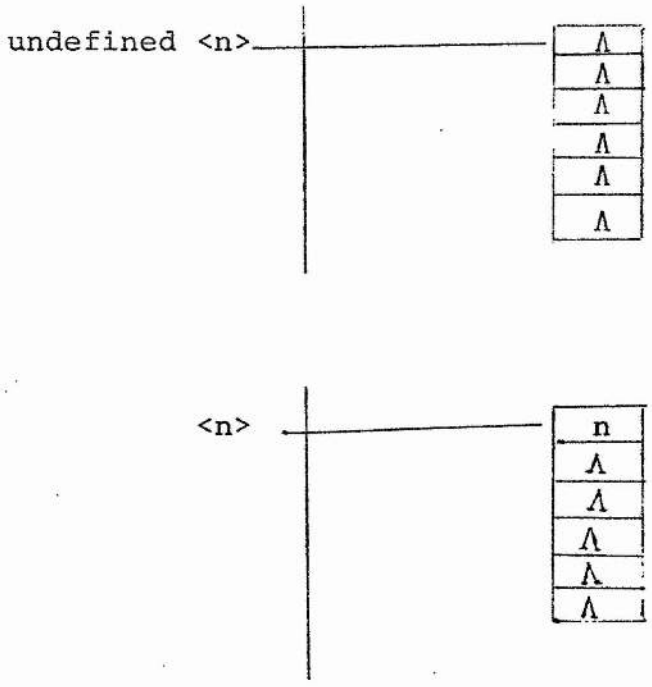


FIG. 7.3

Moreover the appropriate number of SNOBOL4 statements are generated internally in the form of a string, complete with semicolons, separating them from one another. The statements perform the following tasks.

a) Find out if the current terminal represents an undefined nonterminal.

b) If (a) is true, call the appropriate function and on returning from the function pass the control back to the MSYL SCHEDULER.

c) If (a) is false, return control to the SCHEDULER with a signal of failure.

d) Both in (b) and (c), on returning from a function, before executing any other statement in the SCHEDULER, the following statements must be executed.

&TRIM = 0; &ANCHOR = 1; &FULLSCAN = 0

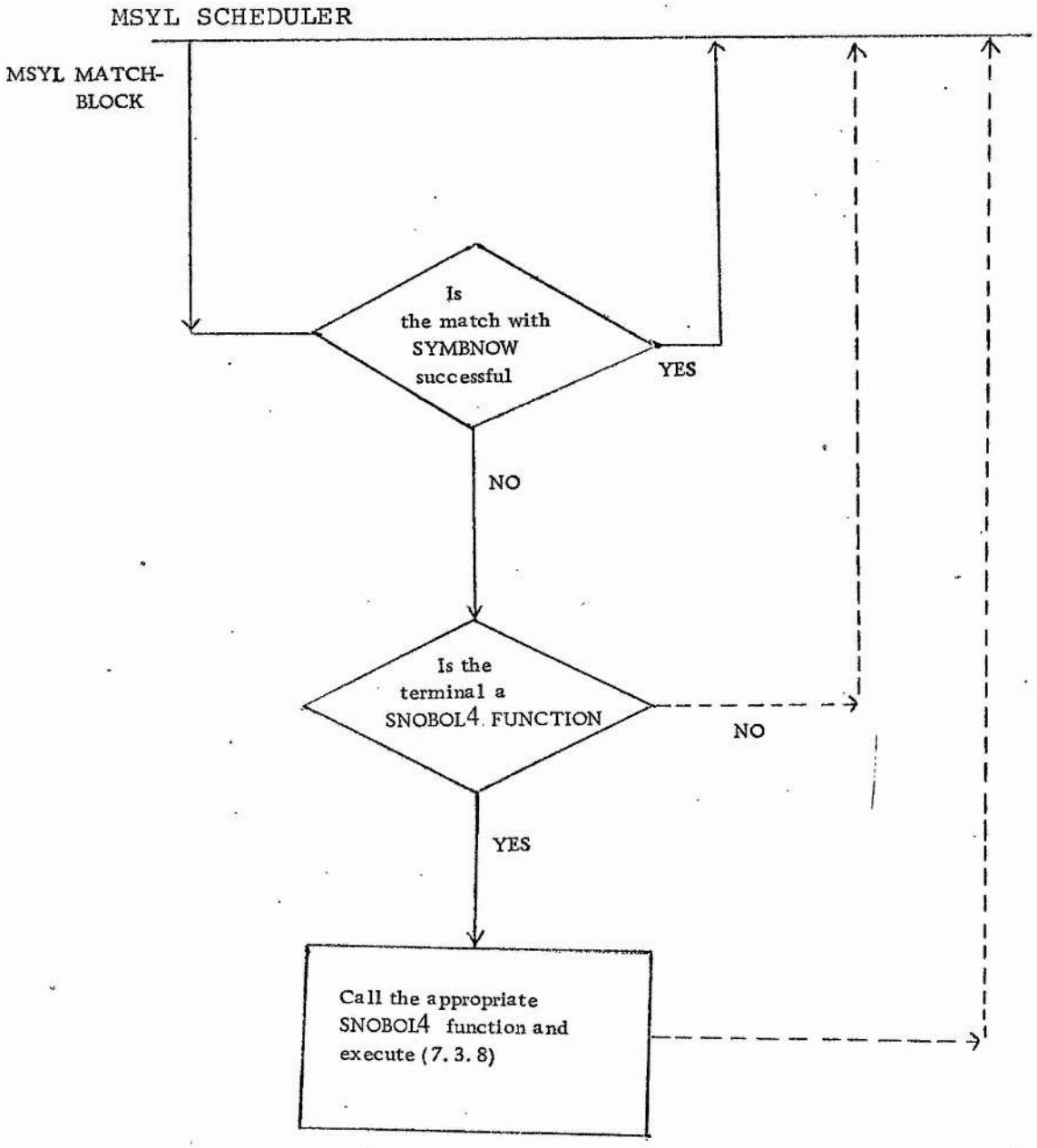


FIG. 7.4

As can be seen from the diagrams given below, the basic principle underlying the linking mechanism for both methods of MSEAL extension is similar to the one for MSYL extension. The difference being that MSEAL SCHEDULER and MSEAL MATCH-BLOCK respectively are considered at the place of MSYL SCHEDULER and MSYL MATCH-BLOCK. Moreover the point of extension is determined in different pattern .

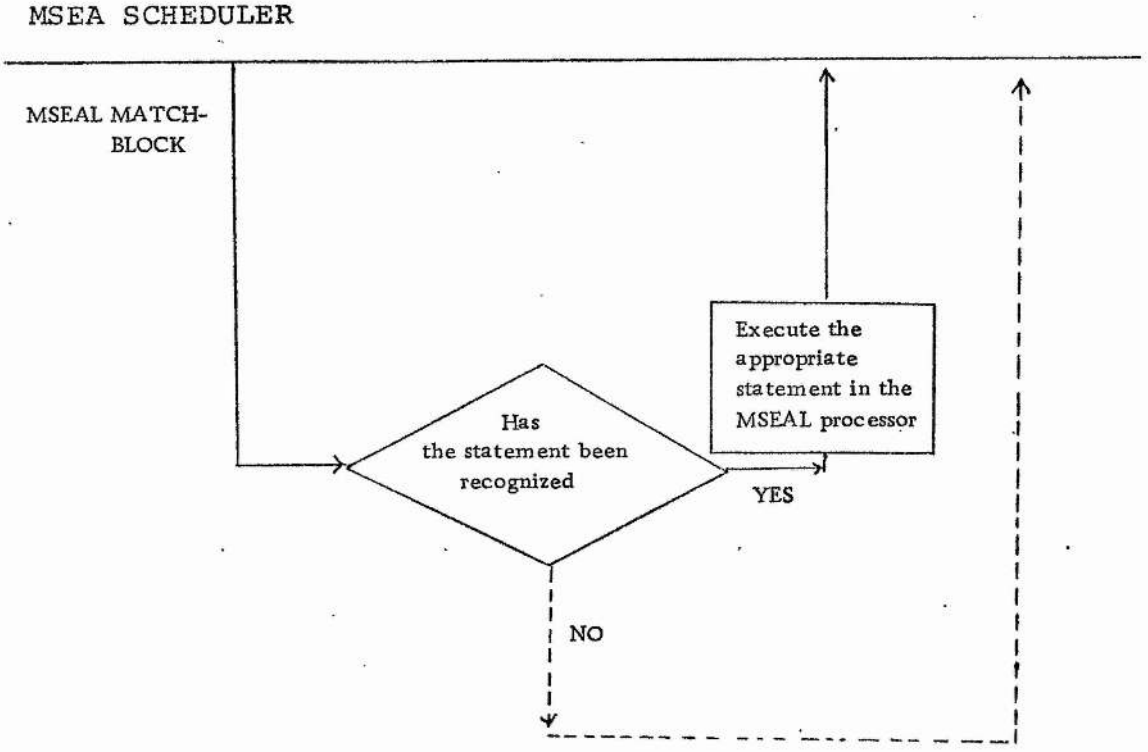


FIG. 7.5

If a statement is not recognised, normally a path should be provided to the part of the process which displays an error message. At the time of writing the MTL processor, this path has been kept open. On recognising the %GENERATE command this path is closed, by generating and compiling internally, the SNOBOL4 code, as shown in the diagram by the dotted line. The linking process in this case is carried out at the time of constructing the semantic tables as well as during compiling the SNOBOL4 extension programs. This is because information about extensibility is received at both stages. After meeting the %GENERATE command, the MSEAL MATCH-BLOCK with its various linking provisions is as in the diagram given below.

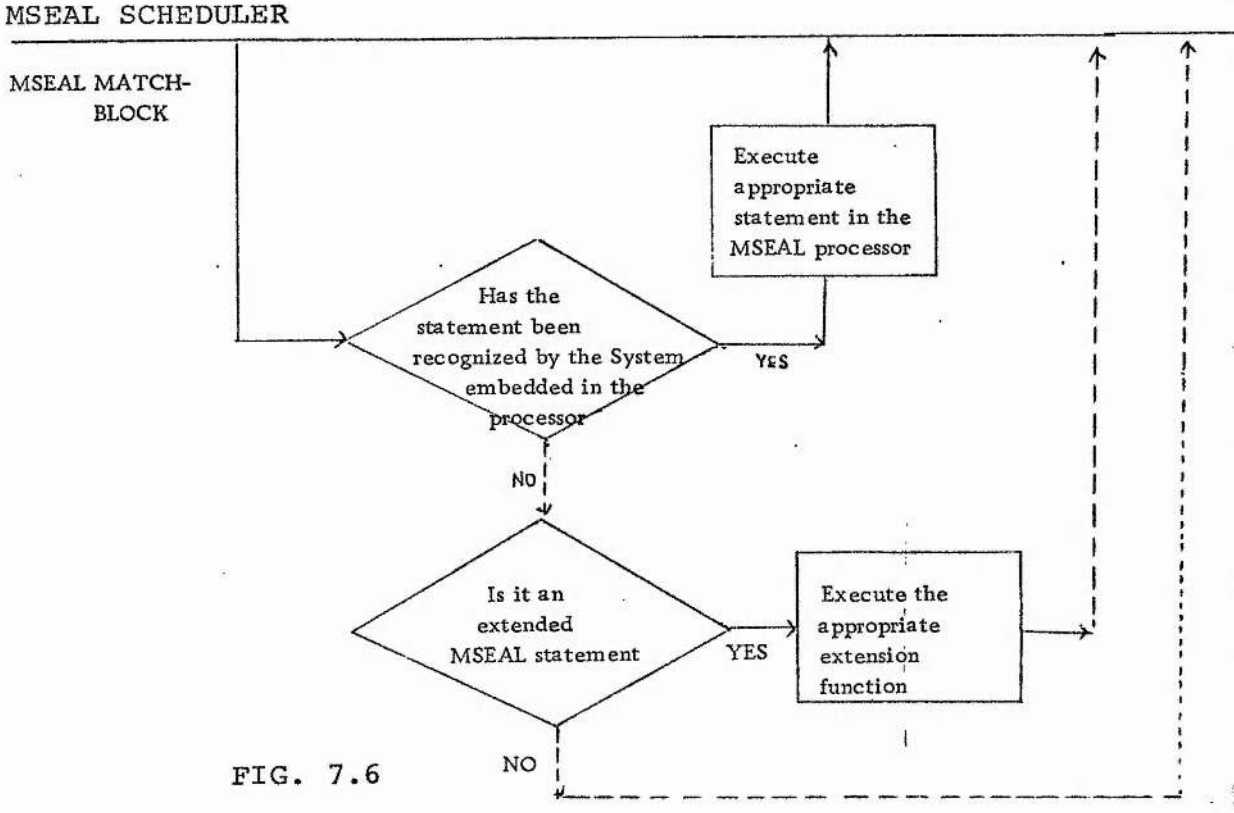


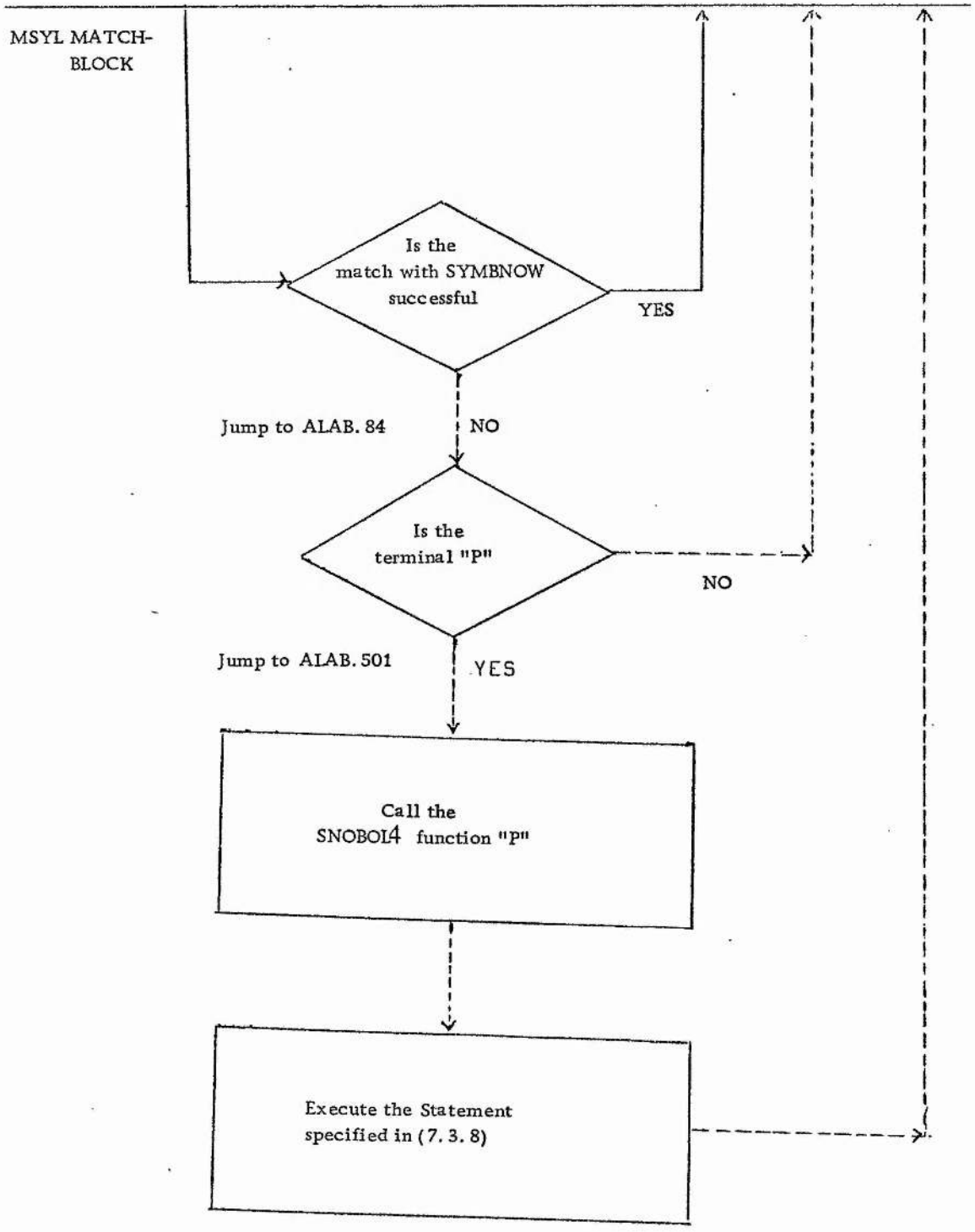
FIG. 7.6

During the construction of the semantic tables when a statement starting with an undefined nonterminal appears in the action field, the linking mechanism for calling the corresponding SNOBOL4 function is provided. At this time a new entry is made in the symbol table of already defined nonterminals. This entry does not point to any node in the syntax graph, but is necessary to make sure that every such nonterminal has a unique linking mechanism. At execution time, the processor treats it like any other statement without even noticing that it is an extension rather than part of the original processor. The linking mechanism for extensions provided by %EXTENSION commands is exactly the same as above except that the name following the %EXTENSION command is considered as the function name.

```
<R> = <R> B | K
<L> = <L> T | T
<Y> = <Y> A | E F
<X> = <Z> A <L> | P C <R>
<Y> = <X> A A A | <P> C D
<S> = <Y>
%SOURCE
*** UNDEFINED NONTERMINALS
P
ALAB.84  IDENT(DEF(R),'P') :S(ALAB.501) ;  :(LABEL.41) ;ALAB.501 P() :(ALAB.158)
 ;
%SNOBOL
 Z = LEN(*FINPOS) SPAN(' ') 'C' SPAN(' ') 'D'
 DEFINE('P()','Z1') :(Z3)
Z1 CARD Z :F(Z2)
 CARD LEN(FINPOS) 'P' :F(RETURN)
 MATCHED = 1 :(RETURN)
Z2 OBSTACLE = 1 :(RETURN)
Z3 ANCHOR = 1
%FINISH
B G D
   £
*** THIS SYMBOL = C OR ITS EQUIVALENT
B C D A A T T A A A
B C D A A T T N
                    §
*** ERROR FOUND
B C K B B B A A A
E E A T T T A A A
E E A T T T A Q A
                  ¥
*** THIS SYMBOL = A OR ITS EQUIVALENT
B C K A A A
B C K A A C
            ?
*** THIS SYMBOL = A OR ITS EQUIVALENT
```

TABLE 7·1

MSYL SCHEDULER



This flow chart should be read in conjunction with table 7.1.

FIG. 7.7

Table 7.2 is a modification of table 7.1. In this table, the edited version of SNOBOL4 statements is also shown. It should be compared with (7.6) to (7.8). (7.8) however is embedded in the MTL processor and is not displayed.

```
<R> = <R> B | K
<L> = <L> T | T
<Z> = <Y> A | E F
<X> = <Z> A <L> | B C <R>
<Y> = <X> A A A | <P> C D
<S> = <Y>
%SOURCE
*** UNDEFINED NONTERMINALS
P
ALAB.84  IDENT(DEF(R),'P') :S(ALAB.501) ;  :(LABEL.41) ;ALAB.501 P() :(ALAB.158)
 ;
%SNOBOL
ALAB.502  Z = LEN(*FINPOS) SPAN(' ') 'C' SPAN(' ') 'D'
          :(ALAB.503)
ALAB.503 DEFINE('P()','Z1') :(Z3)
Z1 CARD Z :F(Z2)S(ALAB.506)
ALAB.506 CARD LEN(INPOS) 'P' :F(RETURN)S(ALAB.508)
ALAB.508 MATCHED = 1 :(RETURN)
Z2 OBSTACLE = 1 :(RETURN)
%FINISH
Z3 %ANCHOR = 1 ; :(ALAB.65)
ALAB.510 :(ALAB.502)
B G D
   $
*** THIS SYMBOL = C OR ITS EQUIVALENT
E C D A A T T A A A
B C D A A T T N
                $
*** ERROR FOUND
P C K P B B A A A
E F A T T T A A A
F F A T T T A D A
                $
*** THIS SYMBOL = A OR ITS EQUIVALENT
B C K A A A
P C K A A P
            $
*** THIS SYMBOL = A OR ITS EQUIVALENT
```

TABLE 7.2

CHAPTER 8

PATE - PROCESSING OF ARITHMETIC AND TEXTUAL EXPRESSIONS.

8.1 INTRODUCTION:-

In this chapter we will describe the programming language PATE (processing of arithmetic and textual expressions). It is a problem oriented language, specially designed for arts students and school children.

It has its basis in SNAP [BARNETT 69] which is a text processing language with restricted arithmetic facilities. Most of its text processing facilities with some modifications have been carried over to PATE. New features have been introduced to handle arithmetic expressions. In the next section the PATE syntax is defined fully. Since it is quite like English, details of its semantics have been omitted. For clarification, reference may be made to working PATE example and the already existing informal SNAP documentation.

At this stage, for reasons stated already further research was diverted towards the automatic translation system which forms the bulk of this thesis and is described in the previous chapters.

8.2 FUNDAMENTALS:-

⟨letter⟩ = A | B | C ..... | Z

⟨digit⟩ = 0 | 1 | 2 ..... | 9

⟨decimal⟩ = .

⟨sign⟩ = + | -

⟨quote⟩ = "

⟨op.brac⟩ = (

⟨cl.brac⟩ =)

⟨alphanumeric⟩ = ⟨digit⟩ | ⟨letter⟩

⟨separator⟩ = , | blanks AND blanks

⟨accumulation⟩ = ANSWER | IT | ITSELF | RESULT | RESULTING FROM

⟨dummy word⟩ = CALCULATE | EVALUATE | TAKE | LIST | ELEMENT | ELEMENTS | OF | THE | BY | TO | AN | A | BE | FOR | SPACE | NUMBER | NUMBERS

⟨system control command⟩ = IGNORE | TERMINATE | EXECUTE

⟨affirmative comparison word⟩ = EQUAL TO | GREATER THAN | LESS THAN | SAME AS | LESS THAN OR EQUAL TO | GREATER THAN OR EQUAL TO

⟨negative comparison word⟩ = NOT ⟨affirmative comparison word⟩

⟨comparison word⟩ = ⟨affirmative comparison word⟩ | ⟨negative comparison word⟩

⟨type⟩ = INTEGER | DECIMAL

⟨integer⟩ = &integer& &digit& | &digit&

⟨decimal number⟩ = &decimal& &integer& | &integer& &decimal& &integer&

⟨unsigned number⟩ = ⟨integer⟩ | ⟨decimal number⟩

⟨number⟩ = &unsigned number& | &sign& &unsigned number&

⟨identifier⟩ = &identifier& &alphanumeric& | &letter&

⟨label⟩ = &brac op& &identifier& &brac.cl.&

⟨quoted string⟩ = &quote& &string& &quote&

⟨ordinal adjective⟩ = 1-ST | 2-ND | 3-RD | &integer& -TH

⟨extracted element⟩ = ⟨ordinal adjective⟩ ⟨identifier⟩

⟨extraction⟩ = ⟨ordinal adjective⟩ TO ⟨ordinal adjective⟩ |
⟨ordinal adjective⟩ THROUGH ⟨ordinal adjective⟩ |
⟨ordinal adjective⟩ AND SUBSEQUENT |
⟨ordinal adjective⟩ AND PRECEDING

⟨variable⟩ = ⟨identifier⟩ | ⟨extracted element⟩ | ⟨accumulation⟩

⟨extracted expression⟩ = ⟨extraction⟩ CHARACTERS OF ⟨identifier⟩ | ⟨ordinal adjective⟩ CHARACTER OF ⟨identifier⟩ | ⟨extraction⟩ ⟨identifier⟩

⟨object⟩ = ⟨identifier⟩ | ⟨extracted expression⟩

⟨length exp⟩ = LENGTH OF ⟨object⟩ | LENGTH OF ⟨character string⟩ | LENGTH of ⟨numeric string⟩

⟨numeric string⟩ = &numeric string& &separator& &number& &number&

⟨string list⟩ = &string list& &separator& &string& | &string&

⟨character string⟩ = ⟨quoted string⟩ | &quote& &string list& &quote&

8.3 ARITHMETIC STATEMENT:-

⟨co-ordinate conjunction⟩ = AFTER THAT | THEN

⟨term⟩ = ⟨formula⟩ | ⟨primary⟩

⟨multiple function word⟩ = TOTAL | PRODUCT | ADD | MULTIPLY

⟨multiple function exp⟩ = ⟨multiple function word⟩ ⟨term⟩ ⟨separator⟩ ⟨term⟩
⟨multiple function exp⟩ ⟨separator⟩ ⟨term⟩

⟨diadic function word⟩ = DIFFERENCE | QUOTIENT | DIVIDE | SUBTRACT

⟨binary function exp⟩ = ⟨diadic function word⟩ ⟨term⟩ ⟨separator⟩ ⟨term⟩

⟨function exp⟩ = ⟨binary function exp⟩ | ⟨multiple function exp⟩

⟨infix exponent word⟩ = EXPONENT | POWER

⟨exponent formula⟩ = ⟨quantity⟩ ⟨infix exponent word⟩ ⟨quantity⟩

⟨infix division word⟩ = DIVIDED BY | OVER

⟨division formula⟩ = ⟨quantity⟩ ⟨infix division word⟩ ⟨quantity⟩

⟨infix multiplication word⟩ = MULTIPLIED BY | PLUS

⟨multiplication formula⟩ = ⟨quantity⟩ ⟨infix multiplication word⟩ ⟨quantity⟩

⟨multiplication formula⟩ ⟨infix multiplication word⟩ ⟨quantity⟩

⟨infix subtraction word⟩ = SUBTRACTED | MINUS

⟨subtraction formula⟩ = ⟨quantity⟩ ⟨infix subtraction word⟩ ⟨quantity⟩

⟨infix addition word⟩ = ADDED TO | PLUS

⟨quantity⟩ = ⟨primary⟩ ⟨function exp⟩;

⟨addition formula = ⟨quantity ⟨infix addition word⟩ ⟨quantity⟩ | ⟨addition formula⟩ ⟨infix addition word⟩ ⟨quantity⟩

⟨formula⟩ = ⟨exponent formula⟩ | ⟨division formula⟩ | ⟨multiplication formula⟩ | ⟨subtraction formula⟩ | ⟨addition formula⟩

⟨basic exp⟩ = ⟨formula⟩ | ⟨function⟩

⟨primary exp⟩ = ⟨basic exp⟩ | ⟨specification exp⟩ ⟨co-ordinate conjunction⟩ ⟨specification exp⟩

⟨specification exp⟩ = ⟨primary exp⟩ | ⟨set statement body⟩

8.4 MISCELLANEOUS STATEMENTS:-

⟨declaration body⟩ = LET ⟨identifier⟩ BE ⟨type⟩

⟨reserve statement body⟩ = RESERVE ⟨integer⟩ ⟨identifier⟩

⟨call statement body⟩ = CALL ⟨character string⟩ ⟨identifier⟩

<set statement body> = SET <identifier> TO <numberic string> | SET <extracted element> TO <numeric strig>

<copy statement body> = COPY <object> AND CALL IT <identifier>

<identification statement body> = <declaration body> | <reserve statement body> | <call statement body> <set statement body> <copy statement body>

<delete statement body> = DELETE <object>

<append statement body> = APPEND <object> TO <object>

<overwrite statement body> = OVERWRITE object> OVER <object>

<editorial statement> = <delete statement body> | <append statement body> | <overwrite statement body>

<read statement body> = READ <identifier>

<printable item> = <object> | <number> | <character string> | <numeric list>

<print statement body> = PRINT <printable item>

<I/O statement body> = <read statement body> | <print statement body>

<value> = <number> | <numeric list> | <character string>

<unconditional jump> = CONTINUE WITH <identifier> | REPEAT FROM <identifier>

<conditional exp = IF <object> <comparison word> <object> | IF <value> <comparison word> <object> IF <object> <comparison word> <value>

Every element of a list must be a scalar. An EXECUTE statement at the end of a program causes the execution mode to be entered. DEFINE statements are used to define the system commands. The portion of a card on the right of an IGNORE statement is ignored.

8.6 IMPLEMENTATION:-

The PATE processor is an interpreter. Before going any further, we shall try to justify this decision. In a PATE program an identifier may be used to refer to any type of value : string, decimal number, integer, vector. During the execution of a program, this type may change. Thus it is not possible, at compile time to know the type of an identifier and it is difficult to generate code without knowing the type in advance.

A polymorphic operator is one whose action depends on the context in which it occurs. Almost all the PATE operators are polymorphic. Closely related to the no-type-nature of PATE are problems arising from its polymorphism.

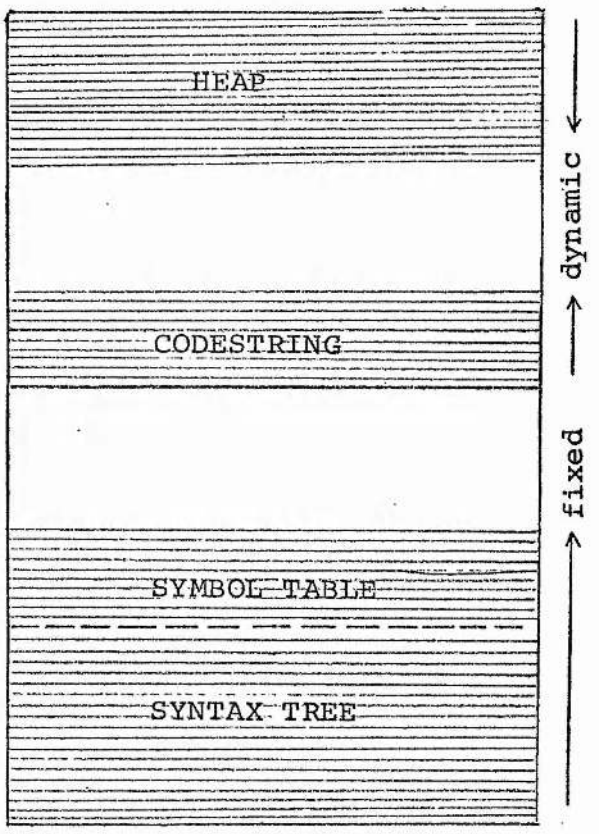
EXAMPLE 8.1:-

A PLUS B.

If and how this statement is to be executed depends on whether one or both of A and B are integers, decimal numbers or strings.

When the implementation was begun (Oct. 1971) the only high level language available which gave access to all the facilities of the computer system was FORTRAN IV. Since it does not have any string manipulation facilities, it was natural to write an interpreter for PATE.

All expressions are first processed by the syntax analyser and code strings of the intermediate language are generated internally. This code string is then used for execution of the program. Syntax analysis of arithmetic expressions is done by the top-down parser described in the 3rd and 4th chapters. All other expressions form a finite state grammar and are analysed separately. We will first describe the general layout of the workspace and then consider its different parts one by one.

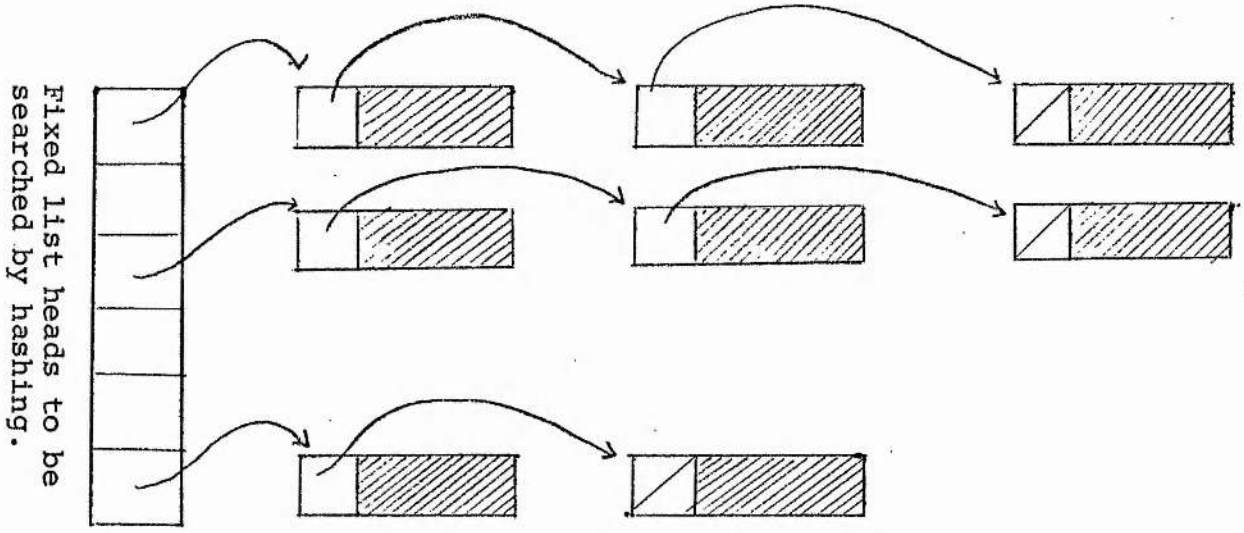


Layout of the PATE workspace at execution time.

FIG. 8.1

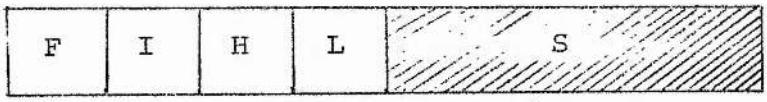
The organisation of the PATE symbol table is based on the concept of hash addressing and linked lists. Symbols are classified in categories and the number of categories coincide with the total number of linked lists. Each linked list accommodates one category of information. The symbols along with necessary

information about their size and forward pointers are stored in blocks of storage known as descriptors. Values are stored in blocks of storage known as qualifiers.



Layout of the PATE Symbol table

FIG. 8.2



Layout of an element of
the PATE descriptor.

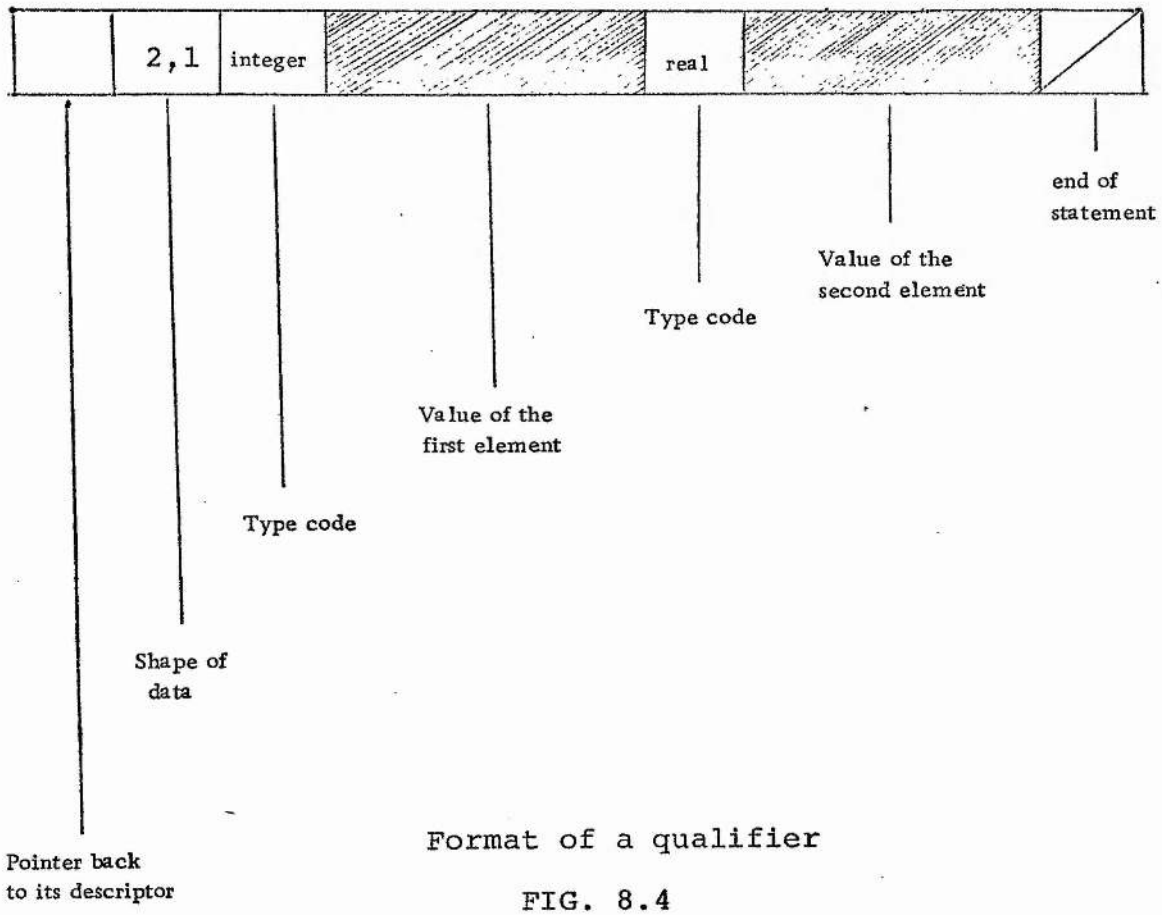
FIG. 8.3

"F" is the forward pointer field

"I" is the identification field. It holds a unique code for each different type of data object and for each one of the system defined symbols.

field "H" is only used for user defined symbols. Integers and decimal numbers, when used as scalars are stored in this field. In all other cases it points to the value of the symbol in the heap.

field "L" is used to store the number of characters in the symbol. The symbol itself is stored character by character in the "S" field.



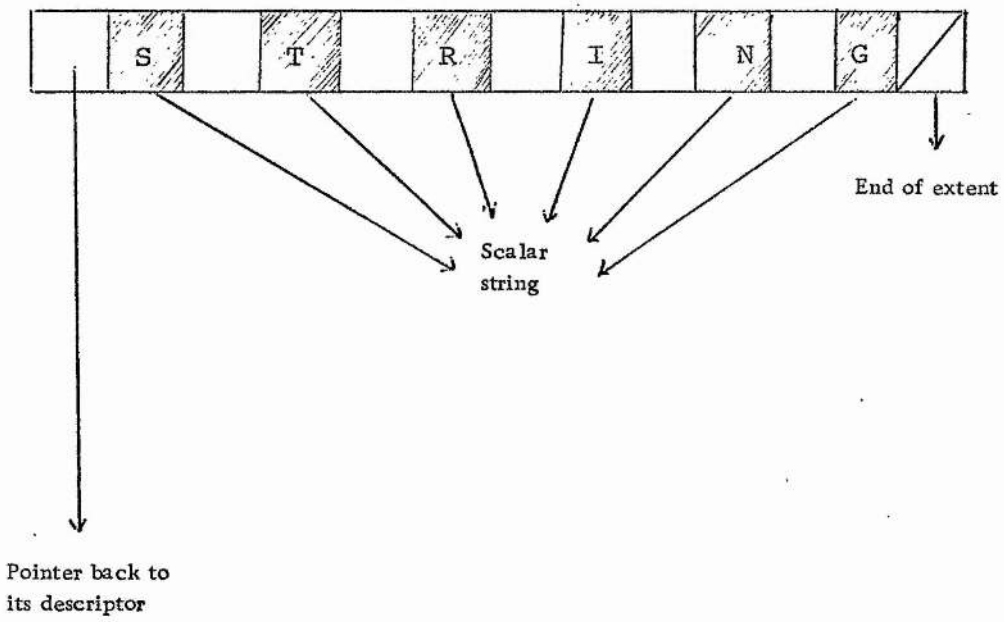
Format of a qualifier

FIG. 8.4

The first cell of a qualifier is a pointer to its descriptor. These pointers are used in garbage collection.

The second cell of the qualifier indicates the size and shape of the qualifier.

Each element is preceded by the appropriate "type code" rather than one over all type for the qualifier. A whole character string however big it may be is considered as one element of a qualifier.



Format of a character string qualifier

FIG. 8.5

A character string is always stored in a linked list as this makes the task of editing the string much simpler.

The code string of most statements is straight forward. Arithmetic expressions are divided into meangful subexpressions of the smallest possible size. These subexpressions are converted into reverse polish form, reassembled and then stored in the system.

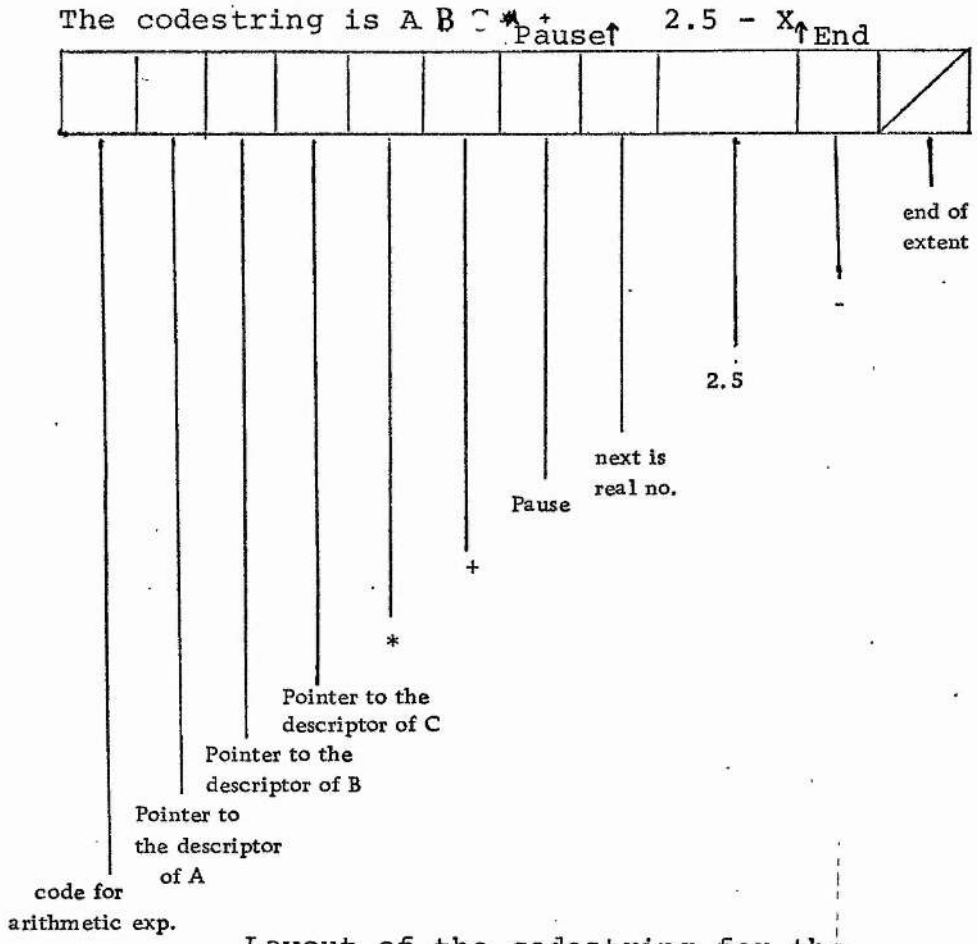
EXAMPLE 8.2:-

ADD A AND B TIMES C THEN FROM ITS RESULT SUBTRACT 2.5.

The above expressions can be subdivided into the following subexpressions:

ADD A AND B TIMES C

FROM THE RESULT SUBTRACT 2.5



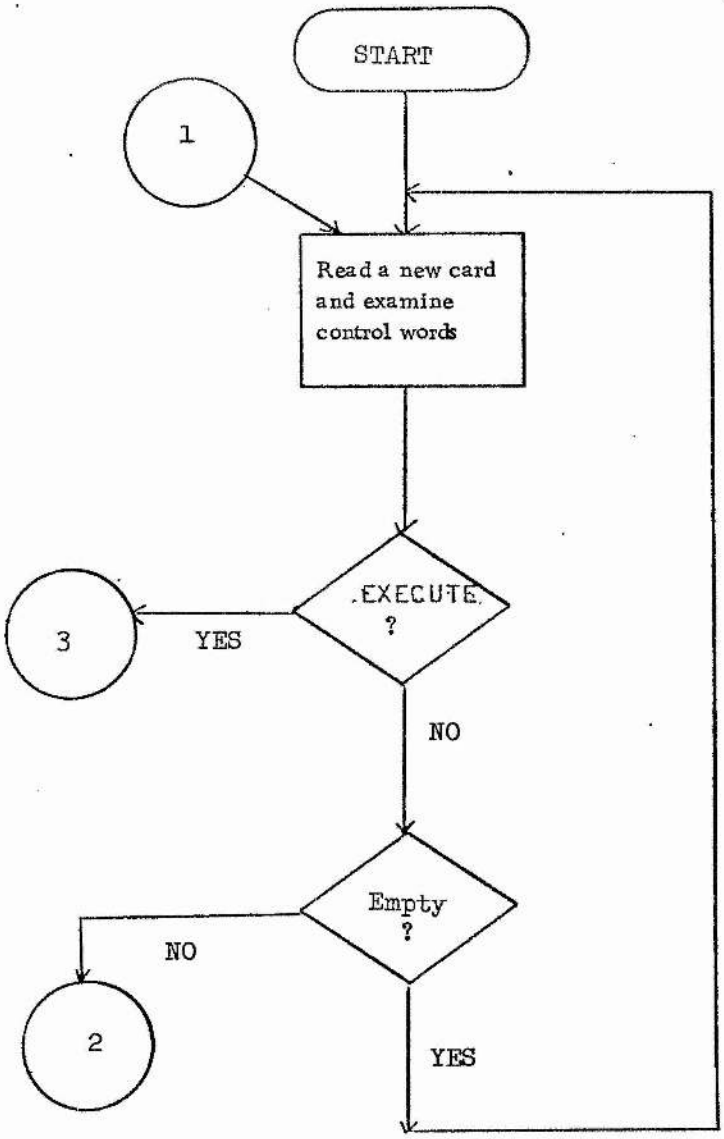
Layout of the codestring for the above mentioned arithmetic expression

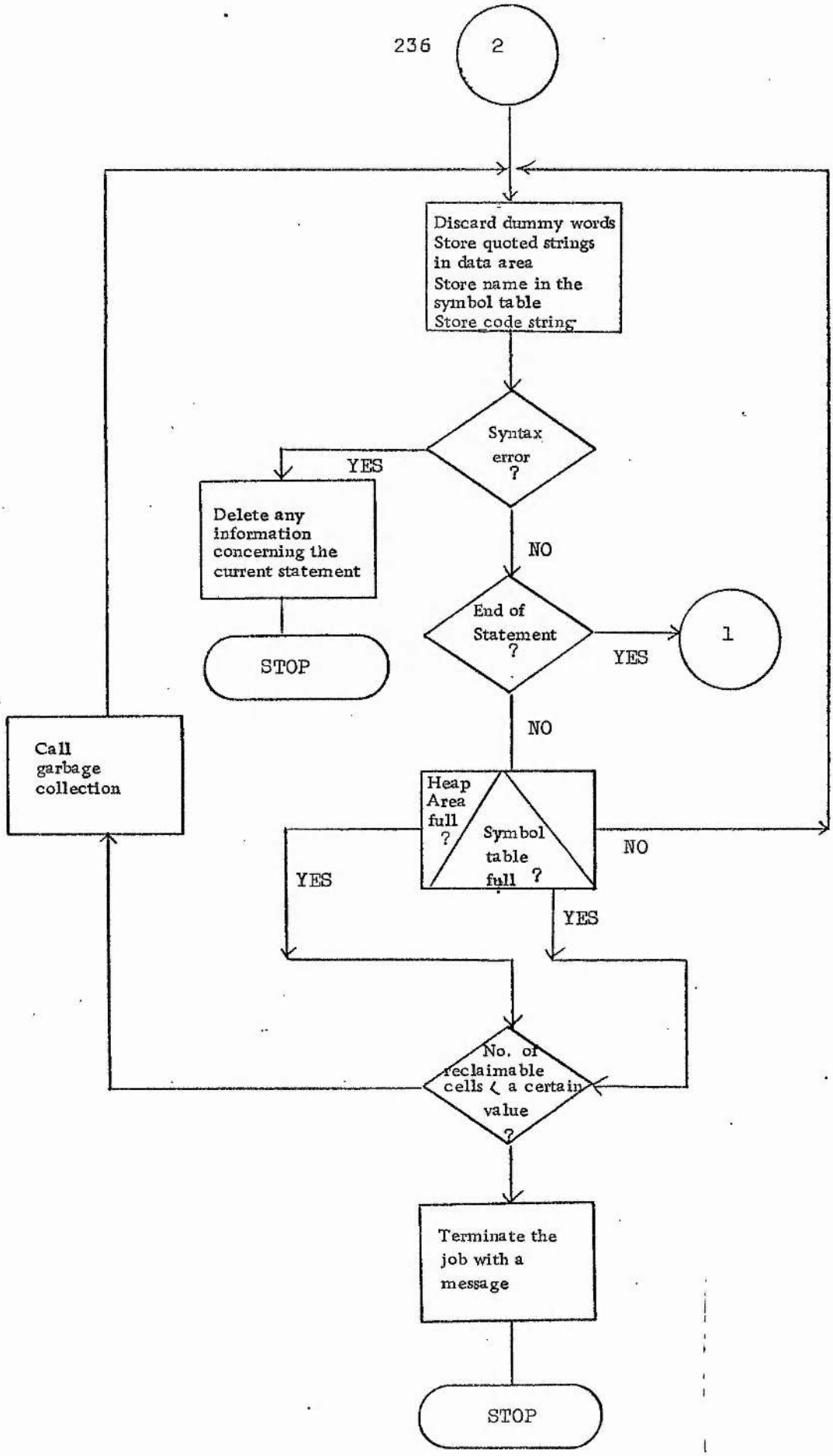
FIG. 8.6

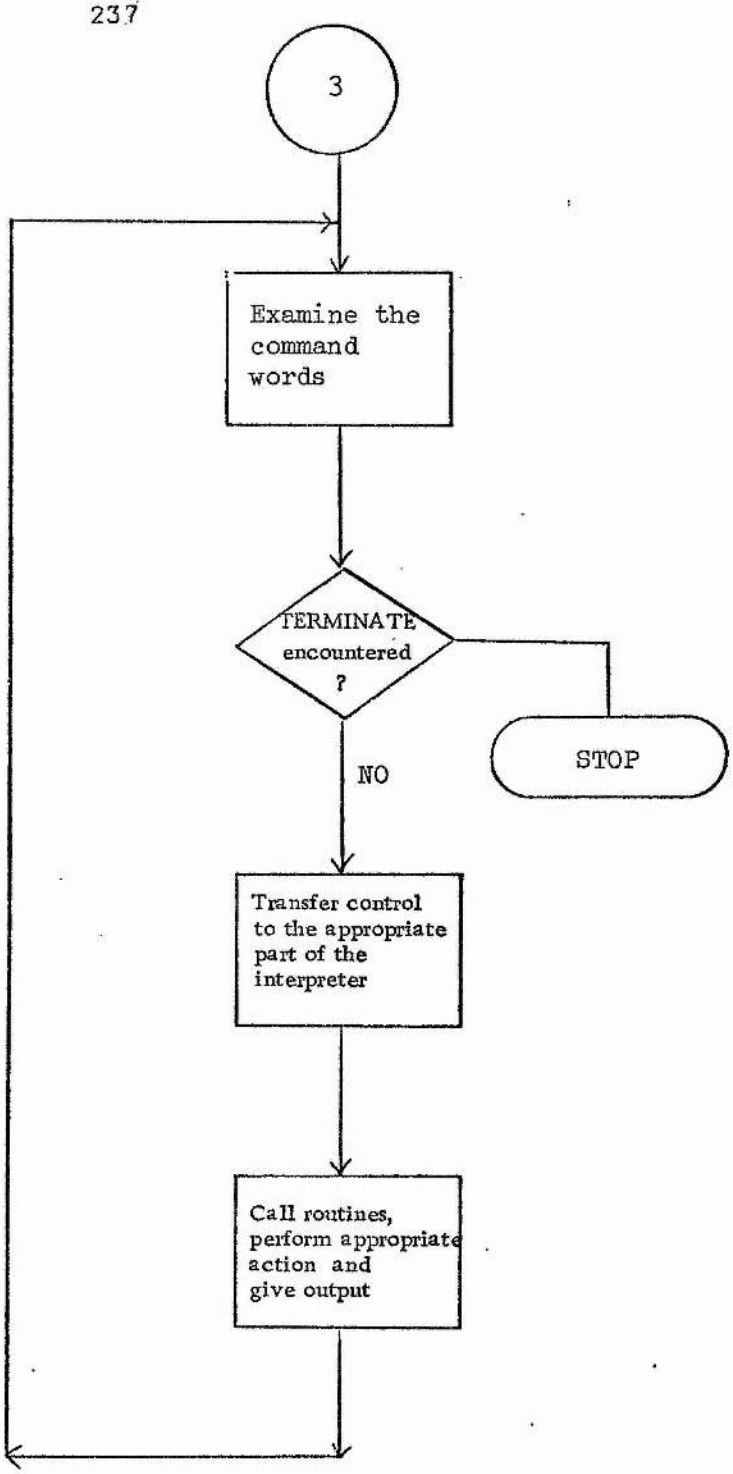
Each descriptor points to its qualifier, which in turn points back to it. Using these pointers heap entries can be compacted in one side of the work space. Qualifiers are shifted one at a time. Pointers from descriptors to the qualifiers are updated as the shifting takes place.

Although it is not possible to compact the symbol table entries, garbage can be collected by using a free space list. To start with, the space list is empty. As the processing proceeds, the deleted cells are added to it. The system is so organised that the space list always consists of the biggest possible blocks of storage. For all subsequent demands of space in the symbol table, it is acquired from the smallest possible block of storage in the space list. The required number of cells are taken out of the list and the remainder (if any) maintains its identity in the space list. Only if there is no adequate chunk of storage available in the space list is the space acquired from the main storage.

At compile time, the syntax analysis stack and the reverse polish stack develop towards each other in a separate area. At execution time this space can be reclaimed.


START
1
Read a new card and examine control words
EXECUTE ?
YES
3
NO
Empty ?
NO
2
YES


2
Discard dummy words
Store quoted strings in data area
Store name in the symbol table
Store code string
Syntax error ?
YES
Delete any information concerning the current statement
STOP
NO
End of Statement ?
YES
1
NO
Call garbage collection
Heap Area full ?
Symbol table full ?
NO
YES
YES
No. of reclaimable cells < a certain value ?
Terminate the job with a message
STOP


3
Examine the command words
TERMINATE encountered ?
NO
STOP
Transfer control to the appropriate part of the interpreter
Call routines, perform appropriate action and give output

8.7 CONTROL CARDS:-

The management routine of the processor is divided into different control blocks. Each control block represents a different mode and can be entered by using appropiate control cards. Control cards are also used to control the listing of the program. Some of the control cards are system defined while other are explained below.

After introducing a card "%SNAP" or "%PATE", the program follows. In addition to the above mentioned obligatory cards, the following optional cards may be used. "%LIST" is used to start the listing of the program, which is assumed to be the case at the start of a program. "%NOLIST" is used to stop the listing. Listing can be started and stopped as many times as user wishes.

The introduction of "%QUIT" card terminates the job.

```
%SNAP
IGNORE. ***************************************************************************
IGNORE.                    ILLUSTRATION OF LIST DECLARATION,
IGNORE.                          ARITHMETIC STATEMENT
IGNORE.                                AND
IGNORE.                           CONDITIONAL LOOP
IGNORE. ***************************************************************************
IGNORE.
IGNORE.
IGNORE.
RESERVE SPACE FOR 10 ELEMENTS OF NAME LIST.
CALL"CAROLINE MACAULY"1-ST NAME.
CALL"MARY SMITH"2-ND NAME.
CALL"ANN STOUT"3-RD NAME.
CALL"DIANE ROGER"4-TH NAME.
CALL"DEBORAH IMLACH"5-TH NAME.
CALL"ROBERT MCCARDY"6-TH NAME.
CALL"SUSAN STEWART"7-TH NAME.
CALL"AGNES WILSON"8-TH NAME.
CALL"MARY CROSS"9-TH NAME.
CALL"ELSIE RITCHIE"10-TH NAME.
PRINT"                                   NAME LIST".
PRINT"     ".
PRINT THE NAME LIST.
PRINT"     ".
PRINT"     ".
PRINT"THE FOLLOWING NAMES HAVE BEEN PRINTED ONE AT A TIME".
SET TOTAL TO THE REGISTERED LENGTH OF NAME LIST.
SET N TO 0.
(LABEL)
SET N TO THE SUM OF 1 AND N.
PRINT"     ".
PRINT THE N-TH NAME.
IF N IS EQUAL TO TOTAL TERMINATE,OTHERWISE REPEAT FROM THE LABEL.
TERMINATE.
EXECUTE.
```

TABLE 8.1

NAME LIST

CAROLINE MACAULY,MARY SMITH,ANN STOUT,DIANE ROGER,DEBORAH IMLACH,ROBERT MCCARDY,
SUSAN STEWART,AGNES WILSON,MARY CROSS,ELSIE RITCHIE

THE FOLLOWING NAMES HAVE BEEN PRINTED ONE AT A TIME

CAROLINE MACAULY

MARY SMITH

ANN STOUT

DIANE ROGER

DEBORAH IMLACH

ROBERT MCCARDY

SUSAN STEWART

AGNES WILSON

MARY CROSS

ELSIE RITCHIE

CHAPTER 9

9.1 CONCLUSION

Before embarking on the project, an extensive survey was made on the existing techniques in topics related to the automatic translator e.g. Syntax analysis, Semantic Synthesis and extensibility. It is in the light of this comparative study that various decisions were made in the automatic translator. To throw light on the reasons behind various decisions, this discussion has been included in the thesis.

The translator translation system described in this thesis deals with a special class of languages which can be described by ELL(k) grammars. Since ELL(k) parser parses without backup, it is easy to give prompt syntactic diagnostics which is not possible with back-up oriented algorithms. It is also hoped that ability to make correct decision at every stage of the parse should make it faster.

ELL(k) grammars allow very general left recursion. To the best of our knowledge, no other top down parser can use a left recursive grammar for left to right recognition. Hence there are grammars which can be recognised by no top-down parser but the ELL(k).

Conventional top-down parsers generate a parse tree which is consumed by the corresponding semantic synthesiser. The MTL processor generates code in a single scan. No interim parse is generated, but it is used for semantic synthesis. During the recognition of a source language statement, at the appropriate state of parsing, the semantic synthesiser is called for action. A parse tree could be generated from the information in MSEAS as an optional facility for the user.

The only formal specification in the conventional compiler compilers have been in the syntax specification. The MTL processor provides a great degree of formalism for semantic specification.

Semantics are specified in a metasemantic language in the form of semantic productions. These semantic productions are used repeatedly by the processor in a manner which is quite similar to that of semantic productions.

In this thesis various reasons have been stated for favouring extensible programming languages and need: no repetition. We have shown how SNOBOL4 (SPITBOL) can be used as an implementation language for extensible languages. The extensibility provides both the syntax analyser and the semantic synthesiser with extra power.

When we add the syntax analyser, the semantic synthesiser and the features of extensibility, what do we get?

In biblical terms, a classus on gold feet, silver legs, iron thighs topped with a clay head. Those parts of a compiler which really matter to the machine, especially code generation, machine dependent and machine independent optimisation are missing. Still the parts we have shown

here are sufficiently general to be a sizeable part of any compiler design. These parts can be considered as "off the shelf" compiler components. We believe that it is possible to write these parts of a compiler in an appropriate language and then incorporate them in the desired compiler. Alternatively, the MTL can be extended to incorporate features such as to be required for code generation and other parts of a translator. We do not contend that the net result will be an ideal compiler compiler, but we are of the opinion that MTL can be extended to form an efficient translator translation system.

One objection to our approach might be that it is too much dependent on SNOBOL4. It is true so far as the existing version of MTL is concerned. The object of this project was to test the algorithms, which has been achieved. We believe our existing design can be easily modified to make it SNOBOL4 independent.

Another objection against MSEAL is that it does not provide block structure as a tool for structured programming. It is so because the MSEAL was designed to be a notation for specifying semantics rather than a programming language. If experience shows that such a structure is desirable, investigation should be made into its feasibility.

The present processor has a number of known sources of inefficiency:

(1) In the current implementation of MSEAL, the tables are searched sequentially. There exists scope for improving this process.

(2) The prototype processor reads the syntax and the semantic specification and constructs internal tables before every run. In the production processor, it should be possible to initialise the internal tables once and for all. This process can be automated and the initialisation code generated by the processor itself or by a library routine called by the processor.

There are various improvements that can be made in the semantic synthesiser. For example the current version can be extended to cater for left recursion even in the production mode. To handle left recursion, the syntax analyser takes a rather unconventional approach. Every production is not tried individually as otherwise is the case. All the mutually left recursive productions are first stacked on the SAS and the symbols covered by them are recognised. It is necessary to have a compatible approach for the semantic synthesiser.

Due to the constraints of time, this was not implemented in the present proto type process and can be introduced in the new version.

Lewis and Stearns show that it is computable problem to show whether a given grammar is LL(k). We do not know whether this is the case for ELL(k) and whether a practical algorithm could be constructed even for special cases.

APPENDIX I

SOME FACTS ABOUT LL(k) GRAMMARS:-

1) A grammar $G = (V_T, V_N, P, S)$ is said to be an LL(k) grammar for some positive integer k if and only if given

   a) a word ω in V_T^* such that $|\omega| \leq k$;

   b) a nonterminal A in V_N;

   c) a word ω in V_T^*;

   there is at most one production p in P such that for some ω_2 and ω_3 in V_T^*;

   d) $S \Rightarrow \omega_1 A \omega_3$;

   $A \Rightarrow \omega_2$ (p)

   $(\omega_2 \omega_3) : k = \omega$

   stated informally in terms of parsing, an LL(k) grammar is a context free grammar such that for any word in its language, each production in its derivation can be identified with certainty by inspecting the word from its beginning (left end) to the k-th symbol beyond the beginning of the production. Thus when a nonterminal is to be expanded during a top down parse, the portion of the input string which has been processed so far plus the next k input symbols determine which production must be used for the nonterminal. Thus the parse can proceed without backtrack.

2) If G is LL(k), then for all A in V_N, ω in V_T^* : k, $R \subseteq V_T^*$: k satisfying $R \subseteq R(\omega_1) = \{\omega_3 : k \mid S \Rightarrow \omega_2 A \omega_3\}$ for some ω_1 in V_T^*, there exists at most one production p such that $A \Rightarrow \omega_2$ (p) and $(\omega_2 \omega_3)$: k = ω for some ω_2 and ω_3 in V_T^* such that ω_3 is in R.

It states that a production in LL(k) grammar can also be identified using only k symbols which follow and the set R(ω1) where R(ω1) is the set of all k symbol sequences which can follow the rightmost descendant of that production.

3) An LL(k) grammar is always LR(k) as defined by Knuth.

4) An LL(k) grammar is unambiguous.

5) Given a grammar G and k, it is decidable whether or not G is LL(k)

6) A grammar G = (V_T, V_N, P, S) is said to be a strong LL(k) grammar for some positive integer k if and only if given

(a) a word ω in V_T^* such that $|\omega| \leq k$

(b) a nonterminal A in V_N;

There is at most one production p in P such that for some ω_1, ω_2 and ω_3 in V_T^*

(c) $S \Rightarrow \omega_1 A \omega_3$;

(d) $A \Rightarrow \omega_2$ (p);

(e) $(\omega_2\ \omega_3):k = \omega$

The only difference between this definition and that of an LL(k) grammar is the qualifier "for all ω_1" has been moved within the scope of the "there is at most one production p".

7) Given an LL(k) grammar $G = (V_T, V_N, P, S)$, one can find a structurally equivalent strong LL(k) grammar.

8) Given an LL(k) grammar $G = (V_T, V_N, P, S)$, an LL(k+1) grammar without Λ-rules can be constructed which generates the language $L(G) - \{\Lambda\}$

9) An LL(k) grammar can have no left recursive nonterminals. (this statement is not valid in the light of our algorithm)

10) Given an LL(k+1) grammar without Λ-rules for $k \geq 1$, there exists an LL(k) grammar with Λ-rules for the same language.

11) There exists no LL(k) grammar without Λ-rules for the language $\{a^n (b^k d + b + cc)^n \mid n > 1\}$ where $k \geq 1$.

12) For every $k \geq 1$, the class of languages generated by LL(k) grammars is properly contained within the class generated by LL(k+1) grammars.

13) For every $k \geq 1$ the class of languages generated by LL(k) grammars without Λ-rules is properly contained within the class of languages generated by LL(k+1) grammars without Λ-rules.

14) It is decidable if two LL(k) grammars generate the same language.

15) Given a context free language, it is decidable whether or not there exists a k such that the grammar is LL(k).

16) Given an LR(k) grammar of known k, it is decidable if there exists a k such that the grammar is LL(k)

17) It is undecidable whether or not an arbitrary context free grammar generates an LL(k) grammar, even for a fixed k.

18) A grammar is said to be in Greïbach normal form if the right hand side of every production begins with a terminal symbol. Given an LL(k) grammar without Λ-rules, another LL(k) grammar in Greibach normal form can be obtained for the same language.

19) Given an LL(k) grammar G with Λ-rules, a strong LL(k+1) grammar in Greibach normal form can be obtained for $L(G) - \{\Lambda\}$.

20) Let G be a context free grammar. Suppose that every production p in G is of the form $A \Rightarrow bB$ or $A \Rightarrow a$, where A and B are nonterminals and a, b are terminals. Then G is called a regular grammar. If the finite union of disjoint LL(k) language is regular, then all the languages are regular.

21) If $A \subseteq B$, then the complement of A with respect to B is the set B-A. The complement of a nonregular LL(k) language is never LL(k).

22) The LL(k) languages are not classed under complementation, union, intersection, reversal, concatenation, or Λ-free homomorphisms.

EXCLUSION OF LEFT RECURSION:-

An LL(k) grammar G can have no left recursive nonterminals.

PROOF:-

Assume that an LL(k) grammar has a left recursive symbol. Then for some nonterminal A, $A \Rightarrow^* A\ Y(p)$ and $A \Rightarrow^* X(p')$ where X and Y are in V_T^*, and p and p' are different production. Because G is unambiguous, $Y \neq \Lambda$. Furthermore $S \Rightarrow^* uAv$ for some u and v. Now consider the derivations.

$S \Rightarrow^* uAv \Rightarrow^* uAy^k v \Rightarrow^* u\ x\ y^k v$

$S \Rightarrow^* uAv \Rightarrow^* uAy^k v \Rightarrow^* u\ A\ y^{k+1} v \Rightarrow^* u\ x\ y^{k+1} v$

Thus $S \Rightarrow^* u\ A\ y^k v$ $A \Rightarrow^* xy(p)$ $A \Rightarrow^* X\ (p')$ and

$(X\ y^{k+1} v) : k = (x\ y^k v) : k$

Therefore, since the grammar is LL(k) it can not have a left recursive nonterminal.

APPENDIX II

CROSS REFERENCE:-

In a typical SNOBOL4 program, all labels and a good deal of identifiers are global. It is necessary to make sure that the conflicts do not arise. It is therefore recommended that the user should use labels and identifiers according to some systematic scheme and make separate tables for them (for example labels can be of the form LABEL.1, LABEL2.....etc). However the following SNOBOL4 program can be used to cross reference a user program. The user program appears as its data. In fact the program itself has been used as a user's program in the following example.

```
-NOLIST
-INBO
 1          OUTPUT(.HEADING,'SYSOPT','1')
 2          &STLIMIT = 1000000
 3          FIELDWIDTH = 72
 4          SYSUNITS = 'SYSIPT'
 5          NAME = TAPLE()
 6          BRANCH = TABLE()
 7          LABEL = TABLE()
 8          FUNCTION = TABLE()
 9          VARIABLE = TABLE()
10          NAME<'1'> = '9'
11          WORD = ANY('ABCDEFGHIJKLMNCPQRSTUVWXYZ&')
+                (SPAN('ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789') | '')
12          DEFINE('APPEND(IDENTIFIER,TABULL)')
13          DEFINE('PROCESS()')
14          DEFINE('READ()')  :(DEFNEW)
15  READ    READ = INPUT  :F(FRETURN)
16          PUT = READ  :(RETURN)
17  DEFNEW  DEFINE('NEWNAME(NAM)A,PLACE')  :(NXTCRD)
18  NEWNAME PLACE = '1'
19          MAXSIZE = GT(SIZE(NAM),MAXSIZE) SIZE(NAM)
20  NXTNAME A = NAME<PLACE>
21          PLACE = A LGT(NAM,A)  :S(NXTNAME)
22          NAME<PLACE> = NAM
23          NAME<NAM> = A  :(RETURN)
    *
24  NXTCRD  CARD = READ()  :F(TSTEOF)
25          CARD POS(FIELDWIDTH) REM . TEMP =
26          CARD POS(0) ANY('-*')  :S(CTRLCRD)
27          LINE = CARD
28  DELLIT  LINE '"' BREAK('"') '"' | "'" BREAK("'") "'" =  :S(DELLIT)
29          LINE POS(0) '+'  =  :S(CONTCRD)
    *
    * PROCESS A NORMAL STATEMENT CARD
    *
30          STATEMENTS = STATEMENTS ';' LINE
31          OUTPUT = LPAD(STMTCT + 1,5) '  ' CARD '.' TEMP
32  NXTSTAT STATEMENTS BREAK(';') . STMT ';' =  :F(NXTCRD)
33          PROCESS()
34          STMT POS(0) 'END '  :S(TOEOF)
35          STMTCT = STMTCT + 1  :(NXTSTAT)
36  TOEOF   ?READ()  :S(TOEOF)F(PRINT)
    *
    * PROCESS A CONTINUATION CARD
    *
```

```
37  CONTCRD STATEMENTS = STATEMENTS LINE
38          OUTPUT = '        ' CARD '.' TEMP  :(NXTSTAT)
    *
    * PROCESS A CONTROL OR COMMENT STATEMENT
    *
39  CTRLCRD OUTPUT = '        ' CARD '.' TEMP
40          CARD POS(0) '-IN' LEN(2) . FIELDWIDTH  :S(NXTCRD)
41          CARD POS(0) '-COPY' SPAN(' ') BREAK(' ') . SYSUNIT  :F(NXTCRD)
42          INPUT(.INPUT,SYSUNIT)
43          SYSUNITS = SYSUNITS SYSUNIT  :(NXTCRD)
    *
    * TEST EOF WAS IN SYSIPT & IF SO PRINT RESULTS
    *
44  TSTEOF  IDENT(SYSUNITS,'SYSIPT')  :S(SETEND)
45          SYSUNITS RPOS(12) LEN(6) . SYSUNIT REM = SYSUNIT
46          INPUT(.INPUT,SYSUNIT)  :(NXTCRD)
47  SETEND  LABEL<'END'> = IDENT(SUBSTR(CARD,1,3),'END') ' ' LPAD(STMTCT,4,'0')
    +                    :S(PRINT)
48          OUTPUT = '    **** END CARD MISSING ****'  :(PRINT)
    *
    * SUBROUTINE TO PROCESS A STATEMENT AND EVALUATE REFERENCES IN IT
    *
49  PROCESS (STMT ' :') BREAK(' ') . LAB BREAK(':') . BODY REM . BRFIELD
50          STNUM = ' ' LPAD(STMTCT,4,'0')
51          DIFFER(LAB) APPEND(LAB,LABEL)
52  GETBR   BRFIELD '(' WORD . BR =  :F(GETVAL)
53          APPEND(BR,BRANCH)        :(GETBR)
54  GETVAL  BODY WORD . VAR LEN(1) . CHAR =  :F(RETURN)
55          DIFFER(CHAR,'(')  APPEND(VAR,VARIABLE)  :S(GETVAL)
56          APPEND(VAR,FUNCTION)        :(GETVAL)
57  APPEND  IDENT(NAME<IDENTIFIER>) NEWNAME(IDENTIFIER)
58          TABULL<IDENTIFIER> RPOS(5) STNUM  :S(RETURN)
59          TABULL<IDENTIFIER> = TABULL<IDENTIFIER> STNUM  :(RETURN)
    *
```

```
    * PROGRAM TO PRINT RESULTS OF CROSS-REFERENCE
    *
60  PRINT   NAMESIZE = ((MAXSIZE + 4) / 5) * 5 + 1
61          VARSIZE = 61 - NAMESIZE
62          NAM = '1'
63  PRTNAME NAM = NAME<NAM>
64          IDENT(NAM,'9')  :S(END)
65          WORD = NAM
66          CHAR = '-'
67  NXTLIN  DIFFER(VARIABLE<NAM> VARSTR)  :F(CONT1)
68          VARIABLE<NAM> (TAB(VARSIZE) | REM) . VARSTR =
69  CONT1   DIFFER(LABEL<NAM> LABSTR)  :F(CONT2)
70          LABEL<NAM> (TAB(5) | REM) . LABSTR =
71  CONT2   DIFFER(BRANCH<NAM> BRSTR)  :F(CONT3)
72          BRANCH<NAM> (TAB(25) | REM) . BRSTR =
73  CONT3   DIFFER(FUNCTION<NAM> FUNSTR)  :F(CONT4)
74          FUNCTION<NAM> (TAB(30) | REM) . FUNSTR =
75  CONT4   LINCT = GT(LINCT,0) LINCT - 1  :S(PRT)
76          HEADING = '              VARIABLE OR REFERENCE BY NAME              '
    +           ' NAME          FUNCTION                    LABEL       BRANCH '
    +             'TO LABEL'
77          OUTPUT =
78          LINCT = 42
79  PRT     OUTPUT = CHAR LPAD(VARSTR,VARSIZE) ' ' CHAR ' ' RPAD(WORD,NAMESIZE)
    +           CHAR RPAD(FUNSTR,31) CHAR RPAD(LABSTR,6) CHAR RPAD(BRSTR,26)
    +           CHAR
80          CHAR = ' '
81          WORD = DIFFER(TRIM(OUTPUT))  :S(NXTLIN)F(PRTNAME)
82  END
```

VARIABLE OR REFERENCE BY NAME	NAME	FUNCTION	LABEL	BRANCH TO LABEL
0002	&STLIMIT			
0020 0021 0023	A			
	ANY	0011 0026		
	APPEND	0051 0053 0055 0056	0057	
0049 0054	BODY			
0052 0053	BR			
0006 0053 0071 0072	BRANCH			
	BREAK	0028 0032 0041 0049		
0049 0052	BRFIELD			
0071 0072 0079	BRSTR			
0024 0025 0026 0027 0031 0038 0039 0040 0041 0047	CARD			
0054 0055 0066 0079 0080	CHAR			
	CONTCRD		0037	0029
	CONT1		0069	0067
	CONT2		0071	0069
	CONT3		0073	0071
	CONT4		0075	0073
	CTRLCRD		0039	0026
	DEFINE	0012 0013 0014 0017		
	DEFNEW		0017	0014
	DELLIT		0028	0028
	DIFFER	0051 0055 0067 0069 0071 0073		

VARIABLE OR REFERENCE BY NAME	NAME	FUNCTION	LABEL	BRANCH TO LABEL
		0081		
	END		0082	0064
0003 0025 0040	FIELDWIDTH			
	FRETURN			0015
0008 0056 0073 0074	FUNCTION			
0073 0074 0079	FUNSTR			
	GETBR		0052	0053
	GETVAL		0054	0052 0055 0056
	GT	0019 0075		
0001 0076	HEADING			
	IDENT	0044 0047 0057 0064		
0057 0058 0059	IDENTIFIER			
0015 0042 0046	INPUT	0042 0046		
0049 0051	LAB			
0007 0047 0051 0069 0070	LABEL			
0069 0070 0079	LABSTR			
	LEN	0040 0045 0054		
	LGT	0021		
0075 0078	LINCT			
0027 0028 0029 0030 0037	LINE			
	LPAD	0031 0047 0050 0079		
0019 0060	MAXSIZE			

VARIABLE OR REFERENCE BY NAME	NAME	FUNCTION	LABEL	BRANCH TO LABEL
0019 0021 0022 0023 0062 0063 0064 0065 0067 0068 0069 0070 0071 0072 0073 0074	NAM			
0005 0010 0020 0022 0023 0057 0063	NAME			
0060 0061 0079	NAMESIZE			
	NEWNAME	0057	0018	
	NXTCRD		0024	0017 0032 0040 0041 0043 0046
	NXTLIN		0067	0081
	NXTNAME		0020	0021
	NXTSTAT		0032	0035 0038
0031 0038 0039 0048 0077 0079 0081	OUTPUT	0001		
0018 0020 0021 0022	PLACE			
	POS	0025 0026 0029 0034 0040 0041		
	PRINT		0060	0036 0047 0048
	PROCESS	0033	0049	
	PRT		0079	0075
	PRTNAME		0063	0081
0016	PUT			
0015 0016	READ	0024 0036	0015	
0025 0045 0049 0068 0070 0072 0074	REM			
	RETURN			0016 0023 0054 0058 0059
	RPAD	0079		

VARIABLE OR REFERENCE BY NAME	NAME	FUNCTION	LABEL	BRANCH TO LABEL
	RPOS	0045 0058		
	SETEND		0047	C044
	SIZE	0019		
	SPAN	0011 0041		
0030 0032 0037	STATEMENTS			
0032 0034 0049	STMT			
0031 0035 0047 0050	STMTCT			
0050 0058 0059	STNUM			
	SUBSTR	0047		
0041 0042 0043 0045 0046	SYSUNIT			
0004 0043 0044 0045	SYSUNITS			
	TAB	0068 0070 0072 0074		
	TABLE	0005 0006 0007 0008 0009		
0058 0059	TABULL			
0025 0031 0038 0039	TEMP			
	TOEOF		0036	0034 0036
	TRIM	0081		
	TSTEOF		0044	0024
0054 0055 0056	VAR			
0009 0055 0067 0068	VARIABLE			
0061 0068 0079	VARSIZE			
0067 0068 0079	VARSTR			

VARIABLE OR REFERENCE BY NAME	NAME	FUNCTION	LABEL	BRANCH TO LABEL
0011 0052 0054 0065 0079 0081	WORD			

REFERENCES

The following abbreviations are used:

CACM	Communications of ACM
JACM	Journal of the ACM
COMP J	Computer Journal
NACM	Proceedings of the national ACM conference

Aho, A.V. and Johnson, S.C. LR Parsing, ACM Computer Survey Vol. 6, 1974 pp 99-124.

Backus, J. The Syntax and Semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, in Proc. Int. Conf. Inf. Processing, UNESCO, June, 1959, pp 125-132. Reprinted in Ann Rev. in Automatic Programming, Vol 1, pp 268-291 (Pergamon Press 1961).

Barnett, M.P. Computer programming in English. Harcourt, Brace and World, Inc. 1969.

Bell, J.R. A new method for determining Linear precedence functions for precedence grammars, CACM Vol. 12, 1969, pp 567-569.

Bloomfield, L. Language. New York, Holt, Rinehart and Winston, 1933 and Lond. Allen and Unwin, 1935.

Brooker, R.A. Top-to-bottom parsing rehabilitation? CACM Vol 10, 1967 pp 223-225.

Chomsky, N. Syntactic Structures, The Hague: Mouton 1957.

Conway, F.L. Design of a Separable Transition diagram compiler, CACM Vol 6, 1963, pp 396-408.

Deremer, F.L. Practical translation for LR(K) languages, M.I.T. thesis Oct. 1969.

Deremer, F.L., Lecture Notes, Advanced course on compiler construction Techn. Uni. Munich. Mar. 4-15, 1974.

Dewar, R.B.K. SPITBOL version 2.0
Illinois Institute of Technology Feb. 12, 1971.

Floyd, R.W. Bounded Context Syntactic Analysis,
CACM Vol.7 1964 PP 62-67.

Floyd, R.W. The Syntax Analysis of Programming
Languages - a survey.
IEEE transaction on Electronic Computers
1964 PP 346-353.

FORTRAN IV Language IBM 360/370 (S360-25 GC28-6515-8)

Frieman, J. Direct Random Generation of Sentences,
CACM Vol.12 1969 PP 40-46.

Friedman, J. A Computer Model of Transformational Grammar.
Elsevier 1971.

Gries, D. Compiler Construction for Digital Computers,
Wiley 1971.

Griffiths, T.V. and Petrick, S.R. On the Relative
Efficiencies of Context - Free Grammar Recognizers.
CACM Vol.8 1965 PP 289-299.

Gaifman, H. Dependency Systems and Phrase Structure
Systems.
Information and Control Vol.8 1965 PP 304-337.

Griswold, R.E., Poage, J.F., Polonsky, I.P.
The SNOBOL4 Programming Language (second edition)
Prentice - Hall.

Haskell, R. Symmetrical Precedence Relations on General
Phrase Structure Grammars
Comp. J. Vol. 17 No.3 1974.

Hays, D.G. Dependency Theory - A formalism and some
observations lg. 40 1964 PP 511-525.

Hoare, C.A.R. Proof of a program FIND.
CACM Vol.14 Jan. 1971 PP 39-45 .

Hopcoft, J.E. and Ullman, J.D. Formal Languages
and their relation to automata.
Addison - Wesley Publishing Company 1969.

Horning, J.J. and Lalonde, W.R. Empirical comparison
of LR(k) and precedence parses.
Technical report c.s. RG-1 September 1970.
Computer Systems Research Group, University of
Toranto.

Irons, E.T. "Structural Connections" in
Formal Languages
CACM Vol.7 1964 PP 67-72.

Irons, E.T. Experiments with an extensible language,
CACM Vol.13 1970 PP 31-40.

Kanner, H. An algebraic translater.
CACM Vol.2 Oct. 1959 PP 19-22.

Knuth, D.E. On the translation of languages
from left to right.
Information and Control Vol.8 1965
PP 607-639.

Lewis II, P.M. and Stearns, R.E.
Syntax - Directed Transduction.
JACM Vol.15 1968 PP 465-488.

Lietzke, M.P. A method of Syntax-Checking ALGOL60
CACM Vol. 7 1964 PP 475-478.

Lyons, J. Introduction to theoretical Linguistics,
Cambridge University Press 1970.

Mana, Z. Properties of programs and the fixed
order predicate calculus.
JACM Vol.16 April 1969 PP 244-255 .

Mana, Z. and Waldinger, R.J. Towards
Automatic program synthesis.
CACM Vol.14 Mar. 1971 PP 151-155.

Martin, D.F. Boolean Matrix methods for the
detection of Simple precedence grammars.
CACM Vol. 11 1968 PP 685-687.

McCarthy, J. Recursive functions of symbolic expressions
and their computation by machine, Part 1.
CACM Vol.3 1960 PP 184-195.

Metcalfe, H. A parameterized compiler based on
mechanical linguistics.

Naur, P. Revised report on the algorithmic language
ALGOL 60.
Numerical Math. 2 1960 PP 106-136
and CACM Vol.3 1960 PP 299-314.

Paul, M. ALGOL 60 processors and a processor generator.
NACM 1962 PP 493-497.

Rabin, M.O. and Scott, D.
Finite automata and their decision problems.
IBM Journal of Research 3; 2 1959.

Rosenkrantz, D.S. and Stearns, R.E.
Properties of deterministic top-down grammars.
ACM symposium on theory of computing 1959
PP 165-180.

Sager, N. Syntactic analysis of natural languages.
Advances in computers Vol. 8 1967 PP 153-158.

Satterthwaite, C.A. Programming languages for
computational linguistics.
Advances in computers Vol. 7 1966 PP 209-238.

Scott, D. Outline of a mathematical theory of computation. Proceedings of the 4th Princeton Conference on information sciences and system Mar. 1970.

Solntseff & Yezerski. A Survey of Extensible Programming Languages, Annual Review in Automatic Programming Vol. 7, Part 5, 1974.

Unger, S.H. A global parser for context-free phrase structure grammars. CACM Vol. 11, 1968, pp 240-247.

Vigor, D.B. Urguhart, D. and Wilkinson, A., PROSE - Parsing recogniser outputting sentences in English. Machine Intelligence 4 1969 pp 271-284.

Wirth, N., Weber, H., EULER: A generalization of ALGOL and its definition: Part I. CACM Vol 9. 1966, pp 13-25.