A SYNTAX DIRECTED TRANSLATOR

A thesis presented by Abdul Khaliq Kadwa, B.Sc., to the University of St. Andrews in application for the Degree of Master of Science.

June 1969

DECLARATION

I hereby declare that the accompanying Thesis is my own composition, that it is based upon research carried out by me and that no part of it has previously been presented in application for a higher degree.

Signed.../ ...

CERTIFICATE

I certify that Abdul Khaliq Kadwa, B.Sc., has spent seven terms as a research student in the Computing Laboratory of the United College of St. Salvator and St. Leonard in the University of St. Andrews, that he has fulfilled the conditions of Ordinance 51 (St. Andrews) and that he is qualified to submit the accompanying Thesis in application for the degree of Master of Science.

Research Supervisor.

ACKNOWLEDGMENTS

My deepest gratitude goes without question to Mr. A.T.Davie both for bringing to my attention in November 1967 the subject matter on which this Thesis is a development and for his untiring surveillance of my work, without which a large part of this Thesis would never have been completed.

My thanks go to Dr. A.J.Cole for accepting me as a prospective Advanced Course student in Spring 1967 and for helping me in many ways during the preparation of this work.

Finally, my gratitude goes to Mr. H.Abrahamson for a discussion on Syntax-orientated compilers and of course to UNESCO for providing the Advanced Course Studentship.

CONTENTS

INTRODUCTION

This Thesis describes a program, based on the mechanism discussed by Metcalfe [A1], which performs indicated operations automatically to check syntax of various source languages and also translates a source language into a pre-determined target language. A mechanism, an improvement on that of Metcalfe or rather not discussed by Metcalfe, which in certain respects will speed up translation is also described.

The program written in the IBM 1620 symbolic programming language simulates a special purpose stored-program computer (Syntax machine [A1]). A program for this machine (syntax program [A1]) represents the syntax and semantics of some language to be translated. Since the syntax machine can be programmed, it can translate any number of source language-target language pairs, that is it is a parametrized compiler. Two sets of parameters have to be provided for the compiler in order to carry out a specific translation : a specification of the source language and a specification of the target language. Only one set of parameters is required if only syntax checking is to be performed, that is, the specification of the source language.

The source language description will be in an expanded form of constituent (phrase-structure) grammar. The target language description will be in a form which is consistent wi-

th and embedded in the source language description. This total specification of the source and target languages is referred to as a "grammar", and the notation in which it is written as a "meta-language". Thus the scope of translation is limited to those language pairs which are completely definable by phrase -structure analysis and synthesis. Such analysis can be referred to as "parsing" or analysis in terms of the syntax for a particular grammar. The synthesis is often called "unparsing".

CHAPTER 1

SYNTAX DIRECTED TRANSLATION

1A. Syntax Directed Translation

In this chapter a brief discussion is made of syntax-directed translation. No attempt is made to describe in detail any particular system of translation, but a general and brief discussion of compiler writing is attempted, since compilers contain all the essential problems found in assemblers and interpreters. No comparison of different methods of translation is attempted as there are as yet no accepted standards of performance for translators such as efficiency. "The efficiency of a compiler depends on its ability to conserve both time and space while translator and during execution of the object program. The ease of use, the error detection and recovery facilities, the editing facilities and the speed of recompilation have important effects on efficiency. As not all these goals are mutually compatible, one can expect no absolute measure of performance for compilers [A2]."

All translators are syntax-directed in the sense that the translator must obviously recognise the various syntactic structures and the output of the translator is a function of the syntax of the language [A3]." The phrase "syntax-directed" in the title refers to the method by which the translator is given the syntactic specification of the language it is to

compile. That is, rather than having the syntactic structure of the language reflected in the actual encoding of the compiler algorithm, a "syntax-directed" compiler contains (or uses, as parametric data) a relatively simple and direct encoding of the syntactic structure of the language, for example, as it might be expressed in Backus Form [A4]. By simple and direct encoding, it is meant, for instance numerical codes for the distinct syntactic types of the language and direct pointers representing the relations of concatenation and alternative choice [A5]".

Most translators produce an intermediate form of the program before producing the final machine code. Once this intermediate form is produced the distinction between syntax-directed translation and other methods disappears. The primary goal of the intermediate form is to encode the program in a form which is easily and efficiently processed by the computer. Most optimization algorithms, such as elimination of common sub-expressions [A9] and optimum evaluation of Boolean expressions, are much simpler when applied to some intermediate form rather than to the original expression or the final machine language version. The intermediate form exhibits the structure (that is which sub-expressions are the operands of each operator) and the order of evaluation of sub-expressions.

1B. Syntax Description.

Several essentially equivalent formalisms for the representation of syntax have been developed. These include:

Post Production Systems, developed by the logician Emil Post during the 1940's as a tool in the study of Symbolic Logic;

Phrase-Structure Grammars, developed by the linguist Noam Chomsky during the 1950's as a tool in the study of natural languages; and

Backus Normal Form, developed by the programmer John Backus during the late 1950's as a tool in the description of programming languages.

In order to unambiguously state a set of syntactic definitions, an expanded version of the "Backus Normal Form" of constituent (phrase-structure) grammar notation is used in this thesis. A Syntactic Specification of a language is a concise and compact representation (description) of the structure of that language. It is merely a description of structure and does not by itself constitute a set of rules either for producing allowable strings in that language, or for recognising whether or not a proffered string is, in fact, an allowable string.

However the rules are formulated to produce or recognise, strings according to the specification. In a syntax-directed compiler it is an algorithm which performs the recognition of allowable input strings, and it does this by using the Syntax Specification as data. Such an algorithm is called an "Analyser".

In the discussion of the structure of a language, strings in that language may be classified by "syntactic types". Some of these classes consist only of a single character in the

source language - these are called "Terminal Types" ("terminal characters"). Most syntactic types are more complicated in structure and are defined in terms of other classes. Such types are called the "Defined Types" and also referred to as "meta-linguistic variables" (abbreviated as meta-variables). Backus notation represents the grammar of a language as a set of definitions of grammatical structures. Each definition has three components:

(i) the name of the structure being defined (defined type or meta-variable),

(ii) the symbol "::=" which is read "is defined to be", and

(iii) an expression which specifies the permitted forms of the structure.

The symbol "::=" belongs to the meta-language, not the language whose grammar is being defined, and so is termed as a meta-symbol. Syntactic variables or names of the structures being defined may be spelt using symbols which do belong to the language being defined, but are enclosed in angled brackets "< >", called meta-brackets. An expression is a list of alternative permitted forms of a structure. Pairs of alternatives are separated by the meta-symbol "|" which is read as "or". Syntactic constants (for example terminal characters) are usually denoted by themselves; for example

< assignment> ::= < variable> = <arith expr> .

The Defined Type on the left of the "::= " is called the Defined Type of the Definition; and the Definition is said to be a Definition of its defined type. The right hand side of the Definition is said to be a construction with three components, which are, in order of their appearance, the Defined Type <variable>, the Terminal Character "=", and the Defined Type <arith expr> . This definition states that among the strings of the source language belonging to the defined type of the definition, are concatenations of sub-strings such that each sub-string belongs to the syntactic type named by the corresponding component.

A complete set of grammatical rules for a language written in the format equivalent to that of the example above is a Phrase-Structure Grammar; a language definable by a Phrase-Structure Grammar is a Phrase-Structure Language [A6].

In general, a phrase-structure grammar, taken as a set of definitions, provides a list of alternative constructions in a definition for each syntactic type, where each construction is a list of characters and syntactic type names. A construction represents the set of phrases which can be formed by replacing each syntactic type name with a phrase of that type; the phrases of a certain type are all those represented by some construction in the definition of that type. Ther is usually a single syntactic type, called "program", which is used in the definition of no other type; the set of phrases of this type is the language defined by the grammar [A6].

1C. The Analyser.

Given a synthetic phrase-structure grammar one can randomly generate sentences by deriving them and their syntax trees from the distinguished syntactic type "program". Compilers on the other hand have the opposite problem: given a sentence x and a grammar G, construct a derivation of x and find a corresponding syntax tree [A2]. This is called parsing, recognising or analysing the sentence.

There are two techniques, in common use, of parsing or analysing in syntax-directed translators, called "top-down" and "bottom-up". These techniques are described diagramatically in Chapter 2, but their mechanism is discussed here.

Syntax-directed analysis by computer is made possible by the concept of "goals", first introduced in this context by Irons [A7]. A syntactic type is interpreted as a goal for the analyser to achieve, and the definition of a defined type is considered to be a condition for achieving the goal of the defined type.

The pure top-down analyser is entirely goal-orientated. The main goal is the distinguished non-terminal symbol "program". It sets up a goal and tries all possible ways of achieving that goal before giving up and replacing the goal with an alternative. New sub-goals are continually being generated and attempted. If a sub-goal is not met, failure is reported to the next higher level, which must try another alternative. Left recursion some-

times causes trouble in left-right, top-down analysers. This is overcome by changing the grammar or modifying the analyser. A top-down analyser may be programmed in many different ways - as recursion subroutines, as a single routine with stack to store links [A8], etc.

A pure bottom-up analyser has essentially no long-range goals except the implicit goal "program". "The bottom-up analyser having recognised a Syntactic Type, checks whether it has gone astray in trying to reach its goal or whether that Type is indeed a possible first Component of a first Component of...of the goal. If the latter, it continues processing input until it has built another Type of which the previous one is a first Component, and goes back to the checking. If it has gone astray, it backs down and tries to see if it can construe the input differently, to approach its goal along a different chain of intermediate types" [A5].

The pure bottom-up analyser, like the pure top-down analyser will normally make links which turn out to be incorrect. This may be corrected in two different ways. The first method is to back up to a point where another alternative may be tried. This involves restoring parts of the string to a previous form or erasing some of the connections made. The second method is to carry out all possible parses in parallel [A13, A14]. As some of them lead to "dead ends", they are dropped.

In order to reduce the probability of making incorrect reductions, more sophisticated analysers have been developed. For instance, before starting out on a new sub-goal, a modified top-down recogniser might look in a pre-constructed table to see whether some derivative of the sub-goal can actually start with the initial symbol of the sub-string in question (a look-ahead table), or whether the sub-goal being attempted could occur in the partial tree formed so far in memory [A7, A9].

1D. Some Syntax-Directed Translators.

In this section a few practical techniques for analysing and translating sentences of languages definable by grammars are discussed.

1D.1. A Table Driven Compiler: Stephen Warshall and Robert M. Shapiro [A9], are responsible for developing a General-Purpose Table Driven Compiler, which is designed to generate efficient object code by several different kinds of optimization phases. The compiler is in the form of a general - purpose table driven program for getting from syntax trees to macro-instructions. The compiler is composed of five phases:

1. A syntactic analyser (modified top-down) which converts a piece of input string into a tree-representation of its syntax.

2. A generator, which transforms the tree into a sequence of n-address macro-instructions, investigating syntactic context to decide the emission.

3. An "in-sequence optimizer" (ISO) which accumulates

macros, recognises and eliminates the redundant computation of common sub-expressions, moves invariant computations out of loops, and assigns quantities to special registers.

4. A code selector which transforms macros into symbolic machine code, keeping a complete track of what is in special registers at each stage of the computation.

5. An assembler which simply glues together the code syllables in whatever form is required by the system with which the compiler is to live: symbolic, absolute, or relocatable, with or without symbol tables etc.

The first four phases are encoded as general-purpose programs; the fifth is handled as a special-purpose job in each version of the compiler.

The analyser is of the "top-down" syntax-directed variety, driven by tables which are in effect an encodement of the source language's syntax as given by a description in the meta-linguistics of the Algol 60 report [A4] . The analyser tables contain some information which is not syntactic, but rather concerned with compiler control (for example, how much tree should be built before the generator is called) or with specifying additional information to be placed in the tree for later use by the generator.

The generator algorithm "walks" through the tree from node to node, following directions carried in its tables. As it walks, macro-instructions are emitted from time to time, also as indicated

in the tables.

The ISO accepts macro-instructions emitted by the generator. The processing of a macro usually results in placing the macro in a table and sending a "result" message back to the generator. The processing of a macro is controlled by a table of macro descriptions. If a macro may be handled as (part of) a common sub-expression and is not computable at compile time, the ISO will recognise a previous occurrence of the same macro as equivalent if none of its arguments have been changed in value. At intervals the ISO performs "global" optimizations over the region of macros just completed.

The code selector produces symbolic machine code for a region of macros after the ISO has collected these macros and performed its various optimizations. The code selector is driven by a table of code selection strategy. The code selector views the macros as nodes of a tree structure; that is, certain partial orderings exist which guarantee that the code emitted preserve the computational meaning of the macro, but within these constraints the strategy can order the computation according to its convenience, as a function of the availability of registers and results.

The assembler is hand tailored for particular implementations and is therefore not discussed.

1D.2. The Syntax-Directed Compiling Technique of G.E.Millard [A10].

Millard's syntax-directed compiling technique is discussed because it can be applied directly to any programming language which can be described by the Backus notation. He has implemented the Mercury Autocode on the Elliot 503, using this technique. Millard's translator consists principally of three algorithms which require as initial data the syntax definitions and associated target language constructions appropriate to the desired translation. Figure D.21 shows the basic flow chart of the translation process which may be considered as consisting of three stages:

1. The syntax rules processor accepts as input the syntax definitions and associated constructions and codes them into tabular form for use by the source language analyser (SLA) and the target language generator (TLG).

2. The SLA takes as input the source language string to be translated and produces a tree-like structure which represents the analysis of the string.

3. The TLG uses the information contained in the tree-structure to generate the appropriate target language string.

When the syntax definitions for a particular implementation has been fully developed the first stage is dispensed with, the tables then being permanent.

The source language analyser is a top-down analyser. Target language generation is specified in the syntax definitions by adding a string of characters enclosed in the meta-brackets "{ }".

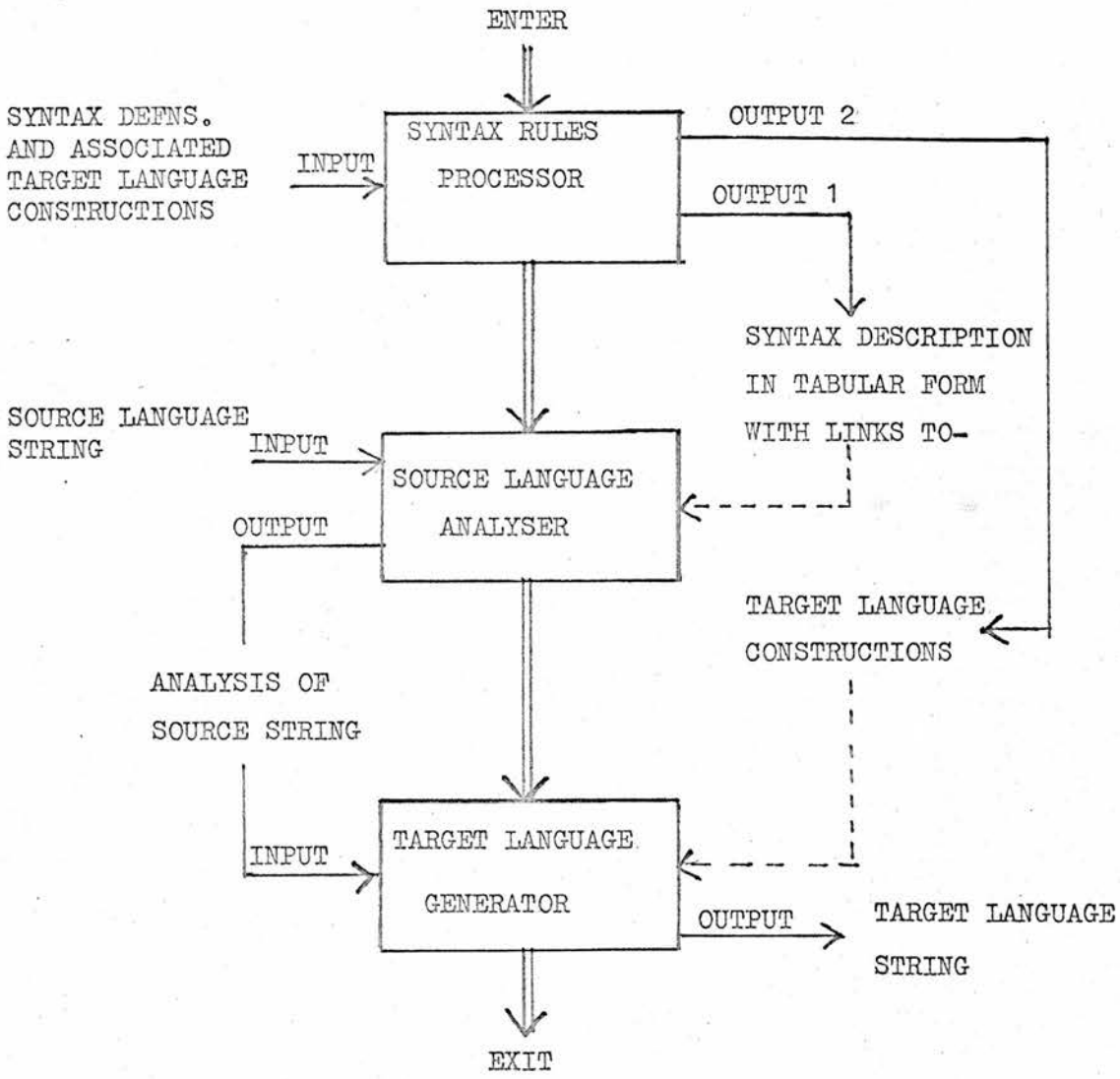


FIGURE D.21 BASIC FLOW DIAGRAM OF TRANSLATOR

This string of characters represents the "construction" in the target language associated with that definition. The characters in the strings are of two types: characters which are directives to the TLG, and terminal characters belonging to the target language. In the example below there is one special character, ρ.

< assignment> ::= < variable> = < arith expr> $\{\rho_1 \rightarrow \text{STO}\rho_2\}$...D.22

During the processing of a target language construction string any terminal characters encountered are output immediately; the character $\rightarrow$ is output as "new line". When a special character is encountered, followed by a digit n, this is interpreted as "process the node associated with the n-th meta-variable form the right in the corresponding syntax definition" - a concept introduced by Irons [A7] . During the processing of the above construction the actions of the target language generator would be:

1. process the target language construction with the definition of < arith expr > found to be satisfied, outputting any terminal characters generated thereby;

2. output on a new line the characters STO ;

3. process the target language construction associated with the definition of < variable > used in the analysis.

Thus ρ_1 refers to the right-most branching node of the node < assignment > and ρ_2 to the node on the second branch (numbering from right to left) of < assignment > .

The translation process is controlled through syntax definitions. This is done through ↑mn routines, where mn is a two digit integer and ↑ is a special character. As an example three routines are mentioned which perform the following actions:

↑10 Read a line of source language

↑11 Enter TLG as a sub-routine

↑17 Stop

If the syntax definition

< program> ::=↑10 < assignment>↑11 ↑17{ } D.23

is now prefixed to the definition D.22, the analysis proceeds as follows (the first component of the definition being the starting type):

↑10 is interpreted as a directive to enter ↑ routine 10, which reads a line of source language. The routine then returns control to the analyser which proceeds to search for the next component, < assignment> , as a sub-goal. The analysis proceeds until the definition of < assignment> is satisfied. The next component of the definition of < program> ,↑11, results in an entry to ↑ routine 11 and thus to the TLG, producing the target language code corresponding to the preceeding analysis. On exit from this routine the next component causes an entry to be made to ↑ routine 17, terminating the translation. The meta-brackets "{ }" are always present but they enclose a null string if no target language construction is required.

The introduction and use of π routines in the syntax definitions is probably the most novel feature of the technique described, and it may be convenient to make use of a large number of π routines in the translation of a complete programming language.

The syntax rules processor which converts rules of syntax into a tabular form is not discussed as a technique for converting syntax rules is described in the latter part of this thesis.

1D.3. A One Pass Algol Compiler. The syntax analyser and its coding mechanism of the one pass Algol compiler written by H.Kanner et al [A11] is discussed particularly because the analyser routines are surprisingly easy to write and modify. The structural detail of the compiler is not discussed. Their analyser algorithm is based on a singularly simple method of analysis suggested by A.E.Glennie [A12].

The compiler proper accepts as input a string of characters coded in an internal representation. Each basic symbol is uniquely represented by a single character. The translation from the hardware representation to the internal character set is the responsibilty of an input routine.

The syntax analyser is composed of a number of modules called recognisers. A given recogniser is a routine for identifying a specified phrase class. It has a principal exit, taken if the phrase class is successfully identified, and one or more alternate exits, taken upon failure to find the phrase class.

Recognisers themselves may be composed of recognisers and elementary building blocks called comparators. A given comparator examines the current input character (internal code), which is found in a standard location. If the current character is the basic symbol corresponding to that comparator, the next input character is transmitted to the standard location and a "true" exit is taken; otherwise a "false" exit is taken. The exit address for a recogniser is stored in a push-down list at the time of entry in order to enable a recogniser to call itself recursively.

In figure D.31 are shown recognisers that correspond to the definitions of digit, unsigned integer, and integer, given below:

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<unsigned integer> ::= <digit> | <unsigned integer> <digit>

<integer> ::= <unsigned integer> | + <unsigned integer> |

- <unsigned integer>

The convention used for exits from components of a recogniser is that true exits are drawn horizontally to the right, and false exits vertically downward. The exits that are explicitly labelled "true" and "false" apply to the entire recogniser.

The examples in figure D.31 illustrate two characteristics of this form of syntax analysis. The first is seen in the recogniser for < unsigned integer > , in which the recursive definition given in the text has to be replaced by an iterative

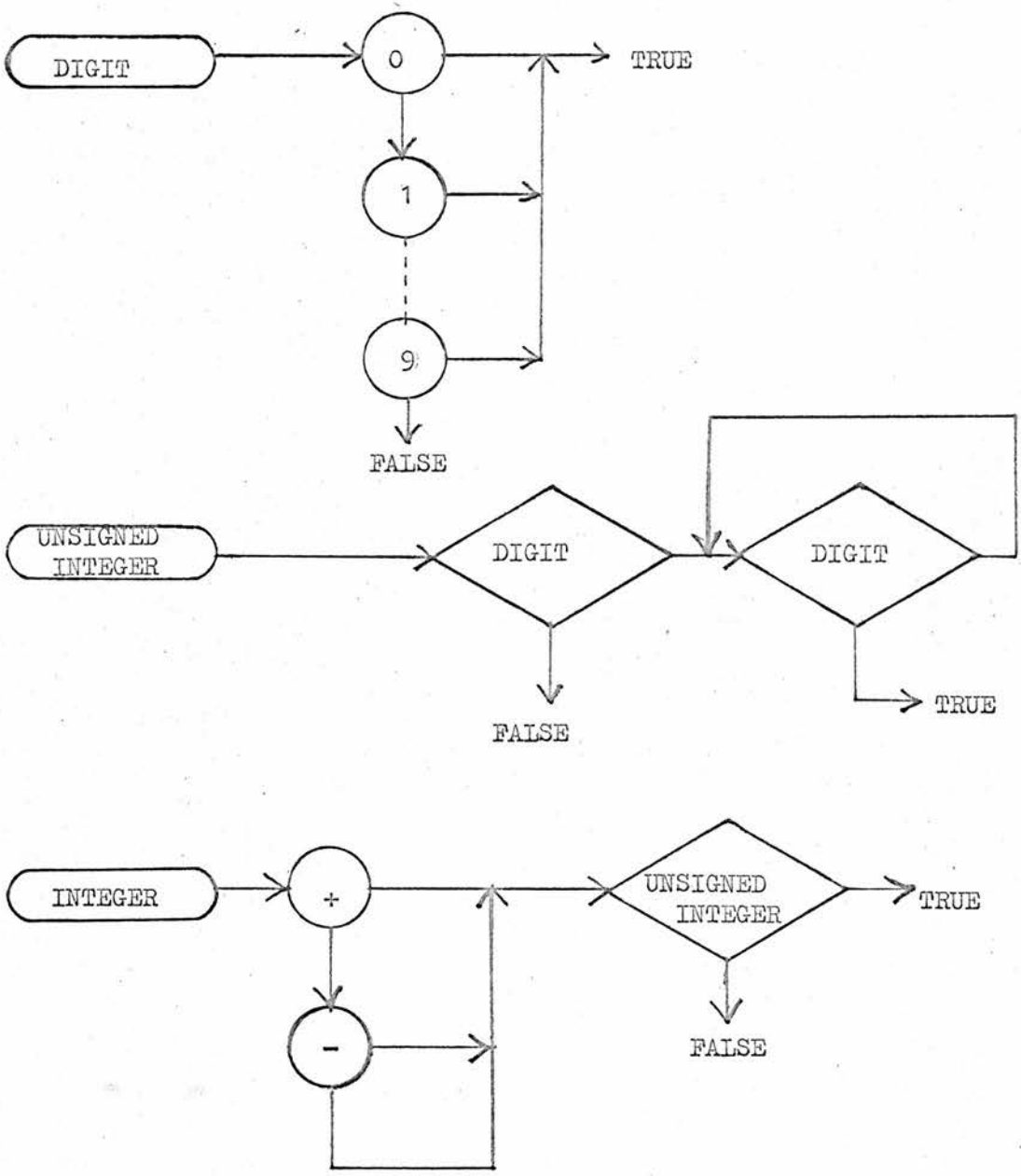

DIGIT
0
1
9
TRUE
FALSE
UNSIGNED INTEGER
DIGIT
DIGIT
TRUE
FALSE
INTEGER
+
-
UNSIGNED INTEGER
TRUE
FALSE

FIGURE D.31

definition. This is a requirement of any method of top-down analysis. The second characteristic is illustrated in the < integer > recogniser. A "false" exit may be taken from < integer > as a consequence of failure to find an unsigned integer after having found a plus or minus sign.

If < integer > was the first of several alternatives in a recogniser, it would be necessary upon false exit from < integer > to back up the input string to where it had been at the time of entry to < integer > . The addition of means of output generation to the recogniser further complicates the situation. In essence, upon false exit from a recogniser and prior to entry to an alternative one, the entire system must be restored to the state in which it had been before the initial entry. This is usually achieved easily by the use of push-down lists.

Generation routines for the production of output can be easily imbeded into the syntax analyser. Figure D.32 illustrates an analyser capable of transforming a polynomial from conventional infix notation into Polish suffix notation. This analyser is based on the syntax:

< polynomial > ::= < term > | < polynomial > < adop > < term >

< adop > ::= + -

< term > ::= < factor > | < term > * < factor >

< factor > ::= < variable > | (< polynomial >)

< variable > ::= A | B | C | ..... | Y | Z

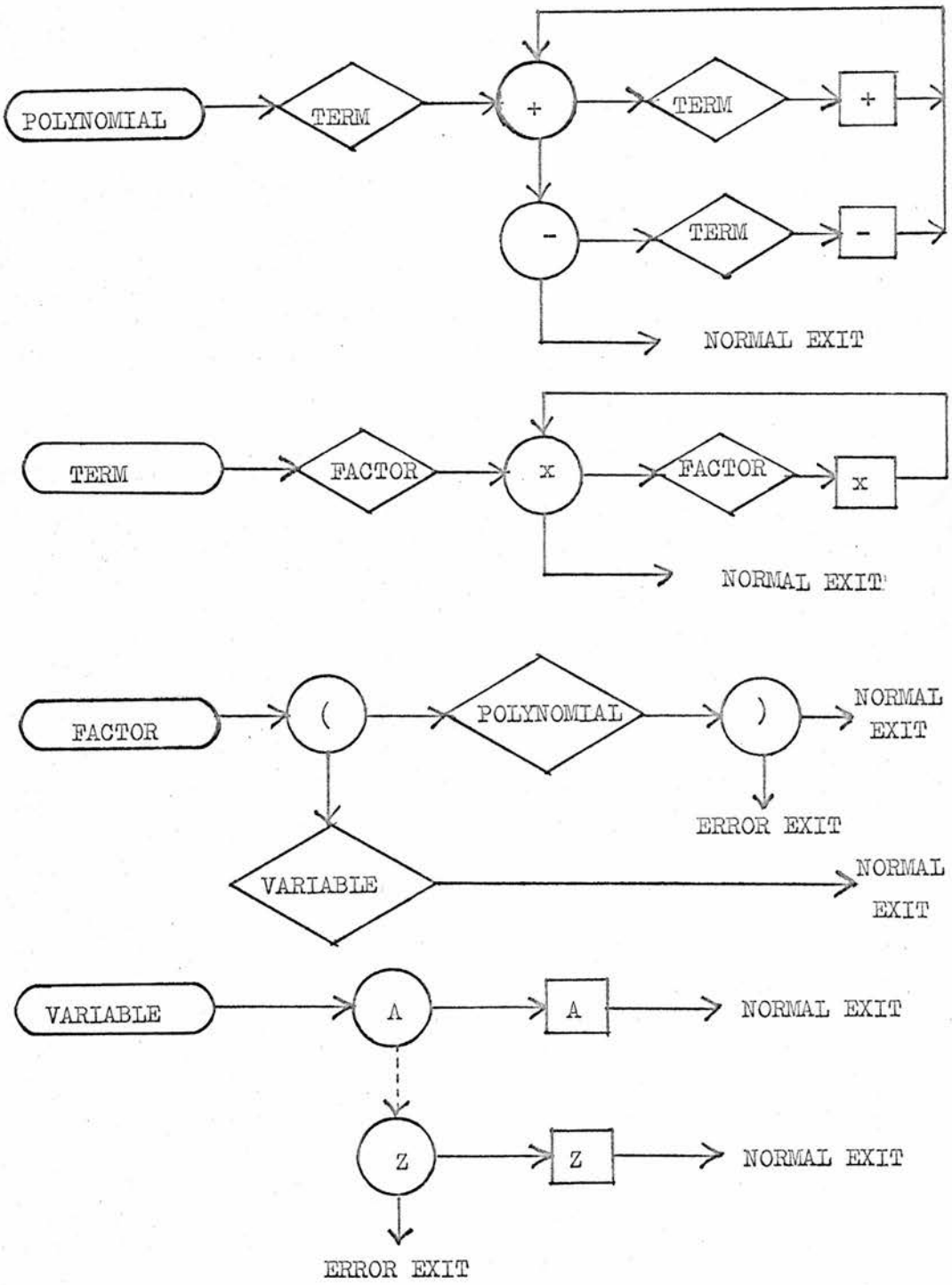

POLYNOMIAL
TERM
+
TERM
+
-
TERM
-
NORMAL EXIT
TERM
FACTOR
x
FACTOR
x
NORMAL EXIT
FACTOR
(
POLYNOMIAL
)
NORMAL EXIT
ERROR EXIT
VARIABLE
NORMAL EXIT
VARIABLE
A
A
NORMAL EXIT
Z
Z
NORMAL EXIT
ERROR EXIT

FIGURE D.32

The square boxes in figure D.32 denote output generators. If their only action is to emit into an output string the indicated symbol, then the transformation to Polish suffix form will be accomplished.

Two techniques used are worthy of note. The first is that of representing every basic symbol of Algol by a single character in the internal code of the compiler. Several advantages accrue from the use of this strategy. Back-up problems in the recognition of basic symbols are reduced in severity, simplifying, among other things, the design of schemes for input buffering. The compiler is not dependant upon the choice of hardware representation, the conversion to internal code being performed solely within an input routine.

A second and more significant technique lies in the heavy use of macro-instructions in encoding the compiler itself. They have used about forty macro-instructions, representing commonly used operations within the compiler, to implement it. This set of macros in effect provides a "language" for description of the compiler. All the recognisers and associated generators have been encoded as sequences of macro calls.

The syntax-directed translator which is the subject of this thesis is based on similar techniques of analysis, output generation and compiler coding techniques.

CHAPTER 2.

AUTOMATIC PARSING.

2A. Diagramming.

Before "top-down" or "bottom-up" scanning methods are considered in detail it would be worth while to consider the technique of "diagramming". An English sentence can be parsed by the technique of diagramming. For example, the sentence "JACK LIKES LEMONS" may be diagrammed as shown in figure 1.

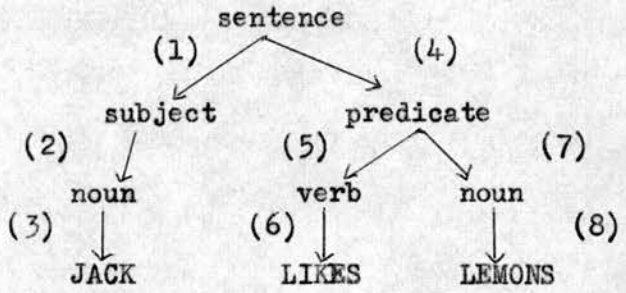


FIGURE 1.

The sentence chosen above is a particular instance of an English It is a simple sentence broken up into a very few parts. This is done in order to make figures representing the diagramming of the sentence as simple as possible.

The diagram in the figure above displays the syntax (grammatical structure) of a sentence in a "tree" fashion. This representation may termed a "syntax tree". Each node represents a "constituent" (or "phrase") of the syntax. The upper-case words are the "ultimate constituents" (or "basic words") and the

lower-case words the "syntactical constituents" of the sentence.

In general, a syntax tree is like a geneological tree for a family whose common ancestor is sentence, where the immediate descendants (sons) of a symbol form one of the alternatives of the definition of that synbol and where only the terminal characters fail to have descendants. Such a tree represents a derivation of the sentence formed by its terminal characters. It also illustrates the structure of the sentence; the terminal descendants of any node of a tree form a phrase in the sentence, of the type designated by that node. In a language satisfactorily described by its grammar, the phrases of a sentence are its meaningful units. Some compilers take advantage of this, creating a syntax tree as a structured representation of the information contained in the source program. Suitable processes then translate the tree into a computer program, or a derivation tree for an equivalent sentence in another language or a related sentence in the same language [A6].

The syntax of the type of the sentence in figure 1 may be described in a more elegant fashion :

(1) sentence ⟶ subject + predicate

(2) subject ⟶ noun

(3) predicate ⟶ verb + noun

(4) noun ⟶ JACK, LEMONS

(5) verb ⟶ LIKES

The arrows separate a syntactical constituent name from its definition in terms of other constituents. The plus sign connects constituents which are defined to appear contiguously. The comma separates alternative definitions of a constituent. The set of definitions given above comprise the "grammar" of this type of sentence, and the notation used to describe the grammar is the "meta-language"; "subject", "noun" etc. are called "meta-components" and "JACK" etc. the basic words or datum.

In order to parse automatically two techniques are necessary:-

(1) Matching: or starting with the ultimate constituents (basic words) and working up the syntax tree;

(2) Searching: or starting with the final constituent ("sentence" in the example) and working down the syntax tree.

2B. Matching.

Matching ("bottom-up") examines the ultimate constituents and attempts to transform them into phrases by matching them against the definitions in the grammar. The resulting constituents are matched again against the definitions and transformed into higher order constituents, and so forth until the root constituent is reached. The following series of figures demonstrate the matching technique in parsing the sample sentence.

sentence

JACK LIKES LEMONS

FIGURE 1.1. Beginning of the parsing process. The symbol "JACK" (ultimate constituent) is at hand and the goal is "sentence".

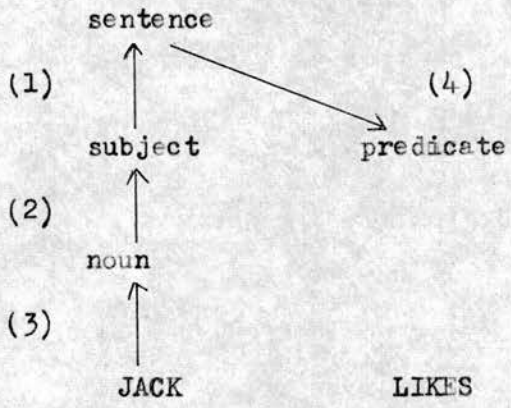


FIGURE 1.2. A chain with single meta-components has been found from "JACK" to "sentence", but however, the definition at the head of the highest order constituent has a second meta-component which is an element of "sentence", and a new parsing is required using "LIKES" as the symbol at hand and "predicate" as the goal.

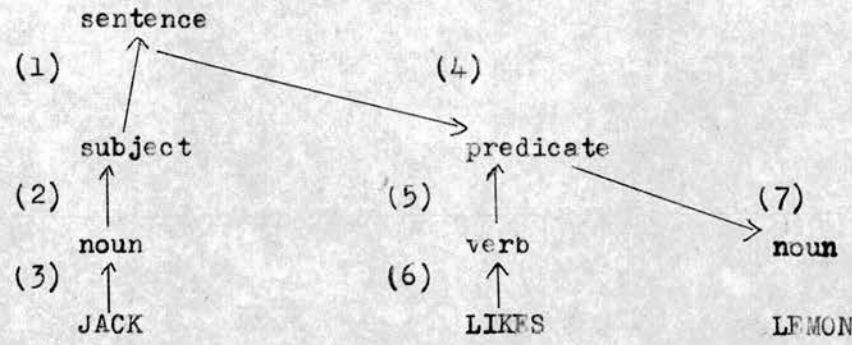


FIGURE 1.3. A definition chain from "LIKES" to "predicate" has been established. The definition at the head of the chain has an additional meta-component and a new parsing is required with "LEMONS" as the symbol and "noun" as the goal.

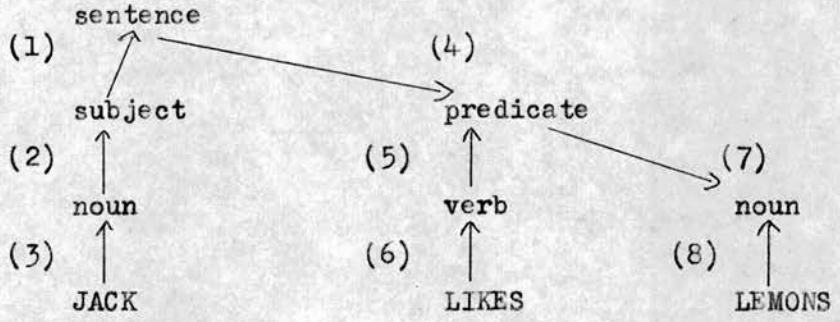


FIGURE 1.4. The definition chain desired for figure 1.3 consisted of a single definition. The parsing is completed and there are no definitions with unlocatable meta-components.

The trace of the matching process is equivalent to the syntax tree of the sample sentence in figure 1.

In order to match "noun", the bottom-up analyser must have the knowledge of a constituent other than that immediately involved in the matching. It must know whether the constituent

preceeding the noun is a verb or not, in order to decide whether the proper transformation is to subject or predicate as shown in figure 1.5.

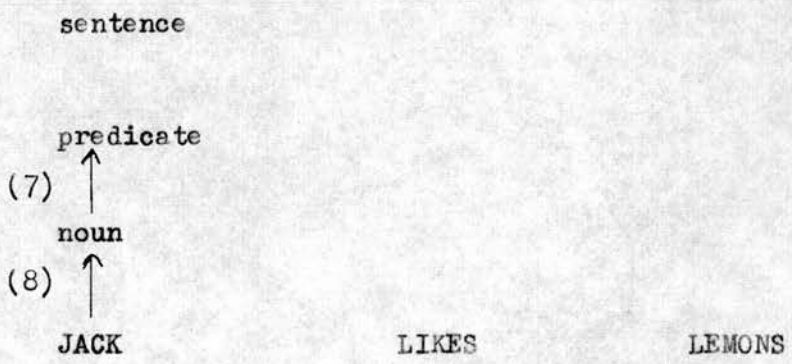


FIGURE 1.5. Working up through the incorrect definition chain will stop at the end of rule chain (8)-(7) as "predicate" cannot be the left part of "sentence". Matching from "noun" upwards will begin again and the rule chain (3)-(2)-(1) will have to be established as in figure 1.2.

Figure 1.5 reflects an inherent "problem" with the matching technique. Acceptability tables of rule chains have to be set up. In some grammars, in fact, it may be impossible to match many constituents without knowledge of all other constituents or without expanding the grammar to an unmanagable size.

Before a grammar can be used efficiently by a parsing processor, it is desirable that rules which represent alternative choices at each stage of the parsing be grouped together in the grammar; this reduces the amount of time necessary for searching the grammar for the applicable rule [A15]. In the example the alternative steps in figure 1.5 should, in practice, not occur

if the definitions are properly ordered in the rule table.

2C. Searching.

"Searching" is essentially a heuristic procedure, that is, it makes use of a complicated hierarchy of sub-ordinate goals in seeking its principal goal. It is a procedure which establishes goals and sub-goals and attempts to relate its environment to these goals by manipulating it in a trial and error fashion.

As applied in the case of the sample sentence, it is required to establish "sentence" as the final goal. A search is made down through the syntax tree, establishing each constituent as a sub-goal, until the ultimate constituents can be matched against the given sentence. All possibilities are examined (that is, all the branches of the tree), until the ultimate constituents have been successfully matched (in order) with the given sentence or there has been a failure to do so.

In order to achieve this, the grammar may be re-written in the form of a search procedure, or a program, in which each definition is used as a closed sub-goal.

Temporarily the following definitions are used for writing the program:-

(i) a variable called "flag" which can be set to a true or false value.

(ii) the following computer like operations:

"Find" establishes its parameter as a sub-goal and searches for it.

"Stop" halts the parsing process and displays the value of the "flag".

"Report" returns a value of the "flag" to the super-goal routine indicating whether the sub-goal was achieved or not. It also transfers control to the operation following the most recent "Find" operation.

"Match" sets the "flag" to a true value if the parameter symbol matches the symbol under scan, otherwise it is set false. If a match occurrs the next symbol (word) is placed under scan. Similar to "Find" but the sub-goal is a basic word.

"Check-F" "Report" if the "flag" is false.

"Check-T" "Report" if the "flag" is true.

The syntax program for the definition of the sample sentence is shown below. Execution of the syntax program begins at the first instruction.

```
            Find        sentence
            Stop
sentence:   Find        subject
            Check-F
            Find        predicate
            Report
```

```
subject:    Find      noun
            Report
predicate:  Find      verb
            Check-F
            Find      noun
            Report
noun        Match     JACK
            Check-T
            Match     LEMONS
            Report
verb:       Match     LIKES
            Report
```

In the program above the main goal is "sentence" which has two immediate sub-goals,namely, "subject" and "predicate". "Subject" has a single sub-goal while "predicate" has two immediate sub-goals. "Noun" and "verb" are sub-goals having no goals to call on. "Subject" and "predicate" have sub-goals, are sub-goals and have a super-goal, namely, "sentence".

A trace of the execution of the parsing program when presented with the sample sentence "JACK LIKES LEMONS", might appear as follows (results after execution of an operation are shown on the right hand side, when necessary):

```
Find      sentence
Find      subject
Find      noun
Match     JACK        set flag true and advance one word.
Check-T               flag true - "Report" to subject.
Report                return from subject to sentence.
Check-F               flag true - do not "Report".
Find      predicate
Find      verb
Match     LIKES       set flag true and advance one word.
Report                return from verb to predicate.
Check-F               flag true - do not "Report".
Find      noun
Match     JACK        set flag false.
Check-T               flag false - do not "Report"
Match     LEMONS      set flag true and advance one word.
Report                return from noun to predicate.
Report                return from predicate to sentence.
Report                return from sentence.
Stop                  display flag which is true.
```

The trace shows that a single left-to-right scan has been made of a simple sentence. The trace of the parsing is essentially equivalent to the "diagram" of the sample sentence.

The parsing-tree which is developed by the "top-down" approach is intuitively more satisfying. A rule is added to the

sub-tree only when it has been shown that it applies; hence the parsing tree, although possibly incomplete, is always connected [A15].

No knowledge of the constituents other than those immediately involved is required, since such information (for the entire sentence, in the example) is embeded in the sequence of the sub-goal calls. No ordering of definitions is required, since each definition is treated as a closed-sub-routine. Compared to the "matching" technique described previously, only the minimal amount of grammar scanning is done.

The parametrized compiler which is described in the following chapters uses the "searching" technique for analysis. Also all terminal characters used in the following chapters, form part of the IBM 1620 character set [A16].

The sample grammar, used in this chapter, is now re-written in the Backus notation as follows:-

```
<sentence>  ::= <subject> <predicate>
<subject>   ::= <noun>
<predicate> ::= <verb> <noun>
<verb>      ::= LIKES
<noun>      ::= JACK | LEMONS
```

Any reference to the "grammar defined in chapter 2" from any following chapters is a reference to the set of definitions above.

CHAPTER 3.

THE SYNTAX MACHINE.

3A. The Syntax Machine.

This chapter describes a simple process for determining whether a source text is grammatical in a language whose syntax is specified as has been described in the previous chapters. The process is performed by a special-purpose stored program computer ("syntax machine").

The data for this machine is a deck of cards (considered internally to be an ordered stream of symbols) on which the symbol strings to be parsed are inscribed. The machine has a set of operations which perform the functions necessary for parsing. These operations will be described in detail later. The machine has an instruction memory to contain the "program" of operations ("parser") to be performed in parsing the input symbol string (that is, the grammar program). It also has a "stack" memory (that is, push-down, pop-up) to contain information about the source input symbol positions and sub-routine exits. A variable register "flag" is used to denote a true or false indication. The program is an ordered stream of instructions which are normally obeyed in sequence, execution beginning with the first lexicographic instruction. Both the data and the program are scanned one element at a time. The two points of scan may be moved in either direction. Their positions may be recorded

in the stack memory and subsequently reset if desired. The machine has two special registers, the "ISP" (input symbol pointer) and the "PAR" (parser address register). The ISP points to the current symbol under scan and the PAR contains the address of the next parser instruction to be executed.

The operations of the syntax machine are of the one address type. The basic instruction word (in symbolic form) consists of three parts, namely:-

Label --- name of the location of an instruction.

Function --- the operation to be performed.

Data --- a label, or a basic symbol, or null;

example:

NOUN MATCH JACK

"NOUN" is the label of the above instruction, "MATCH" the function and "JACK" the basic symbol.

Initially the points of scan are set to the leading elements of the input data stream and the parser, the stack is empty and the flag is set to a true value.

The control cycle of the machine consists of inspecting the instruction of the program specified by the PAR, carrying out the indicated operation which will include changing the PAR (normally to the position of the successor sequence), and returning to commence a new cycle.

Initially the following functions are defined:

CALL — records the value of the input symbol under scan, that is, the value of the ISP, in the stack. It places the data part of the instruction into the PAR. If the data part is null then the location of the successor instruction is placed into the PAR. The value of the PAR is recorded in the stack and the flag is set to a true value.

MATCH — matches the symbol in the data part of the instruction with the input symbol currently under scan, that is, as indicated by the ISP. If a match occurrs, the value of the ISP is advanced by one position and the flag is set to a true value. If the symbols do not match the flag is set to a false value.

TRUE — if the flag is true then the data part is copied into the PAR. If the flag is false then the flag is set true and the value of the ISP is restored from the top of the stack without altering the stack.

FALSE — if the flag is false the data part is copied into the PAR.

NEXT — the flag is set true and the value of the ISP advanced to the successor position.

NOT — the setting of the flag is reversed and the value from the top of the stack is restored to the ISP. If the flag is now true, the value of the ISP is advanced to its successor position. The top of the stack is deleted.

FLAG — if the flag is true then the address part of the instruction is copied into the PAR otherwise the flag is set true and the location of the next instruction is copied into the PAR.

RETURN — if the flag is false the value of the ISP is restored from the top of the stack. The value from the top of the stack is copied into the PAR and the top set of values are deleted from the stack.

STOP — the value of the flag is displayed and the syntax machine is stopped.

It should be noted that "FLAG" is a function while "flag" is a variable.

3B. Programming The Syntax Machine.

A syntax program, using the set of operations defined above, written for the same grammar defined in chapter 2 might be written as shown in figure 2.

Location	Label	Function	Data
0		CALL	SENTENCE
1		STOP	
2	SENTENCE	CALL	SUBJECT
3		FALSE	A1
4		CALL	PREDICATE
5	A1	RETURN	
6	SUBJECT	CALL	NOUN
7		RETURN	
8	PREDICATE	CALL	VERB
9		FALSE	C1
10		CALL	NOUN
11	C1	RETURN	
12	NOUN	MATCH	JACK
13		TRUE	D1
14		MATCH	LEMONS
15	D1	RETURN	
16	VERB	MATCH	LIKES
17		RETURN	

FIGURE 2.

The first instruction to be executed is at location 0. For the time being each basic symbol in the grammar is assumed to be a single symbol.

A trace of the execution of the program in figure 2 when presented with the sentence "JACK LIKES LEMONS" might appear as in figure 3.

PAR	Flag	Return Stack	Input Stack	ISP	Symbolic Instruction	
0	T	-	-	1	CALL	SENTENCE
2	T	1	1	1	CALL	SUBJECT
6	T	1,3	1,1	1	CALL	NOUN
12	T	1,3,7	1,1,1	1	MATCH	JACK
13	T	1,3,7	1,1,1	2	TRUE	D1
15	T	1,3,7	1,1,1	2	RETURN	
7	T	1,3	1,1	2	RETURN	
3	T	1	1	2	FALSE	A1
4	T	1	1	2	CALL	PREDICATE
8	T	1,5	1,2	2	CALL	VERB
16	T	1,5,9	1,2,2	2	MATCH	LIKES
17	T	1,5,9	1,2,2	3	RETURN	
9	T	1,5	1,2	3	FALSE	C1
10	T	1,5	1,2	3	CALL	NOUN
12	T	1,5,11	1,2,3	3	MATCH	JACK
13	F	1,5,11	1,2,3	3	TRUE	D1
14	T	1,5,11	1,2,3	3	MATCH	LEMONS
15	T	1,5,11	1,2,3	4	RETURN	
11	T	1,5	1,2	4	RETURN	
5	T	1	1	4	RETURN	
1	T	-	-	4	STOP	

FIGURE 3.

The program will also return a true value of the flag if the input had been "LEMONS LIKES JACK" because although nonsensical it is syntactically correct. However if the ill-formed sentence "JACK LEMONS LIKES" were input, then the trace of the execution of the program in figure 2 might appear as shown in figure 4.

PAR	Flag	Return Stack	Input Stack	ISP	Symbolic instruction	
0	T	-	-	1	CALL	SENTENCE
2	T	1	1	1	CALL	SUBJECT
6	T	1,3	1,1	1	CALL	NOUN
12	T	1,3,7	1,1,1	1	MATCH	JACK
13	T	1,3,7	1,1,1	2	TRUE	D1
15	T	1,3,7	1,1,1	2	RETURN	
7	T	1,3	1,1	2	RETURN	
3	T	1	1	2	FALSE	A1
4	T	1	1	2	CALL	PREDICATE
8	T	1,5	1,2	2	CALL	VERB
16	T	1,5,9	1,2,2	2	MATCH	LIKES
17	F	1,5,9	1,2,2	2	RETURN	
9	F	1,5	1,2	2	FALSE	C1
11	F	1,5	1,2	2	RETURN	
5	F	1	1	2	RETURN	
1	F	-	-	1	STOP	

FIGURE 4.

3C. Difficulties With Definitions.

The grammar of the last two sample input sentences was very simple and it was easy to write a syntax program for it as was shown in figure 2. When using the top-down method of analysis with programming languages, which have a far more complex grammar, difficulties are experienced.

As an illustration of one such difficulty, consider the definition of an "unsigned integer" in Algol as shown below:

<unsigned integer> ::= <digit> | <unsigned integer> <digit>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

A syntax program for the above definitions is shown in figure 5.

Location	Instruction		
0		CALL	UI
1		STOP	
2	UI	CALL	DIGIT
3		TRUE	L1
4		CALL	UI
5		FALSE	L1
6		CALL	DIGIT
7	L1	RETURN	
8	DIGIT	MATCH	0
9		TRUE	M1
10		MATCH	1
11		TRUE	M1
.		.	.
.		.	.
.		.	.
26		MATCH	9
27	M1	RETURN	

FIGURE 5.

Suppose it was required to test whether "12$" is acceptable as an integer (the "$" symbol is used as a delimiter of the input stream). The trace of the execution of the syntax program in figure 5, with "12$" as input is shown below in figure 6.

PAR	Flag	Return Stack	Input Stack	ISP	Symbolic instruction	
0	T	-	-	1	CALL	UI
2	T	1	1	1	CALL	DIGIT
8	T	1,3	1,1	1	MATCH	0
9	F	1,3	1,1	1	TRUE	M1
10	T	1,3	1,1	1	MATCH	1
11	T	1,3	1,1	2	TRUE	M1
27	T	1,3	1,1	2	RETURN	
3	T	1	1	2	TRUE	L1
7	T	1	1	2	RETURN	
1	T	-	-	2	STOP	

FIGURE 6.

The program has accepted the leading digit on its own as as an integer. This is not what was intended. It should have accepted the whole stream of digits up to the first non-digit. This is an example of what is termed by Metcalfe as "mis-ordered alternatives".

The definition of "unsigned integer" (abbreviated "ui") can be re-ordered as follows:

$$\langle \text{ui} \rangle ::= \langle \text{ui} \rangle \langle \text{digit} \rangle \mid \langle \text{digit} \rangle$$

Re-writing the syntax program (the digit routine remaining unaltered) we get what is shown in figure 7.

Location	Instruction		
0		CALL	UI
1		STOP	
2	UI	CALL	UI
3		FALSE	L3
4	L3	CALL	DIGIT
5		RETURN	

FIGURE 7.

It is obvious from the instruction at location 2 that the recursive call causes the program execution to be stuck in a loop. This is what is termed by Metcalfe as "circular definition", and it has to be avoided in top-down analysers. The definition can be once again re-ordered as follows:

<ui> ::= <digit> <ui> | <digit>

and the syntax program for the above definition is shown in figure 8.

Location	Instruction		
0		CALL	UI
1		STOP	
2	UI	CALL	DIGIT
3		FALSE	L3
4		CALL	UI
5		TRUE	L3
6		CALL	DIGIT
7	L3	RETURN	

FIGURE 8.

In figure 9 below (continued on the next page) is shown a trace of the execution of the syntax program in figure 8 with the input "12$". The "--" in the figure points to the current symbol under scan.

NO.	PAR	Flag	PAR in stack	ISP in stack	ISP	Symbolic instruction	INPUT
1	0	T	-	-	1	CALL UI	-- 12$
2	2	T	1	1	1	CALL DIGIT	-- 12$
3	8	T	1,3	1,1	1	MATCH 0	-- 12$
4	9	F	1,3	1,1	1	TRUE M1	-- 12$
5	10	T	1,3	1,1	1	MATCH 1	-- 12$
6	11	T	1,3	1,1	2	TRUE M1	1-- 2$
7	27	T	1,3	1,1	2	RETURN	1-- 2$
8	3	T	1	1	2	FALSE L3	1-- 2$
9	4	T	1	1	2	CALL UI	1-- 2$
10	2	T	1,5	1,2	2	CALL DIGIT	1-- 2$
11	8	T	1,5,3	1,2,2	2	MATCH 0	1-- 2$
12	9	F	1,5,3	1,2,2	2	TRUE M1	1-- 2$
13	10	T	1,5,3	1,2,2	2	MATCH 1	1-- 2$
14	11	F	1,5,3	1,2,2	2	TRUE M1	1-- 2$
15	12	T	1,5,3	1,2,2	2	MATCH 2	1-- 2$
16	13	T	1,5,3	1,2,2	3	TRUE M1	12-- $
17	27	T	1,5,3	1,2,2	3	RETURN	12-- $
18	3	T	1,5	1,2	3	FALSE L3	12-- $
19	4	T	1,5	1,2	3	CALL UI	12-- $
20	2	T	1,5,5	1,2,3	3	CALL DIGIT	12-- $
21	8	T	1,5,5,3	1,2,3,3	3	MATCH 0	12-- $
22 - 39	each of ten digits will be rejected						
40	27	F	1,5,5,3	1,2,3,3	3	RETURN	12-- $
41	3	F	1,5,5	1,2,3	3	FALSE L3	12-- $

No.	PAR	Flag	PAR in stack	ISP in stack	ISP	Symbolic instruction	INPUT
42	7	F	1,5,5	1,2,3	3	RETURN	12-- ∅
43	5	F	1,5	1,2	3	TRUE L3	12-- ∅
44	6	T	1,5	1,2	2	CALL DIGIT	1-- 2∅
45	8	T	1,5,7	1,2,2	2	MATCH 0	1-- 2∅
46 - 51 the same as step nos. (12) - (17).							
52	7	T	1,5	1,2	3	RETURN	12-- ∅
53	5	T	1	1	3	TRUE L3	12-- ∅
54	7	T	1	1	3	RETURN	12-- ∅
55	1	T	-	-	3	STOP	12-- ∅

FIGURE 9.(continued from page 42).

This method works but the execution is painfully slow. In step number 42, the first alternative of the test that had begun in step number 9 has failed. In step number 44, the second alternative is tried (and will succeed) by re-setting the point of scan as at step number 9. For an n-digit integer there will be n recursive calls and the second alternative will have to be tried for the nth digit by re-setting the point of scan as at the (n - 1)th recursive call.

This method takes fewer steps than for the same definition of "unsigned integer" as outlined by Reeves [A17]. Reeves takes about 72 steps. It would, however, take about the same number as Reeves if the instruction at location 5 in figure 8 was replaced by the instruction "FLAG L3".

An analysis of the trace in figure 9 shows that the "CALL" operation was executed seven times. This requires that the pointers be stacked and unstacked seven times thus making execution

very slow. It will be noticed from step number 21 that for an n-digit integer, n + 2 sets of pointers have to be stacked at the nth recursive call, occupying considerable storage. Also, as noticed from steps number 10 and 44, the nth digit has to be parsed twice before the definition of "unsigned integer " can be satisfied.

In order to avoid all or some of the above mentioned time and space consuming steps, "unsigned integer" is re-defined in words as follows:

an unsigned integer is a digit followed by none or more digits.

To state the above definition in terms of the Backus normal form, an extension is made to it. A new syntactic operator is introduced into the meta-language:

$\int$ ... an integral sign.

which indicates that the following constituent may occur zero or more times [Al]. Now stated in the meta-language the latter definition of an unsigned integer becomes

$$\langle \text{unsigned integer} \rangle \ ::= \ \langle \text{digit} \rangle \int \langle \text{digit} \rangle \ .$$

It will be immediately noticed that this is a non-recursive definition. A syntax program for this definition is shown in figure 10.

Location	Symbolic instruction		
0		CALL	UI
1		STOP	
2	UI	CALL	DIGIT
3		FALSE	K1
4	K2	CALL	DIGIT
5		FLAG	K2
6	K1	RETURN	
7	DIGIT	MATCH	0
8		TRUE	M1
9		MATCH	1
10		TRUE	M1
11		MATCH	2
.		.	.
.		.	.
.		.	.
25		MATCH	9
26	M1	RETURN	

FIGURE 10.

The sample input "12$" can now be tried using the syntax program in figure 10. The trace of the execution is shown in figure 11.

No.	PAR	Flag	PAR in stack	ISP in stack	ISP	Symbolic instruction	INPUT
1	0	T	-	-	1	CALL UI	-- 12$
2	2	T	1	1	1	CALL DIGIT	-- 12$
3	7	T	1,3	1,1	1	MATCH 0	-- 12$
4	8	F	1,3	1,1	1	TRUE M1	-- 12$
5	9	T	1,3	1,1	1	MATCH 1	-- 12$
6	10	T	1,3	1,1	2	TRUE M1	1-- 2$
7	26	T	1,3	1,1	2	RETURN	1-- 2$
8	3	T	1	1	2	FALSE K1	1-- 2$
9	4	T	1	1	2	CALL DIGIT	1-- 2$
10	7	T	1,5	1,2	2	MATCH 0	1-- 2$
11	8	F	1,5	1,2	2	TRUE M1	1-- 2$
12	9	T	1,5	1,2	2	MATCH 1	1-- 2$
13	10	F	1,5	1,2	2	TRUE M1	1-- 2$
14	11	T	1,5	1,2	2	MATCH 2	1-- 2$
15	12	T	1,5	1,2	3	TRUE M1	12-- $
16	26	T	1,5	1,2	3	RETURN	12-- $
17	5	T	1	1	3	FLAG K2	12-- $
18	4	T	1	1	3	CALL DIGIT	12-- $
19	7	T	1,5	1,3	3	MATCH 0	12-- $
Steps 20 - 38, each of ten digits will be rejected							
39	5	F	1	1	3	FLAG K2	12-- $
40	6	T	1	1	3	RETURN	12-- $
41	1	T	-	-	3	STOP	12-- $

FIGURE 11

The last method is much better. It works faster not only because it takes fewer steps to recognise "12" as an unsigned integer, but also as it reduces the number of calls to the various routines from 3 + 2n (figure 9) to 2 + n for a n-digit integer. It will also be noticed from steps number 3, 10 and

19 that the nesting level never exceeds 2 and compared to the previous method only two sets of pointers have to be stacked instead of n + 2 for any n-digit integer.

This method is very useful, particularly so, since it not only solves the commom constituent, mis-ordered alternative, and circular definition problems, but also is faster in execution and simplifies the formulation of syntax definitions. Using these advantages the definition of "identifier" in Algol 60 can be re-stated from

<identifier> ::= <letter> | <identifier> <letter> | <identifier><digit>

to

<identifier> ::= <letter> $\int \left\{ \text{<letter>} \mid \text{<digit>} \right\}$.

The pair of braces "$\{\ \}$" is used to enclose a group. The last definition of "identifier" states that an identifier could be just a single letter, a letter followed by a mixture of letters and digits, a letter followed by other letters, or a letter followed by digits.

3D. Output.

Suppose it was required to translate the English sentence "CATS EAT MICE" into its German equivalent. The corresponding German basic symbols for each English basic symbol is outlined below:

English	German
CATS	KATZEN
EAT	FRESSEN
MICE	MAUSE

For convenience, a simple sentence with a one to one English-German correspondence is chosen for the example. Generally there is unlikely to be a one to one correspondence between various spoken and written languages and their respective rules of grammar are not the same. The aim of the above sample sentence is not to demonstrate spoken or written language translation, but to show how the syntax machine can be used to check the grammatical construction of a source language (English in the example) and, if correct, to output the required equivalent (German in our case) through the use of the output operations of the syntax machine.

During translation the German equivalents have to be output as the elements of the English sentence are parsed. A push-down stack, called output stack, is used for this purpose. In a similar fashion to the input stream, the symbols in the output stream are stacked in a sequential order, discarded from the stack, or left, depending entirely on the failure or success of the parsing mechanism as it searches through its heirarchy of goals and sub-goals. Hence the various positions in the output stack have to be recorded in the program control stack, along with the corresponding positions of the input symbols, when each sub-goal is entered.

A register called the "output symbol (stack) pointer" (OSP) is used to indicate the leading position in the output stack (or output symbol stream).

To accomodate the output, the existing syntax machine operations are modified as indicated below:

CALL — record the value of the OSP in the stack.

RETURN — if the flag is false then restore to the OSP the corresponding value from the top of the stack and delete it from the stack.

NOT — restore to the OSP the value from the top of the stack and delete it from the stack.

TRUE — if the flag is false then restore to the OSP the value from the top of the stack, leaving the stack unaltered.

The new operation to produce output is :-

PRINT — output the symbol in the data part of the instruction into the output stack and advance the OSP by one symbol position.

To specify, in the meta-language, symbols to be output by the "PRINT" operation the meta-brackets "[ ]" are used. They are used such that the enclosed set may be considered as a special kind of constituent.

In the example "CATS EAT MICE", KATZEN will be output for CATS, FRESSEN for EAT and MAUSE for MICE. Only part of the meta-grammar of sentence is now re-written as follows:

< verb> ::= EAT [FRESSEN]

< noun> ::= CATS [KATZEN] | MICE [MAUSE] .

The grammar indicates that as the basic English symbols are recognised, the German equivalents are output. The syntax program for the English-German translation is shown in figure 12.

Location	Instruction			Location	Instruction		
0		CALL	SEN	12		MATCH	CATS
1		STOP		13		FALSE	L3
2	SEN	CALL	SUBJ	14		PRINT	KATZEN
3		FALSE	L1	15		RETURN	
4		CALL	PRED	16	L3	MATCH	MICE
5	L1	RETURN		17		FALSE	L4
6	SUBJ	CALL	NOUN	18		PRINT	MAUSE
7		RETURN		19	L4	RETURN	
8	PRED	CALL	VERB	20	VERB	MATCH	EATS
9		FALSE	L2	21		FALSE	L5
10		CALL	NOUN	22		PRINT	FRESSEN
11	L2	RETURN		23	L5	RETURN	

FIGURE 12.

For the input sentence "CATS EAT MICE" it will produce the output sentence "KATZEN FRESSEN MAUSE".

It would be, however, possible to translate a single input language into several different output languages simultaeneously, that is, during a single execution of the syntax program. Suppose the input language is Ga and the output languages required are Gb, Gc, and Gd. If Na is a noun in Ga and Nb, Nc and Nd are the corresponding nouns in Gb, Gc and Gd respectively, then definitions in the met-grammar could be written as shown in the example below:

<noun> ::= Na [Nb, Nc, Nd] .

Several output stacks would be required, one for each language, or one stack with several pointers.

However, when translating arithmetic expressions, and most programming languages, it will be required to output whatever variables and arithmetic operators that may come in through the input stream and not pre-determined symbols embeded in the grammar as indicated in the last example. A new output operation for the syntax machine is defined:

COPY — output the symbol just read in, that is previous to the one specified by the ISP, and advance the OSP by one symbol position.

The "COPY" operation is for unparsing parts of the source language which do not need to be translated. The "COPY" operation in the meta-language is represented by the meta-symbol "K".

As an illustration, consider the translation of a simple arithmetic expression with only three variables P,Q,and R; the meta-language for the illustration is :-

<program> ::= <arithmetic expression> $

<arithmetic expression> ::= <term> { {<adding operator> K <term>} }

<adding operator> ::= + | -

<term> ::= <factor> { {<multiplying operator> K <factor>} }

<multiplying operator> ::= * | /

<factor> ::= <variable> K | (<arithmetic expression>)

<variable> ::= P | Q | R

The corresponding syntax program is shown in figure 13.

```
PROG    CALL    AE
        MATCH   ø
        STOP
AE      CALL    TERM
        FALSE   B1
B3      CALL
        CALL    AO
        FALSE   B2
        COPY
        CALL    TERM
 B2     RETURN
        FLAG    B3
B1      RETURN
TERM    CALL    FAC
        FALSE   D1
D3      CALL
        CALL    MO
        FALSE   D2
        COPY
        CALL    FAC
D2      RETURN
        FLAG    D3
D1      RETURN

 AO     MATCH   +
        TRUE    C1
        MATCH   -
 C1     RETURN
FAC     CALL    VAR
        FALSE   F2
        COPY
        RETURN
 F2     MATCH   (
        FALSE   F1
        CALL    AE
        FALSE   F1
        MATCH   )
 F1     RETURN
 MO     MATCH   *
        TRUE    E1
        MATCH   /
 E1     RETURN
VAR     MATCH   P
        TRUE    G1
        MATCH   Q
        TRUE    G1
        MATCH   R
 G1     RETURN
```

FIGURE 13.

The syntax program in figure 13 will check the syntax of an arithmetic expression, for example P+Q/(Q+R*P)-Q ⊄, and output it if the structure of the expression satisfies the definition.

If it were required to convert the translated expression into say, Reverse Polish notation, then the translated expression, if found to be correct, has to be manipulated in a way such that the required form is output. Automatic manipulation (editing) is achieved by the use of another abstract machine called the "Editor Machine" (or "Editor") which like the syntax machine is also a stored program special-purpose computer.

The "Editor Machine" uses as input a stream of symbols, interspersed with special edit codes, output by the syntax machine. It edits the stream of symbols in any desired fashion as requested by the edit codes. The editor machine is brought into operation after the syntax machine has completed its translation and produced the interspersed symbol stream.

The Editor Machine could be combined with the syntax machine thus interspersing edition with translation. This would result in a smaller volume of symbols in the output stack as edit codes will no longer have to be output but acted upon. There appears to be some apparent disadvantages with interspersed translation and edition as opposed to sequential translation and edition. The translation process may not be wholly successful and edition time then would be uselessly spent. Also output from one of the many alternative sub-goals which might later fail to satisfy a hierarchy of super-

goals, could result in a waste of unnecessary edition time. Of-course the interspersed method could be used to advantage if edition is performed after a major syntactic block in a source program has been parsed and found to be correct. However, there may be other incorrect major syntactic blocks which might make the entire translation unsuccessful. It appears that the choice of the method for edition of translated code may be problem dependant. Both the methods can be used easily and requires hardly any programming effort to combine or seperate the machines, once programmed.

The sequential method is demonstrated in this chapter.

The syntax machine operation which produces edit codes (or which causes and edition to be performed in the interspersed mode) is introduced:

EDIT — output the symbol specified in the data part of the instruction and advance the OSP by one symbol position.

The "EDIT" operation appears to be similar to the "PRINT" operation, but the symbol contained in the data part of the "EDIT" instruction is an edit code and the negative equivalent of it is output so that the editor will be able to differentiate it from other symbols in its input stream. The various edit codes are:-

C .....concatenate

X .....exchange

W .....write

The editor posses a stack memory operated in a push-down fashion. It executes a set of operations whose result corresponds to the function of the edit code. The process consists of reading in a stream of symbols and placing all non-edit symbols sequentially in its memory until it reads an edit code. It performs the operation requested by the edit code and continues with the stacking again until it senses the next edit code. The editor completes its cycle of operations and comes to a halt after servicing a "W" edit code.

The edit code "C" causes the editor to concatenate the two top-most symbols in its stack memory into one larger symbol; "X" causes the two top symbols to interchange their respective positions in the stack;"W" causes the editor to empty the edited contents of its memory and come to a halt.

For example if it was required to convert "PART" into "TRAP", then the following combination of symbols should be input into the editor:-

P A $\overline{\text{X}}$ $\overline{\text{C}}$ R T $\overline{\text{X}}$ $\overline{\text{C}}$ $\overline{\text{X}}$ $\overline{\text{C}}$ $\overline{\text{W}}$

the flagged letters denote edit codes.

The state of the editor stack memory after each input symbol is shown below (the comma is used to indicate seperation of symbols):

Input	Stack	Output
P	P	
A	P,A	
$\overline{X}$	A,P	
$\overline{C}$	AP	
R	AP,R	
T	AP,R,T	
$\overline{X}$	AP,T,R	
$\overline{C}$	AP,TR	
$\overline{X}$	TR,AP	
$\overline{C}$	TRAP	
$\overline{W}$		TRAP

These edit codes in the meta-language are represented by C, X and W, and the corresponding symbolic instructions are:- EDIT C , EDIT X , and EDIT W .

Returning to the original problem of converting an arithmetic expression into Reverse Polish, the meta-language defined earlier is slightly altered to :-

<program> ::= <ae> $ W

<ae> ::= <term> $\int$ {<ao> K <term> X C C}

<ao> ::= + | -

<term> ::= <factor> $\int$ {<mo> K <factor> X C C}

<mo> ::= * | /

<factor> ::= <var> K | (<ae>)

<var> ::= P | Q | R

The syntax program corresponding to the above definitions is shown in figure 14. Routines for <ao>, <mo> and <var> remain unchanged from those in figure 13 and therefore are not shown.

```
        CALL    PROG
        STOP
PROG    CALL    AE
        FALSE   A1
        MATCH   ¢
        FALSE   A1
        EDIT    W
A1      RETURN
AE      CALL    TERM
        FALSE   B1
B3      CALL
        CALL    AO
        FALSE   B2
        COPY
        CALL    TERM
        FALSE   B2
        EDIT    X
        EDIT    C
        EDIT    C
B2      RETURN
        FLAG    B3
B1      RETURN
```

```
TERM    CALL    FAC
        FALSE   D1
D3      CALL
        CALL    MO
        FALSE   D2
        COPY
        CALL    FAC
        FALSE   D2
        EDIT    X
        EDIT    C
        EDIT    C
D2      RETURN
        FLAG    D3
D1      RETURN
FAC     CALL    VAR
        FALSE   F2
        COPY
        RETURN
F2      MATCH   (
        FALSE   F1
        CALL    AE
        FALSE   F1
        MATCH   )
F1      RETURN
```

FIGURE 14.

The input "P+P/(Q+R*P)-Q ¢" when fed to the syntax program in figure 14 produced the output "PPQRP*+/+Q-" from the editor. A trace

of the execution of the program in figure 14 will occupy many pages. From a trace of the execution it was determined that the following stream of symbols was input to the editor (flagged symbols are edit codes and commas are used to show seperation of symbols) :

P + P / Q + R * P $\overline{X}$ $\overline{C}$ $\overline{C}$ $\overline{X}$ $\overline{C}$ $\overline{C}$ $\overline{X}$ $\overline{C}$ $\overline{C}$ $\overline{X}$ $\overline{C}$ $\overline{C}$ - Q $\overline{X}$ $\overline{C}$ $\overline{C}$ $\overline{W}$

and the process within the editor is shown below.

Input	Editor stack	Output
P	P	
+	P,+	
P	P,+,P	
/	P,+,P,/	
Q	P,+,P,/,Q	
+	P,+,P,/,Q,+	
R	P,+,P,/,Q,+,R	
*	P,+,P,/,Q,+,R,*	
P	P,+,P,/,Q,+,R,*,P	
$\overline{X}$	P,+,P,/,Q,+,R,P,*	
$\overline{C}$	P,+,P,/,Q,+,R,P*	
$\overline{C}$	P,+,P,/,Q,+,RP*	
$\overline{X}$	P,+,P,/,Q,RP*,+	
$\overline{C}$	P,+,P,/,Q,RP*+	
$\overline{C}$	P,+,P,/,QRP*+	
$\overline{X}$	P,+,P,QRP*+,/	
$\overline{C}$	P,+,P,QRP*+/	
$\overline{C}$	P,+,PQRP*+/	

Input	Editor stack	Output
$\overline{X}$	P,PQRP*+/,+	
$\overline{C}$	P,PQRP*+/+	
$\overline{C}$	PPQRP*+/+	
-	PPQRP*+/+,-	
Q	PPQRP*+/+,-,Q	
$\overline{X}$	PPQRP*+/+,Q,-	
$\overline{C}$	PPQRP*+/+,Q-	
$\overline{C}$	PPQRP*+/+Q-	
$\overline{W}$		PPQRP*+/+Q-

3E. Null Symbols.

Consider the problem of affixing"labels" to a statement in the following definitions:

<statement> ::= <label list> <basic statement> C

<label list> ::= <empty> $\int$ { <label> C }

If the label list consists only of "empty" then "empty" has to be concatenated with "basic statement". Even if there are several labels, "empty" has to be concatenated with the first label in order that the definition of "statement" be satisfied. Some sort of a symbol has to be output when the source text is parsed with "label list" as a sub-goal. A syntax machine operation which does this is introduced;

NULL — output a special symbol and advance the OSP by one symbol position; this special symbol when output later by the editor machine is ignored.

Using "0" to represent the null operation in the meta-language, the definition for < label list > can now be re-stated as

$$< \text{label list} > ::= 0 \int \{ < \text{label} > C \} \quad .$$

"NULL" or "0" can be said to be equivalent to a phrase < empty > in the Backus normal form. The syntax program routine corresponding to the last definition of < label list > is shown in figure 15. It will be noticed that the routine will always return a true value of the flag.

LABEL LIST	NULL	
L2	CALL	LABEL
	FALSE	L3
	EDIT	C
L3	FLAG	L2
L1	RETURN	

FIGURE 15.

3F. Markers.

Suppose s syntax program is required to translate any one of two sentences, shown below, from English to its Latin equivalent:

"LIONS EAT LAMBS" and "LAMBS EAT GRASS".

The Latin equivalents of the above two sentences are "LEONES AGNAS EDUNT" and "AGNAE HERBAS EDUNT", respectively.

The grammar for the above translation can be defined as shown

below:

$\langle$sentence$\rangle$ ::= $\langle$subject$\rangle$ $\langle$predicate$\rangle$ W

$\langle$subject$\rangle$ ::= $\langle$noun$_1\rangle$ | $\langle$noun$_2\rangle$

$\langle$predicate$\rangle$ ::= $\langle$verb$\rangle$ $\{\langle$noun$_1\rangle$ | $\langle$noun$_3\rangle\}$ X

$\langle$verb$\rangle$ ::= EAT [EDUNT]

$\langle$noun$_1\rangle$ ::= LIONS [LEONES] | GRASS [HERBAS]

$\langle$noun$_2\rangle$ ::= LAMBS [AGNAE]

$\langle$noun$_3\rangle$ ::= LAMBS [AGNAS]

The set of definitions are cumbersome and implementation would be time consuming since three noun routines will have to be included in the syntax program. To make translation faster only a single noun routine should be included in the syntax program. This does not appear practical at first glance as Latin words, unlike most English words, have a root and its ending which changes according to which syntactic part of a sentence it forms. If a single noun routine is employed, it must be capable of producing the correct translation for the word "LAMBS", wheher it forms a part of the subject or object. Hence it follows that it has to remember whether it has been called upon by the subject or predicate routine to perform the translation. This is made possible by what Metcalfe terms as "markers". A special-purpose register called "mark register" (MR), which holds a sequence of digits(markers) which are used as indicators, is used in the implementation of this facility. The parser stack is augmented at each level so that the setting of the "mark register" can be recorded together with

the values of the ISP, OSP and PAR. Existing functions are modified as follows:

CALL — copy the MR into the top of the stack and set zeros in the MR.

TRUE — if the flag is false, re-set zeros in the MR.

RETURN
NOT — remove the copy of the MR from the head of the stack and copy it into the MR.

and three new parser operations are defined below:

MARK — insert a "1" digit, in the digit position specified by the data part of the instruction, in the copy of the MR at the top of the stack.

SELECT — copy the data part of the instruction into a temporary location called "hold".

TEST — copy the data part of the instruction into the "PAR" if the digit position of the MR specified by "hold" is a zero.

In the meta-language the MARK and TEST functions are represented by the meta-symbols Mn and Tn where n is an integer specifying the particular mark of the set.

It will be noticed from the modifications of the parser instructions and from the three new operations that each activation of a routine has a set of marker digits associated with it in the "MR". This set is clear on entry to each routine. It can be varied by "MARK" instructions within routines which it calls but is re-set to zeros whenever an alternative is rejected. The current values

of the "MR" are tested by the "SELECT" and "TEST" instructions.

The problem of the noun in the English-Latin translation for the selected sample sentences can be solved by:-

(i) the noun routine sets a marker if it recognises the word "LAMB"; it cannot output the Latin equivalent as it has no knowledge of its super-goal.

(ii) the super-goal (subject or predicate) upon testing the marker, will output the Latin equivalent for "LAMB". If the marker is not set, that is, clear, the noun routine did not detect "LAMB" in the input stream.

The meta-grammar can now be re-written to incorporate the above solution as shown below:

```
<sentence>  ::= <subject> <predicate> W
<subject>   ::= <noun> T1 [AGNAE]
<predicate> ::= <verb> <noun> T1 [AGNAS] X
<verb> ::= EAT [EDUNT]
<noun> ::= LAMB M1 | LIONS [LEONES] | GRASS [HERBAS]
```

The grammar defined above is much better - <noun> is defined only once. The corresponding syntax program is shown in figure 16.

If any of the two sample sentences are input, the correct translation will be output by the editor. The sentence, "LIONS EAT GRASS", if input, will also produce the correct grammatical translation. Incorrect endings would be output if the nouns, "LIONS" and "GRASS", became the object and subject respectively.

```
        CALL    SEN
        STOP
SEN     CALL    SUBJ
        FALSE   A1
        CALL    PRED
        FALSE   A1
        EDIT    W
A1      RETURN
SUBJ    CALL    NOUN
        FALSE   B1
        SELECT  1
        TEST    B1
        PRINT   AGNAE
B1      RETURN
VERB    MATCH   EAT
        FALSE   C1
        PRINT   EDUNT
C1      RETURN
PRED    CALL    VERB
        FALSE   D1
        CALL    NOUN
        FALSE   D1
        SELECT  1
        TEST    D2
        PRINT   AGNAS
D2      EDIT    X
D1      RETURN
NOUN    MATCH   LAMB
        FALSE   E2
        MARK    1
        RETURN
E2      MATCH   LIONS
        FALSE   E3
        PRINT   LEONES
        RETURN
E3      MATCH   GRASS
        FALSE   E1
        PRINT   HERBAS
E1      RETURN
```

FIGURE 16.

However if "LIONS" was intended to be used either as subject or object in addition to "LAMBS", then this could be easily done by setting a second marker after the symbol "LIONS" has been recognised by the noun routine as shown in figure 17. The corresponding change in the subject routine (predicate routine not shown, but change is very similar) would be as shown in figure 18.

```
NOUN    MATCH    LAMBS
        FALSE    E2
        MARK     1
        RETURN
E2      MATCH    LIONS
        FALSE    E3
        MARK     2
        RETURN
          .
          .
```

FIGURE 17.

```
SUBJ    CALL     NOUN
        FALSE    B1
        SELECT   1
        TEST     B2
        PRINT    AGNAE
        RETURN
B2      SELECT   2
        TEST     B1
        PRINT    LEONES
B1      RETURN
```

FIGURE 18.

Similarly a third marker could be used if "GRASS" was required to form part of either the subject or predicate.

It will be easily seen that if a language with a large vocabulary is considered, a very large number of markers might be required. The sample sentences in the examples were used to demonstrate the use of the facility of markers, not to formulate an algorithm for spoken or written language translation. If spoken languages were considered, then a computer with a very large store will not necessarily facilitate translation either by syntax-directed methods or table look up for vocabulary, as spoken languages do not usually have a one to one correspondence, and are not free of syntactic and semantic ambiguities. However, it seems that it is not impossible to use the syntax machine for such translations on a resticted vocabulary.

CHAPTER 4.

SPEEDING UP TRANSLATION.

The character set of almost all programming languages consist of letters, digits and special characters. It follows that in syntax-directed analysis that the final sub-goal of most alternatives will be the one that will have to recognise a basic symbol as a letter or a digit. As an illustration consider the definition of "identifier" in Algol, shown below:

<identifier> ::= <letter> $\int$ { <letter> | <digit> }

<letter> ::= A | B | C | ..... | Z

<digit> ::= 0 | 1 | 2 | ..... | 9

The syntax program for the above definitions is shown in figure 19.

```
        CALL    IDENT
        STOP
IDENT   CALL    LETTER
        FALSE   A1
A2      CALL    LETTER
        FLAG    A2
        CALL    DIGIT
        FLAG    A2
A1      RETURN
DIGIT   MATCH   0
        TRUE    B1
```

```
                MATCH   1
                  .
                  .
                  .
                MATCH   9
B1      RETURN
LETTER  MATCH   A
        TRUE    C1
        MATCH   B
                  .
                  .
                  .
        MATCH   Z
C1      RETURN
```

FIGURE 19.

The digit routine consists of 20 syntax machine instructions and the letter routine consists of 52 syntax machine instructions. To recognise the symbol "A" as a letter it would take 3 operations the symbol "Z" would require 52 operations to be recognised as a letter. If "B30$" ($ being treated as string terminator) were input to the program in figure 19, the number of operations required to establish "B30$" as an identifier is summarized below:

B as a letter	9
3 as a letter	54
3 as a digit	9
0 as a letter	54
0 as a digit	5
$ as a letter	54
$ as a digit	22
control	2
Total	209

The number of operations is startling considering that it would require about 84 operations to establish "A" as an identifier. The entire routines for the digit and letter recognisers can be replaced by single hardware functions, similar to the "MATCH" operation, by treating < letter > and <digit > as terminal types instead of defined types.

Two new syntax machine functions are defined below:

DIGIT if the symbol specified by the ISP is a digit then

set the flag to a true value and advance the ISP by one symbol position otherwise set the flag to a false value.

ALPHA — if the symbol specified by the ISP is a letter, that is one of A to Z or any other symbol considered to be a letter, then set the flag to a true value and advance the ISP by one symbol position otherwise set the flag to a false value.

It will be noticed that the two functions defined above are similar to the "MATCH" operation, only taking a fractionally longer time to execute.

The syntax program in figure 19 can now be re-written as shown in figure 20 below.

```
        CALL    IDENT
        STOP
IDENT   ALPHA
        FALSE   A1
A2      ALPHA
        FLAG    A2
        DIGIT
        FLAG    A2
A1      RETURN
```

FIGURE 20.

The syntax routines for digit and letter are no longer required. The number of syntax machine operations now needed to establ-

ish "B30$" as an identifier is summarised below:

B as a letter	3
3 as a letter	2
3 as a digit	2
0 as a letter	2
0 as a digit	2
$ as a letter	2
$ as a digit	2
control	2
Total	17

This certainly is a much faster method. The identifier routine in figure 20 did not make use of any "CALL" operations thus saving more execution time as well as avoiding stacking and unstacking of pointers. There is no fixed ratio for the execution time saved but the table below shows that the time saved is definitely considerable and that it varies according to the length and composition of the identifier:

Identifier	Figure 19	Figure 20
A	84	9
Z99999	501	29
ZZZZZZ	401	19

Another technique to speed up translation is by the avoidance of unnecessary stacking and unstacking of pointers through the use

"CALL" and "RETURN" operations. Due to the mechanism of top-down syntax-directed translation unnecessary stacking of pointers seems unavoidable. This may be seen from the definition of "variable" in Algol:

<variable> ::= <simple variable> | <subscripted variable>

<simple variable> ::= <variable identifier>

<variable identifier> ::= <identifier> .

The complete set of definitions is not shown and is not necessary to illustrate the point. The syntax program for part of the above definitions is shown in figure 21.

1	VAR	CALL	SIMVAR
2		TRUE	A1
3		CALL	SUBVAR
4	A1	RETURN	
5	SIMVAR	CALL	VARID
6		RETURN	
7	VARID	CALL	IDENT
8		RETURN	

FIGURE 21.

Before the sub-goal identifier is established, and, control returned to statement number 2, 3 calls and 3 returns have been made. If the flag is returned with a false value from the identifier routine, the OSP and the ISP will have to restored to their value at statement number 1. Thus stacking of pointers at statement numbers 5 and 7 will have served no useful role since the value

of the pointers at statement number 1 is only useful and important. Execution will be speeded if the pointers at statements number 5 and 7 are not stacked.

A new syntax machine operation is defined:

BRANCH — copy the data part of the instruction into the PAR.

The operation defined above is equivalent to an unconditional branch operation. The syntax program in figure 21 can now be re-written using the newly defined "BRANCH" operation as shown in figure 22.

1	VAR	CALL	SIMVAR
2		TRUE	A1
3		CALL	SUBVAR
4	A1	RETURN	
5	SIMVAR	BRANCH	VARID
6	VARID	BRANCH	IDENT

FIGURE 22.

Since the last set of pointers are stacked at statement number 1, a "RETURN" operation executed by the identifier routine will return control to statement number 2 which is what is required of it.

Whenever a "CALL" operation is immediately followed by a "RETURN" operation, a "BRANCH" operation can be usually used to replace both of them. A "BRANCH" operation is by definition very much faster in execution than either a "CALL" or a "RETURN" operation; hence considerable execution time is saved when it replaces

both of them.

The syntax program in figure 22 can be improved upon and may be re-written as shown in figure 23.

1	VAR	CALL	SIMVAR
2		TRUE	A1
3		BRANCH	SUBVAR
4	A1	RETURN	
5	SIMVAR	BRANCH	IDENT

FIGURE 23.

If statement number 3, in figure 23, is executed then control from the subscripted variable routine will be returned to the super-routine which called on the variable routine (possibly by use of a "BRANCH" operation), and not to statement number 4 which has to be retained because of statement number 2.

CHAPTER 5.

ERRORS AND RECOVERY.

Neither syntax-directed nor syntax-controlled analysers are capable, by themselves, of dealing with non sentences. Syntax-directed analysers are usually incapacitated by syntactic errors in their input sentences [A6]. At the present time there is no completely satisfactory scheme for dealing with syntactic errors discovered in the course of predictive analysis [A5].

Two operations defined below are used to assist error analysis and recovery in the syntax machine. An indicator "errind", denotes whether a source program contained any syntactic errors. At the beginning of the parsing process it is set to a true value. The two new operations are:-

ERROR — list the error code contained in the data part of of the instruction and set both the flag and errind to a false value.

WARN — list the warning code contained in the data part of the instruction and set the flag to a true value.

Control is passed to the next sequential instruction after execution of any one of the above operations. The error and warning messages are listed on to a printer, interspersed with the source deck listing.

These operations by themselves do not carry out any error

checking of the input sentence; they merely list errors and warnings which have to be detected by the previously defined operations.

The use of the last two operations, defined above, is illustrated below by writing syntax programs for a few definitions from the sample programming language defined by Metcalfe [A1]. The definitions are listed in figure 24 below.

<program> ::= <declaration list> ¢ <statement list> ¢.

<declaration list> ::= <declaration> ∫ { ¢ <declaration> }

<statement list> ::= <statement> ∫ { ¢ <statement> }

<declaration> ::= (<variable list>)

FIGURE 24.

The syntax program for the set of definitions in figure 24 is shown below in figure 25 which is continued on to the next page .

```
            CALL    PROGR
            STOP
PROGR       CALL    DECLS
            FALSE   A1
            MATCH   ¢
            FALSE   A1
            CALL    STLST
            FALSE   A1
            MATCH   ¢
A1          RETURN
STLST       CALL    STMNT
            FALSE   B1
B2          CALL
            MATCH   ¢
            FALSE   B3
```

```
                CALL        STMNT
                RETURN
B3              FLAG        B2
B1              RETURN
DECLS           CALL        DECLN
                FALSE       C1
C2              CALL
                MATCH       ⊄
                FALSE       C3
                CALL        DECLN
                RETURN
C3              FLAG        C2
C1              RETURN
DECLN           MATCH       (
                FALSE       D1
                CALL        VARLS
                FALSE       D1
                MATCH       )
D1              RETURN
```

FIGURE 25.

If it was required to give an error indication if either the "declaration list" or the "statement list" is parsed to be incorrect, then the routine for "PROGR" in figure 25 will have to be changed to that as shown in figure 26.

The following declaration statement will be parsed to be incorrect since the right hand parentheses ")", is missing:

(A, B, T ⊄

It may be feasible to assume the ")" to be present and accept the above declaration as a valid declaration and warn the programmer of his carelessness. The "DECLN" routine in figure 25, altered to that as shown in figure 26 will achieve this.

```
PROGR   CALL    DECLS       DECLN   MATCH   (
        FLAG    A2                  FALSE   D1
        ERROR   01                  CALL    VARLS
        RETURN                      FALSE   D1
A2      MATCH   $                   MATCH   )
        FALSE   A1                  FLAG    D1
        CALL    STLST               WARN    51
        FLAG    A3          D1      RETURN
        ERROR   02
        RETURN
A3      MATCH   $
A1      RETURN
```

FIGURE 26.

If the following source statements are input to the syntax program in figures 25 and 26 :-

```
(A,B ,T $
 C,D,E) $
A = 2*B + C $
C = A**2 $.
```

then the resulting output listing on the line printer would be:

```
INPUT
     (A,B ,T $
     WARNING   51
      C,D,E) $
     ERROR   01
     SYNTAX INCORRECT
```

The second declaration statement is erraneous since the "(" is missing. The parsing process comes to a halt when the first syntactic error is detected. In order to continue the parsing process the syntax program has to be designed such that the incorrect statement is ignored and scan continued from the beginning of the next source statement. Source text has to be skipped until the statement delimiter "$" is recognised. The following set of syntax machine operations will achieve this :-

```
          .
          .
          .
K1    MATCH    $
      FLAG     K2
      NEXT
      BRANCH   K1
K2      .
        .
        .
```

In Algol the following set of operations will advance the ISP just pass the delimiter "END":-

```
L1    MATCH    E
      FLAG     L2
      NEXT
      BRANCH   L1
L2    MATCH    N
      FALSE    L1
      MATCH    D
      FALSE    L1
L3      .
        .
        .
```

The syntax program for the set of definitions in figure 24 can now be re-written using the following arbitrary rules for errors and recovery:

i) a declaration without the ")" will be accepted but a warning indication (code 51) given.

ii) an error code of "01" will be given for an invalid declaration list and the list will be skipped.

iii) any statement that is unrecognisable will be skipped and an error indication (code 02) given.

Only a few possible errors have been considered in the above assumptions. Since all statements have to be parsed the "statement list" (STLST) routine will return a true value of the flag at the end of the parsing process; hence all error indication is given at statement level and the syntax machine function "STOP" is modified as follows:

STOP — if the value of errind is false then display a false flag otherwise display the value of the flag and stop the machine.

The syntax program follows:

```
        CALL    PROG
        STOP
PROG    CALL    DECLS
        FLAG    A3
        ERROR   01
```

```
A2      MATCH   ;
        FLAG    A4
        NEXT                    skip to end of
        BRANCH  A2              declaration list
A3      MATCH   ;
A4      CALL    STLST
        FALSE   A1
        MATCH   ;
A1      RETURN
DECLS   CALL    DECLN
        FALSE   C1
        CALL
        MATCH   ;
        FALSE   C3
        CALL    DECLN
        RETURN
C3      FLAG    C2
C1      RETURN
STLST   CALL    STMNT
        FLAG    B2
B6      ERROR   02              ---- statement unrecognisable
        MATCH   ;               skip to end of unrecogn-
        FLAG    B4              isable statement.
        NEXT
        BRANCH  B5
```

```
B4      CALL
        BRANCH  B7
B2      CALL
        MATCH   ⊄
        FALSE   B6
B7      CALL    STMNT
        FALSE   B8
        RETURN
        FLAG    B2
B8      MATCH   .
        FALSE   B9
        NOT
B9      RETURN
        NEXT
        BRANCH  B6
DECLN   MATCH   (
        FALSE   D1
        CALL    VARLS
        FALSE   D1
        MATCH   )
        FLAG    D1
        WARN    51
D1      RETURN
```

this ensures that the delimiter of the program "." is not interpreted to be a statement and that every statement of the program is parsed.

The STLST (statement list) routine checks every statement and

will fail to recognise ".", the program delimiter, as a statement. The "NOT" operation followed immediately by the return operation will transfer control to the "PROG" routine with a true flag setting and with the ISP pointing to the "$" of the last statement of the program. The setting of the error indicator, errind, then gives an indication as to whether the parsing was error free.

CHAPTER 6.

AN IMPLEMENTATION ON THE IBM 1620.

The implementation on the IBM 1620 consists of two programs, a program to simulate the syntax machine and a program to simulate the editor machine. The two programs are easily combined into a single program to form what is called a "translator machine". In addition, a "grammar assembly program" (a two pass assembler) is used to assemble syntax programs from their symbolic formats into absolute coded formats.

6A. The Syntax Machine.

The program to simulate the syntax machine is written in the IBM 1620 symbolic programming language. It is essentially an interpreter. It first reads in and stores the absolute coded syntax program for any particular source-target language pair, in its internal memory, then reads in the source statements and executes the stored syntax program. A general flow-diagram of the mechanism is shown in figure 27 and the internal organisation (core lay-out) is shown in figure 28.

6A.1. Storing The Syntax Program.

The absolute coded syntax program which is a series of digits is read in from punched cards. The digits flagged to mark the beginning of each operation code and the data part. The digits have to be flagged since the instructions are of variable length. All

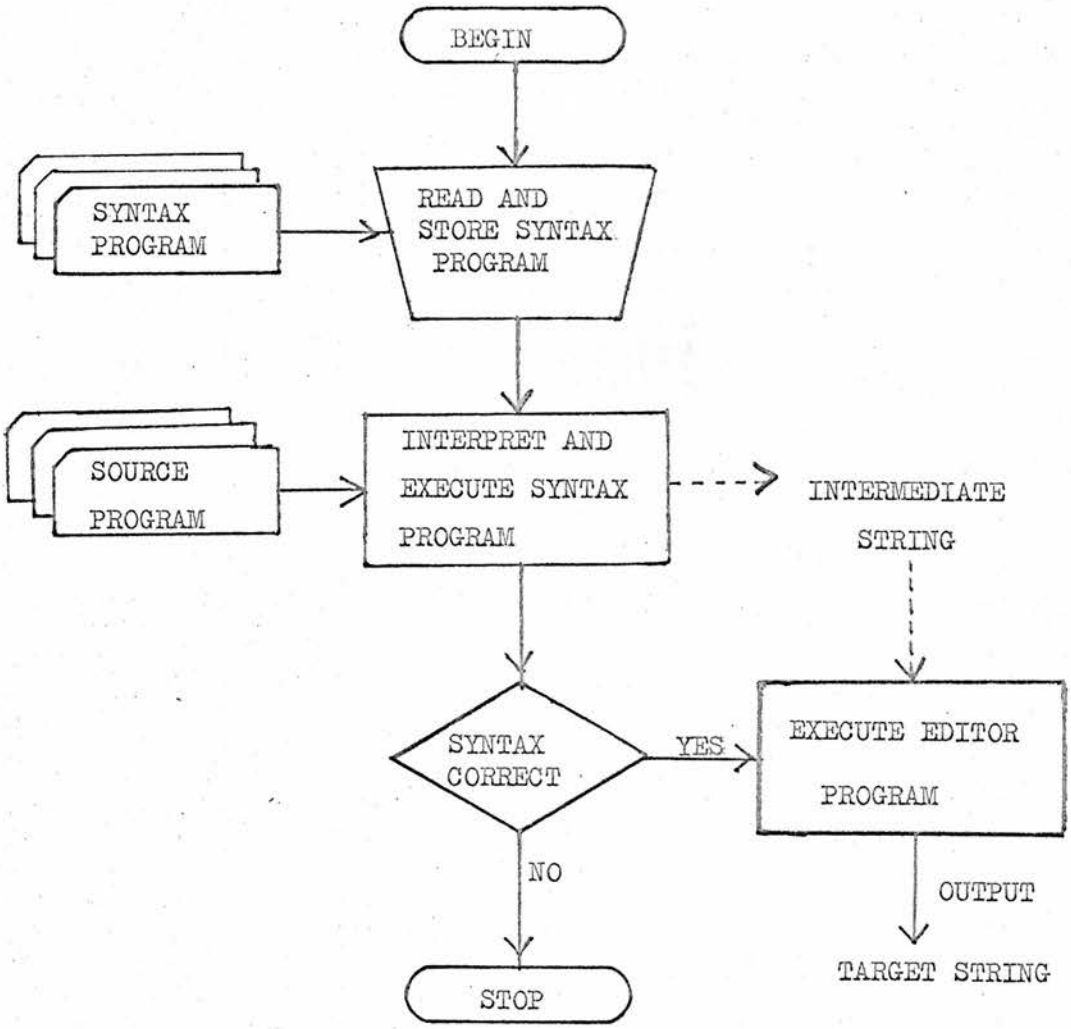


FIGURE 27 BASIC FLOW DIAGRAM OF THE SYNTAX MACHINE

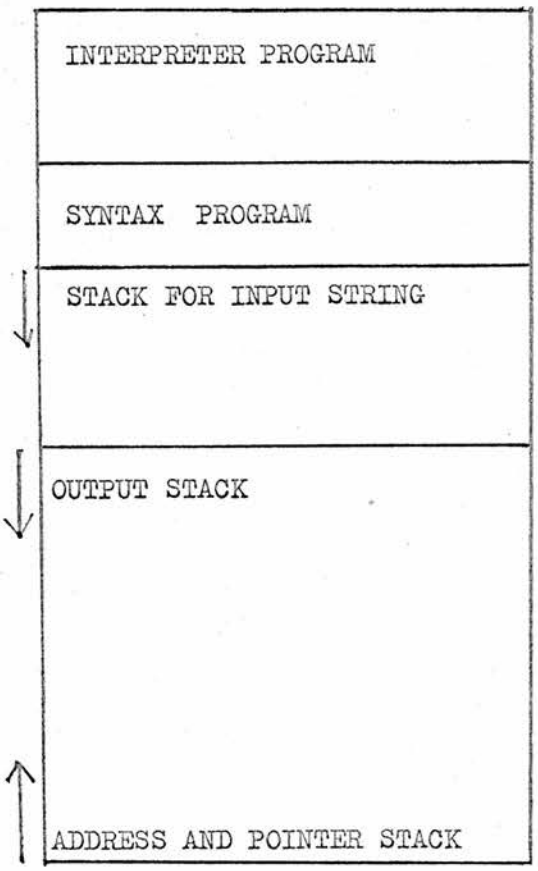


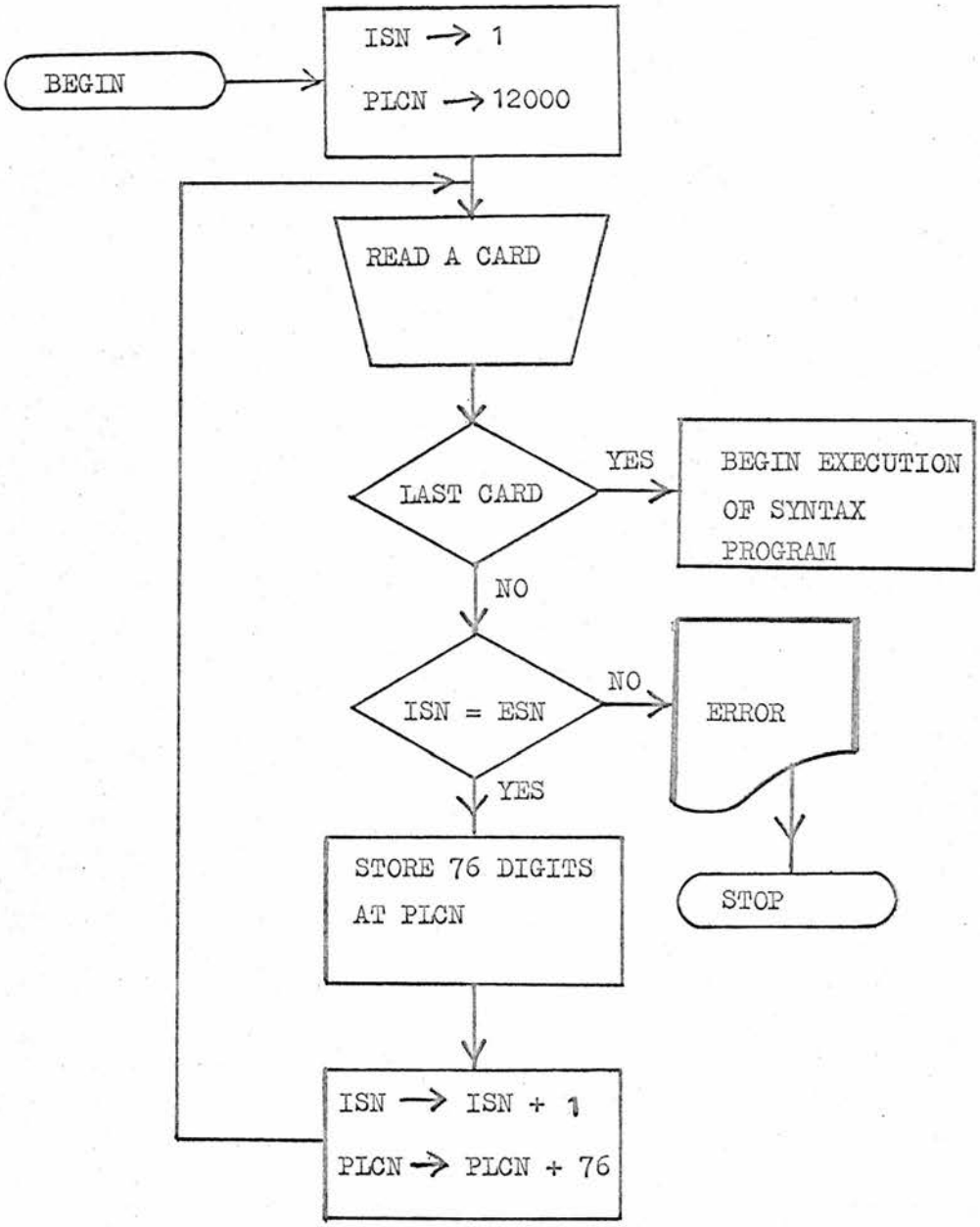
FIGURE 28 INTERNAL (CORE STORE) LAYOUT OF THE SYNTAX MACHINE

edit codes are flagged in their units position as well in order to differentiate them from other basic symbols.

The entire program is held on cards in a fixed format. The first four columns of each card consists of a sequence number and the remaining 76 columns consist of program. The sequence numbering begins at "1" on the first card and increases in steps of one for every card. The program is stored contiguously, internally in the syntax machine, that is, the program from the 1st 2 cards will occupy 152 contiguous locations starting at location 12000. An end of file card, with the first four columns punched "R99R", causes the syntax machine to read in the source statements. All syntax programs have to have an end of file card in order to seperate the program from the source text. If the syntax program cards are not in sequential order, an error message is output on the printer and the syntax machine terminates execution. The cards have to be re-set and execution begun from the initial starting point. The loading of the syntax program is shown diagramatically in figure 29.

6A.2. Execution of the Syntax Program.

The simulation of the Syntax Machine is performed by a cycle of a small number of operations. The two digit code representing a syntax machine operation is interpreted and an appropriate routine which performs the required operation is entered. Upon completion of the operation control is returned to the main cycle. This is shown digramatically in figure 30.



ISN - INTERNAL SEQUENCE NUMBER

ESN - EXTERNAL SEQUENCE NUMBER

PLCN - PROGRAM LOCATION

FIGURE 29

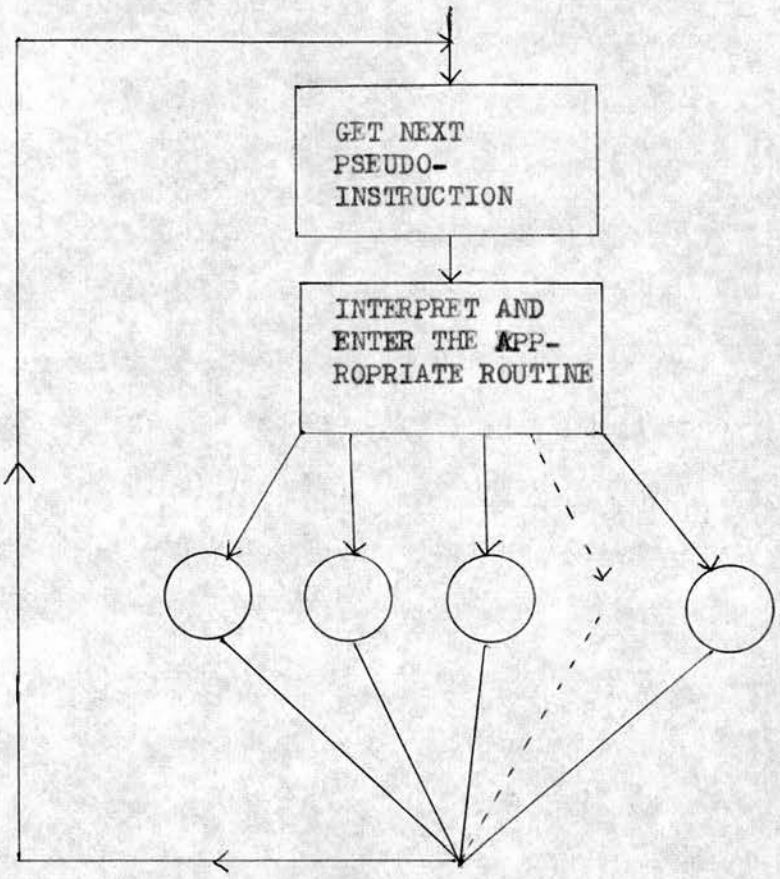


FIGURE 30.

The first pseudo-instruction is stored at location 12000. The addresses of the next pseudo-instructions to be obeyed are determined by the routines which perform the required syntax machine operations. Since the individual routines determine the value of the PAR, the facility of interpreting variable length instructions is easily implemented. The pseudo-instructions have the following formats :-

2 digits, for example "RETURN", "NOT", "NEXT"; these consist of the two digit operation code only.

4 digits, for example "MATCH", "EDIT", "PRINT"; these consist of a two digit operation code and a two digit data part.

6 digits, for example "ERROR", "WARN"; a two digit operation code and a four digit data part.

7 digits, for example "CALL", "FLAG"; a two digit operation code and a five digit address.

A 5-digit address part is used because of the addressing system of the IBM 1620; two digits are used to represent a basic symbol since a character is represented internally in the IBM 1620 by two digits. All symbolic syntax machine instructions are limited in scope, that is they cannot parse more than one basic symbol with a single operation. An operation such as "PRINT *VAR" has to coded as a series of symbolic operations as shown below:

```
PRINT    *
PRINT    V
PRINT    A
PRINT    R
EDIT     C
EDIT     C
EDIT     C
```

The "EDIT" operations have to be used so as to ensure that "*VAR" is output as a single symbol by the editor instead of four one character symbols.

The internal codes used to represent the various syntax machine operations are listed below:

COPY	00
NULL	01
RETURN	02
NEXT	03
NOT	04
MATCH	05
EDIT	06
PRINT	07
MARK	08
SELECT	09
TEST	10
CALL	11
STOP	12
TRUE	13
FALSE	14
FLAG	15
BRANCH	16
ERROR	17
WARN	18
DIGIT	19
ALPHA	20

By making use of the codes just listed, an absolute coded program for the set of definitions below can be written as shown in figures 31 and 32:

<group> ::= <item> T1 [D] <item>

<item> ::= A | B M1 | C

```
START   CALL    GROUP
        STOP
ITEM    MATCH   A
        TRUE    A1
        MATCH   B
        FALSE   A2
        MARK    1
A2      TRUE    A1
        MATCH   C
A1      RETURN
GROUP   CALL    ITEM
        FALSE   B1
        SELECT  1
        TEST    B2
        PRINT   D
B2      CALL    ITEM
B1      RETURN
```

FIGURE 31.

(a)	(b)	(c)
12000	$\overline{11}$	$\overline{12048}$
12007	$\overline{12}$	
12009	$\overline{05}$	$\overline{41}$
12013	$\overline{13}$	$\overline{12046}$
12020	$\overline{05}$	$\overline{42}$
12024	$\overline{14}$	$\overline{12035}$
12031	$\overline{08}$	$\overline{71}$
12035	$\overline{13}$	$\overline{12046}$
12042	$\overline{05}$	$\overline{43}$
12046	$\overline{02}$	
12048	$\overline{11}$	$\overline{12009}$
12055	$\overline{14}$	$\overline{12084}$
12062	$\overline{09}$	$\overline{71}$
12066	$\overline{10}$	$\overline{12077}$
12073	$\overline{07}$	$\overline{44}$
12077	$\overline{11}$	$\overline{12009}$
12084	$\overline{02}$	

FIGURE 32.

The absolute coded program is obtained by following the procedure below:

i) Fill in the location of each symbolic instruction in column (a) of figure 32, the location of the first instruction being the location 12000. The location counter is advanced

depending on the length of each instruction in digits.

ii) For each instruction, fill in the operation code, the IBM 1620 internal representation of the basic symbols in the data part of the instruction or the absolute address. The operation code is filled-in in column (b) and the data or address in column (c) of figure 32. Once the absolute addresses have been filled-in, it is then very easy to read of label addresses from column (a).

iii) Flag the leading digits on each line of columns (b) and (c). All the digits in columns (b) and (c) concatenated and punched serially onto cards in card columns 5 to 80 with sequence numbers in columns 1 - 4 is the absolute coded program for the set of definitions defined above.

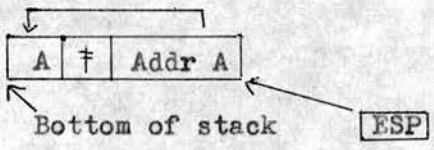
It is fairly easy to hand code a syntax machine program but very tedious especially if alterations have to be made. The grammar assembly program which is described later on in this chapter automatically calculates absolute addresses and produces an absolute coded syntax program from a symbolic syntax program.

The program to simulate the syntax machine required 4178 core positions and 247 executable instructions. The space occupied by stacks varies at execution time and is not included in the above figures.

6B. The Editor Machine.

The input to the editor machine is a stream of basic symbols and edit codes which is left in core by the syntax machine. Negative digital representation of X, C, and W cause the editor to manipulate basic symbols; all other symbols are regarded as basic symbols and are stacked sequentially as they come in. The basic flow diagram of the editor is shown in figure 33.

A pseudo-list processing technique is used for stacking symbols in the editor memory. For example if the symbol A followed by the symbol B is input to the editor, the symbol A is first stacked as shown in the diagram below:



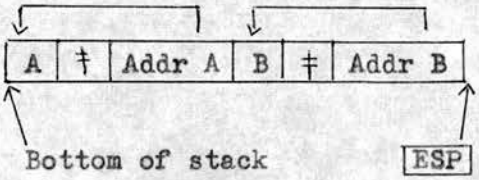
Addr A --- Address of A

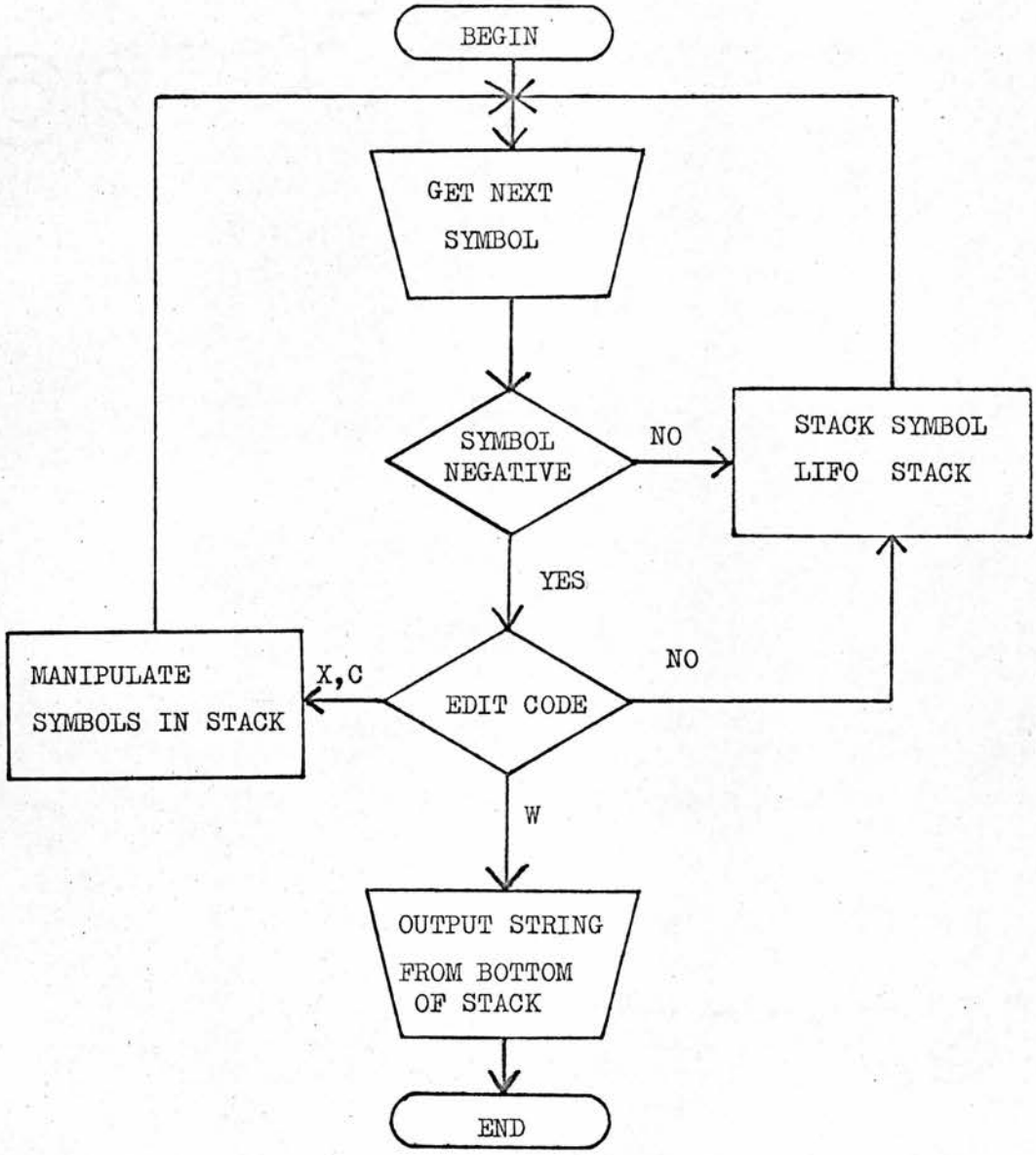
‡ --- Record mark

ESP --- Editor stack pointer

An index register "Editor stack pointer" (ESP) points to the next vacant position in the stack.

When the symbol B is input, it is stacked thus:-

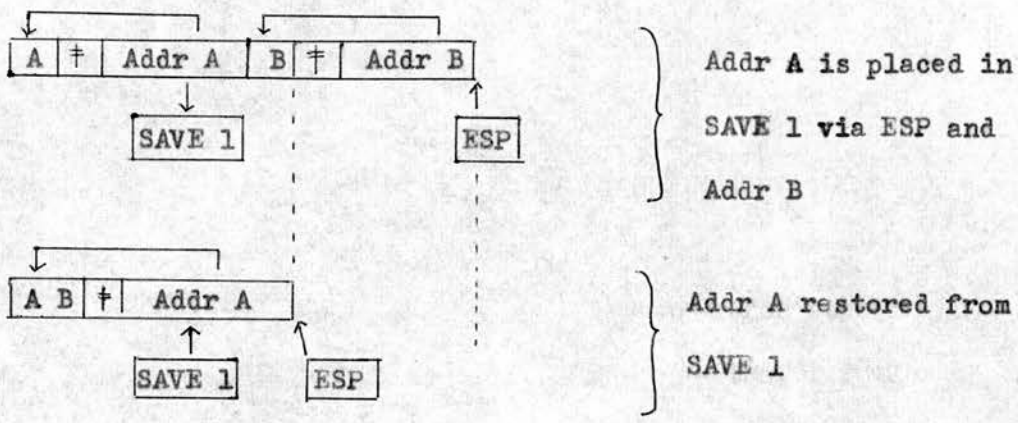




LIFO - LAST IN FIRST OUT

FIGURE 33 BASIC FLOW DIAGRAM OF THE EDITOR MACHINE

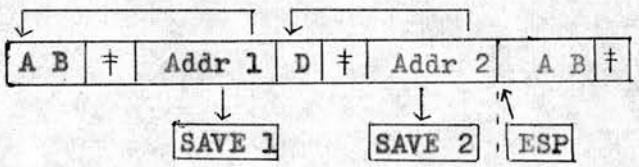
If the edit code C is next input, the following results:-



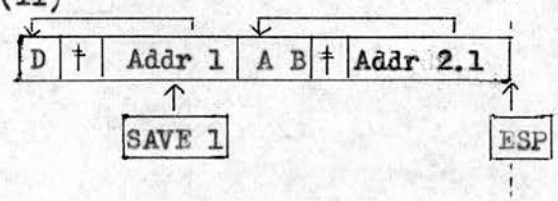
The ESP is decremented by 6 digit positions, 5 for Addr B and 1 digit for the record mark.

Suppose a symbol D is stacked in the memory following the symbol AB and it is followed by an edit code X, then the following results:-

(i) Addr 2 and then Addr 1 are saved and the record at Addr 1 is moved to the top of the stack:



(ii)



The record at Addr 2 is moved to Addr 1 and the record at the top of the stack, that is, the one indicated by the ESP is moved to a new address, Addr 2.1, The value of the ESP remains

unchanged as only the positions of the two records are interchanged. The algorithm for computing Addr 2.1 is given below:

Addr 2.1 = Addr 1 + Addr 2 + ESP .

The program to simulate the editor machine requires 49 executable instructions on the IBM 1620. It is a relocatable program occupying 870 core locations not including the editor stack.

6C. Execution Trace.

A trace routine which lists a trace of every syntax machine operation is included in the program that simulates the syntax machine. The trace gives an indication of the state of the parsing process prior to the execution of the instruction listed. Each line of the trace gives an indication of the flag setting, the value of the ISP, the address of the instruction to be executed currently, the instruction in symbolic form, the value of the PAR at the top of the stack, the level of nesting, and the remaining part of the input string still to be parsed from the current point of scan. All addresses are given in their absolute form. The trace with absolute addresses in not difficult to interpret if the listing of the symbolic syntax machine program that is output by the grammar assembly program is at hand. It would be possible to output the addresses in their symbolic form, but this would require a copy of the symbol table to be input with the parsing program. Since 15 digits will be required for every label in the symbol table, the symbol table might occupy more space than the syntax program itself.

The trace routine becomes operational when switch number 1 on the console panel is set. The trace routine occupies 1034 core positions and has 39 executable instructions. A trace of the execution of the syntax program in figures 31 and 32 is shown in figure 34 below. The input string consists of "A B".

INSTRUCTION Label	Function	Data	Flag	PAR at top of stack	ISP	RCN	INPUT STRING
12000	CALL	12048	T	00000	1	1	AB
12048	CALL	12009	T	12007	1	2	AB
12009	MATCH	A	T	12055	1	3	AB
12013	TRUE	12046	T	12055	2	3	B
12046	RETURN		T	12055	2	3	B
12055	FALSE	12084	T	12007	2	2	B
12062	SELECT	1	T	12007	2	2	B
12066	TEST	12077	T	12007	2	2	B
12077	CALL	12009	T	12007	2	2	B
12009	MATCH	A	T	12084	2	3	B
12013	TRUE	12046	F	12084	2	3	B
12020	MATCH	B	T	12084	2	3	B
12024	FALSE	12035	T	12084	3	3	
12031	MARK	1	T	12084	3	3	
12035	TRUE	12046	T	12084	3	3	
12046	RETURN		T	12084	3	3	
12084	RETURN		T	12007	3	2	
12007	STOP		T	00000	3	1	

RCN = Level of nesting.

FIGURE 34.

6D. The Grammar Assembly Program

A syntax program for the syntax machine can be hand coded as shown in figures 31 and 32, punched onto cards and then loaded into the syntax machine. The method is relatively very simple but tedious. Arithmetical mistakes are likely when incrementing the location counter and the entire coding has to be thoroughly checked to see that correct operation codes are inserted for the symbolic operations. Punching series of digits on cards and then verifying them is painstaking. If an alteration is made to the syntax program all the addresses may have to be computed again and probably the program has to be repunched. The use of the "BRANCH" instruction can be very useful to patch up alterations but all the above is not worth the effort is an assembly program is available. A "grammar assembly program" is used to produce a punched syntax program for the syntax machine from the corresponding symbolic syntax program.

The grammar assembly program (GAP) is a two pass assembler. The first pass checks the syntax of the symbolic instructions and builds up a symbol table. Error indications are given if the syntax of a statement is incorrect. The input to the GAP is in fixed format and on punched cards. All fields are left justified and the format is as shown below:

card columns	1 - 5	Label
card columns	6 - 11	Function
card columns	15 - 19	Data

A non-left justified function is treated as an error but assembled as a no-operation function during the second pass, that is "BRANCH" to the next sequential instruction. A non-left justified label or address is not an error but might not produce the result intended, for example LABX$ and $LABX ($ = blank) are treated as separate labels and are valid. The mistake might however be detected in the second pass if reference is made to both of them and only one appeared in the label part while the other appeared in the data part of the instruction. Duplicate labels are not stored in the symbol table; only its first occurrence is stored. Subsequent occurrences are treated as errors. Any information outside the three fields are treated as comments or rather ignored.

The last card or end of file card for the source deck has the characters "END$$" punched in the first five columns. During the first pass, all statements are stored on disk and statements with errors are listed on the printer together with an error code denoting the type of error. There are two possible errors which can be detected in thefirst pass and these are listed below together with their error codes.

A - Label defined more than once.

B - Invalid function field.

If the following two statements were in a symbolic program then they would be listed as errors as shown below:

```
L1      NATCH       A           ERROR - B
L1      CALL        ITEM        ERROR - A
```

Error - B is also listed during the second pass. At the end of the first pass a symbol table is listed on the printer. The symbol table for the symbolic syntax program in figure 31 would appear as shown below:

START	12000
ITEM	12009
A2	12035
A1	12046
GROUP	12048
B2	12077
B1	12084

If there are any errors during the first pass, an error indicator is set - the setting of which is passed on to the second pass.

The second phase is linked into core from disk at the end of the first phase. During the second phase syntax machine language code is generated for each symbolic statement. When 76 machine code digits have been generated, they are punched onto a card in the last 76 columns. The first four columns of the card contain a sequence number. The numbering begins with number 1 for the first card and is increased by 1 for every card punched. An end of file card with the sequence R99R in the first four columns is punched to denote the end of the machine language deck.

If the error indicator was set during the first pass then the punched output is suppressed. If there were no errors in the first pass, then punched output will be produced but suppressed as soon as an error is detected during the second pass.

An analysis, together with a message if there were any errors, is given at the end of the second pass as shown below:

```
ERROR/S DETECTED - PUNCHED OUTPUT SUPPRESSED OR HALTED -
IGNORE OUTPUT IF ANY

0018 CARDS PROCESSED, 0086 CORE POSITIONS REQUIRED.
```

All source statements are listed during the second pass together with the machine code each statement generates, and a one-character error code if the statement is erroneous. The format of the listing is shown below:

```
A2   TRUE   A1          12035  13  12046
     MATCH  C           12042  05
```

statement cols. 1 - 80. | location | code | address

error code (if any)

Two types of errors are checked for during the second pass, namely illegal syntax machine functions and addresses which have not been defined. It checks first for the illegal function, so that it will generate a null operation if the function is illegal. An error code "C" is used to denote an undefined label or address.

If error "B" is detected no attempt is made to check for error "C." Error "C" is detected by checking the label in the data part of the instruction against the symbol table built up during the first pass. If it does not exist in the symbol table, error code "C" is listed and an absolute address of zero is generated in the data part of the machine code instruction.

The first phase of the GAP occupies 3638 core positions and requires 84 executable statements. The second phase which is a relocatable program occupies 4516 core positions and requires 122 executable statements.

CHAPTER 7.

USING THE SYNTAX MACHINE.

An example of the use to which the syntax machine can be put is demonstrated below. A source language program is to be checked and, if the syntax is correct, is to be translated into the machine language of a particular computer. The example shown demonstrates how the syntax machine can be used to translate the same source language into the machine language of any desired computer by specifying the target language in the meta-grammar. For convenience, since an assembler to translate the intermediate language is not at hand, a hypothetical computer will be considered.

7A. The Assembly Language of a Hypothetical Computer.

On the assumption that the target computer is of the Polish notation class with a stack type accumulator, a symbolic assembly language is defined as follows [A1]:

i) A program is a string of operands and operators separated by commas, terminated by the *END operator.

ii) An operand is a variable (represented by an identifier) or a number.

iii) An operator is represented by an asterisk and three letters.

There are two types of operators, declarative and imperative. The declarative operators are:

*VAR defines the following operand as the location

specified by the assembly location counter, and increments that counter by one.

*LAB identical to *VAR but does not increment the assembly location counter.

*END terminates assembly.

Each imperative operator and operand is contained in separate words of the computer. The program is executed sequentially, jumping to a new operation when so directed by certain operators. The appearance of an operand causes that operand to be placed in the computer stack (a push-down operation). The appearance of an operator causes execution of the corresponding function. Execution is assumed to begin following the last declarative operator and its operand. The imperative operators and their functions are as follows:

*CLA replaces the address in the top entry of the stack with the value of that address.

*ADD replaces the top two entries of the stack with their sum.

*SUB replaces the top two entries of the stack with their difference.

*MUL replaces the top two entries of the stack with their product.

*DIV replaces the top two entries of the stack with their quotient.

*EXP replaces the top two entries of the stack with their

exponentiation.

*NEG replaces the top entry of the stack with its negative value.

*ABS replaces the top entry of the stack with its absolute value.

*STO stores the value in the top entry of the stack in the address specified by the next-to-top entry. Removes both entries from the stack.

*TRA transfer control to the address specified in the top entry of the stack. Removes that entry from the stack.

*TZE transfers control to the address specified in the top entry of the stack only if the next-to-top entry contains a zero value. Removes both entries from the stack.

*TPL functions as *TZE, but for a plus value.

*TMI functions as *TZE, but for a minus value.

*HLT stops execution

7B. The Grammar.

The meta-language formulation of the grammar for the source-target language translation is given below [A1] :

<letter> ::= A|B|C| ....|Z

<digit> ::= 0|1|2| ...|9

<identifier> ::= <letter> K ∫ {{<letter> | <digit>} KC} [,] C

<number> ::= { <integer> { <fraction> C | } | <fraction> } [,] C

<fraction> ::= .K <integer> C

<integer> ::= <digit> K ∫ { <digit> K C }

<variable> ::= <identifier>

<label> ::= <identifier>

<expression> ::= <term> ∫ { + <term> [*ADD,] C C | - <term>
[*SUB,] C C } | + <expression> | - <expression>
[*NEG,] C

<term> ::= <factor> ∫ { * <factor> [MUL,] C C | / <factor>
[*DIV,] C C }

<factor> ::= <primary> ∫ { ** <primary> [*EXP,] C C }

<primary> ::= <variable> [*CLA,] C | <number> | (<expression>)
| / <expression> / [*ABS,] C

<statement> ::= <label list> <basic statement> C

<label list> ::= 0 ∫ { <label> . [*LAB,] X C C }

<basic statement> ::= <set statement> | <go to statement>
| <if statement>

<set statement> ::= <variable> = <expression> [*STO,] C C

<go to statement> ::= GO TO <label> [*TRA,] C

<if statement> ::= IF <expression> = <condition> , GO TO <label> X C C

<program> ::= <declaration list> ∅ <statement list> ∅ C [*END.] C W

<declaration list> ::= <declaration> ∫ { ∅ <declaration> C }

<declaration> ::= (<variable list>)

<variable list> ::= <variable> [*VAR,] X C ∫ { , <variable>
[*VAR,] X C C }

<statement list> ::= <statement> $\int$ { ∅ <statement> C } [*HLT,] C

<condition> ::= 0 [*TZE,] | + [*TPL,] | - [*TMI,]

The syntax program for the grammar defined above is shown in Appendix I.

The following source language program which computes the square root (B) of a number of (A) to a precision of 0.0001 was input to the syntax machine :

```
(A,B,T) ∅
B = (A + 1) / 2 ∅
S1. T = B ∅
B = B + (A/B - B)/2 ∅
IF /B -T/ - 0.0001 = +, GO TO S1 ∅.
```

The output from the syntax machine is given below:

```
INPUT
      { Listing of source program
SYNTAX CORRECT
OUTPUT FROM EDITOR
*VAR,A,*VAR,B,*VAR,T,B,A,*CLA,1,*ADD,2,*DIV,*STO,*LAB,S1,T,B,*CLA,
*STO,B,B,*CLA,A,*CLA,B,*CLA,*DIV,B,*CLA,*SUB,2,*DIV,*ADD,*STO,B,*CLA,
T,*CLA,*SUB,*ABS,0.0001,*SUB, S1,*TPL,*HLT,*END.
```

The above resulting target language program now could be assembled by the target machine and executed.

The syntax program was then modified to list warnings and errors

and take recovery action and parse the entire source program. The following warning and error codes were arbitrarily decided upon and the syntax program changed accordingly.

WARNING CODES

51	No delimiter (⊄) after declaration list. Delimiter is assumed.
51	No ")" in declaration statement. ")" assumed.
53	No "," after "condition" in "If statement." "," is assumed.
54	No delimiter (⊄) after statement list. Delimiter is assumed.

ERROR CODES

01	Invalid declaration list.
04	Invalid declaration statement.
05	Statement unrecognisable.
06	No label after "GO TO" in "if statement."

Nonsensical programs were input and the results are shown below:

EXAMPLE 1.

INPUT

(A,G,GOTO,IF ⊄

WARNING 52

(L,M,N,O,P)

```
WARNING   51
   IF A + B = + GO TO S1 ¢
WARNING   53
   IF A - B = 0, GO TO ¢
ERROR     06
ERROR     05
   K. A = B + 20.25 * GO TO ¢
        G = A - B + GOTO ¢.
SYNTAX INCORRECT
```

The above example did not produce any target language output because of errors. GOTO and IF have been accepted as identifiers and no ambiguity is apparent between them and IF and GOTO in "if statements" and "go to statements."

EXAMPLE 2: The incorrect statement from Example 1 was removed.

```
INPUT
   (A,G,GOTO,IF ¢
WARNING   52
   (L,M,N,O,P)
        GOTO = IF ¢
WARNING   51
   IF A + B = + GO TO S1 ¢
WARNING   53
   K. A = B + 20.25 * GO TO ¢
        G = A - B + GOTO ¢
```

OUTPUT FROM EDITOR

*VAR,A,*VAR,G,*VAR,GOTO,*VAR,IF,*VAR,L,*VAR,M,*VAR,N,*VAR,O,
*VAR,P,GOTO,IF,*CLA,*STO,A,*CLA,B,*CLA,*ADD,S1,*TPL,*LAB,K,A,B,
*CLA,20.25,GOTO,*CLA,*MUL,*ADD,*STO,G,A,*CLA,B,*CLA,*SUB,GOTO,*CLA,
*ADD,*STO,*HLT,*END.

SYNTAX CORRECT

The resulting target language is correct.

EXAMPLE 3.

For this example the syntax machine was modified so that the editor would produce the target language program even if the source program contained errors. The source program in Example 1 was then input to the modified syntax machine.

INPUT

(A,G,GOTO,IF ∅

WARNING 52

(L,M,N,O,P)

GOTO = IF ∅

WARNING 51

IF A ≠ B = + GO TO S1 ∅

WARNING 53

IF A - B = 0, GO TO ∅

ERROR 06

ERROR 05

K. A = B + 2-.25 *GOTO ∅

G = A - B + GOTO ∅

OUTPUT FROM EDITOR

*VAR,A,*VAR,G,*VAR,GOTO,*VAR,IF,*VAR,L,*VAR,M,*VAR,N,*VAR,O,
*VAR,P,GOTO,IF,*CLA,*STO,A,*CLA,B,*CLA,*ADD,S1,*TPL,*LAB,K,A,
B,*CLA,20.25,GOTO,*CLA,*MUL,*ADD,*STO,G,A,*CLA,B,*CLA,*SUB,GOTO,
*CLA,*ADD,*STO,*HLT,*END.

SYNTAX INCORRECT.

No target language has been generated for the incorrect statement. The target language for all the other statements is correct and can be verified from Example 2. This example demonstrates that correct output can be generated even if there are errors in the source program. It also demonstrates correct stacking and unstacking mechanism of the syntax machine.

EXAMPLE 4.

INPUT

(A,B,T)$

B = (A + 1) / 2 $

S1. T = $

ERROR 05

B = B + (A/B - B) 12 $

IF /B - T/ - 0.0001 = +, GO TO S1 $.

SYNTAX INCORRECT

CHAPTER 8.

A SYNTAX CHECKER FOR BASIC.

A brief description of a syntax checker for source statements in the BASIC language [A18] is given in this chapter. The formal definition of the Darthmouth BASIC Language, on which this syntax checker is based, photocopied from "An Anatomy of a Compiler" [A19], is shown at the beginning of Appendix II. A few changes have been made to the formal definition given in Appendix II in order that the source statements can be written in the IBM 1620 character code, and also to avoid problems that have been mentioned in chapter 3 - left recursion, mis-ordered alternatives etc. The notation in which the formal definition of BASIC is written differs slightly from that used in this thesis. The differences are shown below with their equivalent in the Backus extended form next to them:

:= is the same as ::=

$\{<x>\}_0^\infty$ is the same as $\int <x>$

but $\{<\text{digit}>\}_1^9$ specifies the minimum and maximum contiguous occurrence of the group enclosed in the meta-braces; for example

$<\text{integer}> ::= \{<\text{digit}>\}_1^9$

specifies that an integer must at least have one digit but no more than nine digits.

It might appear that the following definitions,

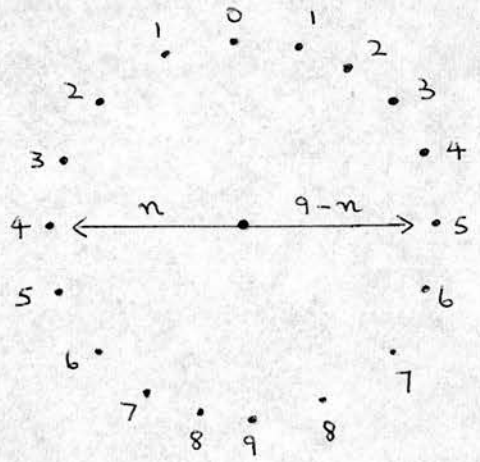
$<\text{integer}> ::= \{<\text{digit}>\}_1^9$

and

$$\langle \text{decimal number} \rangle ::= \{\langle \text{digit} \rangle\}_{1}^{n \leq 9} . \{\langle \text{digit} \rangle\}_{0}^{9-n} ,$$

would be difficult to implement on the syntax machine, since it is not equipped with any index registers or modifiers; this is not entirely so. It is easy but a little tedious as shown diagramatically in figures II.A and II.B. The syntax routine corresponding to the logic in figure II.B is not shown in this chapter. It is shown in Appendix II. In figures II.A and II.B, the ISP is advanced by one position when a digit is detected in the input stream. In figure II.A the flag is always set to true after the first digit has been detected. The scan is terminated as soon as the maximun number of nine digits have been found.

In figure II.B a branching technique, analagous to the dial shown in the figure below, is used.



If n digits are scanned before the decimal point then a branch is made to a position where it will be possible to scan a maximum of (9 - n) digits.

The syntax checker shown in Appendix II is designed to any unrecognisable statement and continue scan from the beginning of the next

statement. An error code is listed immediately following the unrecognisable statement. It merely indicates that the statement is erroneous and does not specify any particular syntactic error.

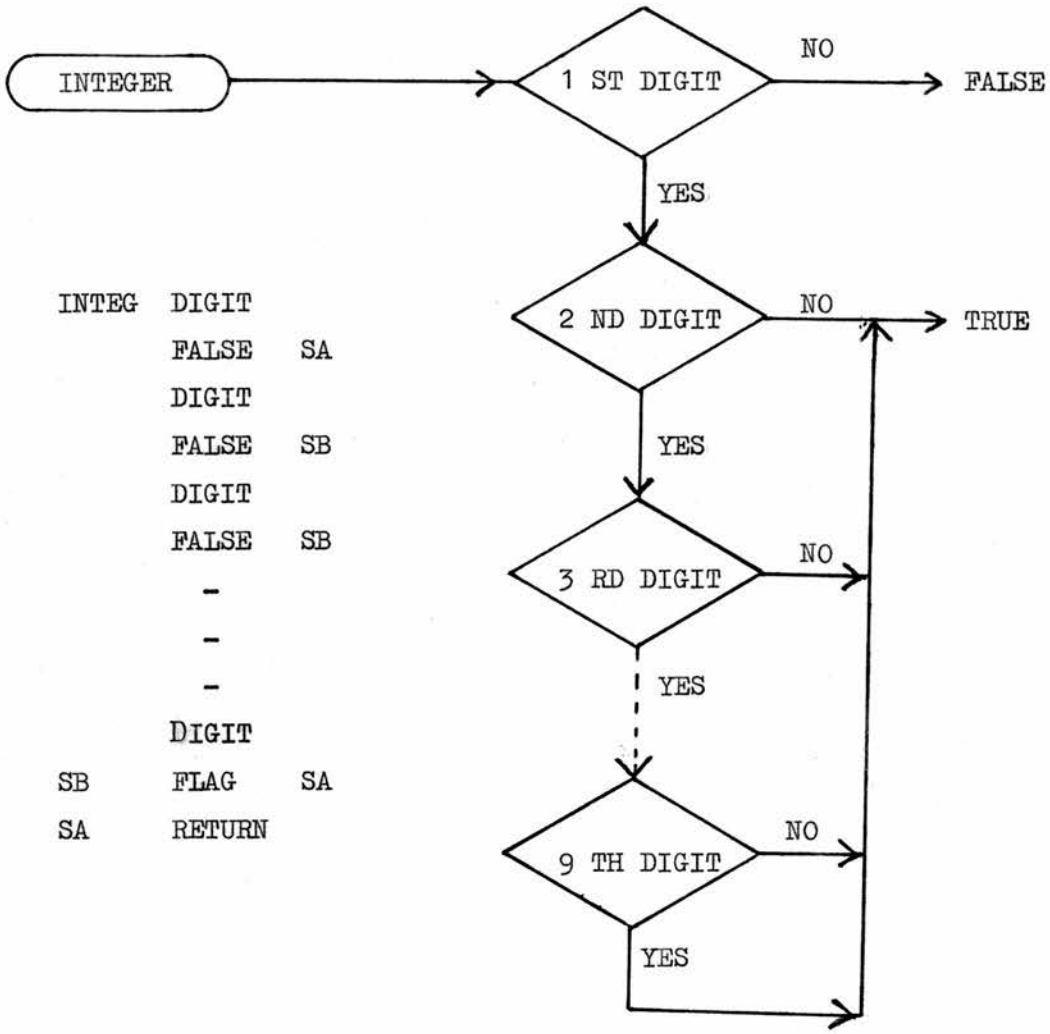


FIGURE II.A RECOGNIZER WITH SYNTAX PROGRAM FOR "<INTEGER>"

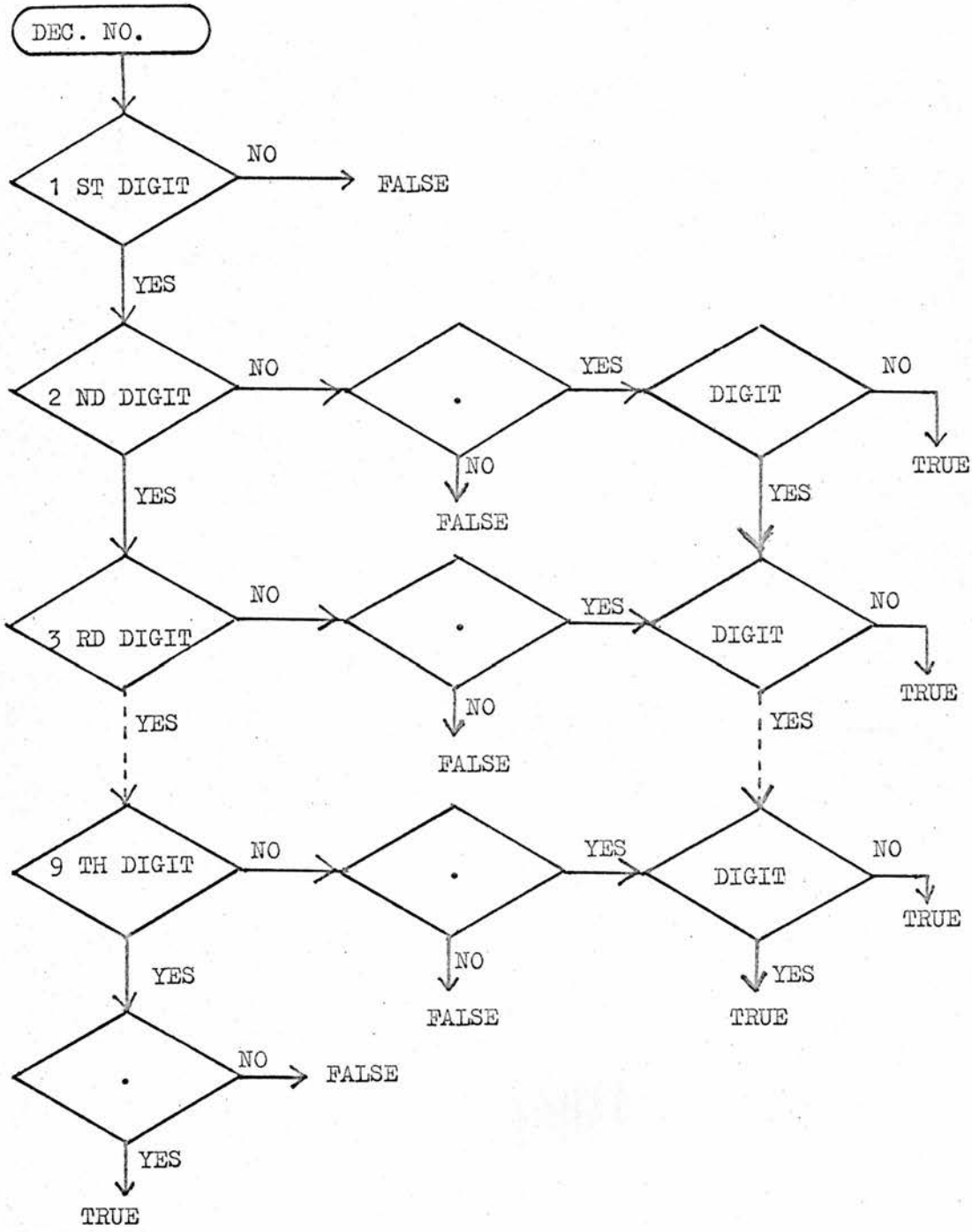


FIGURE II.B RECOGNIZER FOR "<DECIMAL NUMBER>"

APPENDIX I.

This appendix contains the listing of the syntax program for the grammar of the language defined in Chapter 7 on pages 104 to 106. It is a copy of the listing produced by the Grammar Assembly Program. The symbol table is shown first followed by the syntax program. Sub-goal names have been abbreviated to a maximum of five characters. Each sub-goal begins with an abbreviated name and ends before the next lexicographic abbreviated name. All two character symbols are labels and not sub-goal names.

The full names for the abbreviations, written in small letters, were not produced by the Grammar Assembly Program, but are added to make abbreviated names easily identifiable.

Symbol	Location	Abbreviation for
START	12000	
PROGR	12009	program
A1	12107	
IDENT	12109	identifier
D2	12120	
D3	12145	
D4	12151	
D1	12168	
NUMBR	12170	number
E3	12209	
E2	12223	

E1	12238	
FRACT	12240	fraction
F1	12271	
INTEG	12273	integer
G2	12284	
G3	12306	
G1	12315	
VARBL	12317	variable
LABEL	12324	label
EXPRN	12331	expression
J2	12345	
J3	12428	
J4	12497	
J5	12508	
J6	12582	
J1	12593	
TERM	12595	term
K2	12609	
K3	12692	
K4	12761	
K1	12770	
FACTR	12772	factor
L2	12786	
L3	12873	
L1	12882	

PRIMY	12884	primary
S2	12954	
S4	12958	
S1	13061	
STMNT	13063	statement
M1	13095	
LBLLS	13097	label list
T2	13099	
T3	13179	
T1	13188	
BSTMT	13190	basic statement
N1	13225	
SETST	13227	set statement
O1	13310	
GOTST	13312	go to statement
P1	13410	
IFST	13412	if statement
Q1	13554	
COND	13556	condition
R2	13605	
R3	13654	
R1	13701	
DECLN	13703	declaration
W1	13732	
VARLS	13734	variable list
X2	13792	

X3	13872	
X1	13881	
DECLS	13883	declaration list
U2	13897	
U3	13933	
U1	13942	
STLST	13944	statement list
V2	13958	
V3	13994	
V1	14043	

START	CALL	PROGR	12000	11	12009
	STOP		12007	12	
PROGR	CALL	DECLS	12009	11	13883
	FALSE	A1	12016	14	12107
	MATCH	⊄	12023	05	
	FALSE	A1	12027	14	12107
	CALL	STLST	12034	11	13944
	FALSE	A1	12041	14	12107
	MATCH	⊄	12048	05	
	FALSE	A1	12052	14	12107
	EDIT	C	12059	06	
	PRINT	*	12063	07	
	PRINT	E	12067	07	

	PRINT	N	12071	07	
	PRINT	D	12075	07	
	PRINT	.	12079	07	
	EDIT	C	12083	06	
	EDIT	C	12087	06	
	EDIT	C	12091	06	
	EDIT	C	12095	06	
	EDIT	C	12099	06	
	EDIT	W	12103	06	
A1	RETURN		12107	02	
IDENT	ALPHA		12109	20	
	FALSE	D1	12111	14	12168
	COPY		12118	00	
D2	CALL		12120	11	00000
	ALPHA		12127	20	
	TRUE	D3	12129	13	12145
	DIGIT		12136	19	
	FALSE	D4	12138	14	12151
D3	COPY		12145	00	
	EDIT	C	12147	06	
D4	RETURN		12151	02	
	FLAG	D2	12153	15	12120
	PRINT	,	12160	07	
	EDIT	C	12164	06	
D1	RETURN		12168	02	

```
NUMBR   CALL    INTEG       12170  11  12273
        FALSE   E3          12177  14  12209
        CALL    FRACT       12184  11  12240
        FALSE   E2          12191  14  12223
        EDIT    C           12198  06
        BRANCH  E2          12202  16  12223
E3      CALL    FRACT       12209  11  12240
        FALSE   E1          12216  14  12238
E2      PRINT   ,           12223  07
        EDIT    C           12227  06
        FLAG    E1          12231  15  12238
E1      RETURN              12238  02
FRACT   MATCH   .           12240  05
        FALSE   F1          12244  14  12271
        COPY                12251  00
        CALL    INTEG       12253  11  12273
        FALSE   F1          12260  14  12271
        EDIT    C           12267  06
F1      RETURN              12271  02
INTEG   DIGIT               12273  19
        FALSE   G1          12275  14  12315
        COPY                12282  00
G2      CALL                12284  11  00000
        DIGIT               12291  19
        FALSE   G3          12293  14  12306
```

```
        COPY                12300  00
        EDIT    C           12302  06
G3      RETURN              12306  02
        FLAG    G2          12308  15  12284
G1      RETURN              12315  02
VARBL   BRANCH  IDENT       12317  16  12109
LABEL   BRANCH  IDENT       12324  16  12109
EXPRN   CALL    TERM        12331  11  12595
        FALSE   J5          12338  14  12508
J2      CALL                12345  11  00000
        MATCH   +           12352  05
        FALSE   J3          12356  14  12428
        CALL    TERM        12363  11  12595
        FALSE   J4          12370  14  12497
        PRINT   *           12377  07
        PRINT   A           12381  07
        PRINT   D           12385  07
        PRINT   D           12389  07
        PRINT   ,           12393  07
        EDIT    C           12397  06
        EDIT    C           12401  06
        EDIT    C           12405  06
        EDIT    C           12409  06
        EDIT    C           12413  06
        EDIT    C           12417  06
        BRANCH  J4          12421  16  12497
```

```
J3      MATCH   -           12428   05
        FALSE   J4          12432   14   12497
        CALL    TERM        12439   11   12595
        FALSE   J4          12446   14   12497
        PRINT   *           12453   07
        PRINT   S           12457   07
        PRINT   U           12461   07
        PRINT   B           12465   07
        PRINT   ,           12469   07
        EDIT    C           12473   06
        EDIT    C           12477   06
        EDIT    C           12481   06
        EDIT    C           12485   06
        EDIT    C           12489   06
        EDIT    C           12493   06
J4      RETURN              12497   02
        FLAG    J2          12499   15   12345
        RETURN              12506   02
J5      MATCH               12508   05
        FALSE   J6          12512   14   12528
        CALL    EXPRN       12519   11   12331
        RETURN              12526   02
J6      MATCH   -           12528   05
        FALSE   J1          12532   14   12593
        CALL    EXPRN       12519   11   12331
        RETURN              12526   02
J6      MATCH   -           12528   05
```

	FALSE	J1	12532	14	12593
	CALL	EXPRN	12539	11	12331
	FALSE	J1	12546	14	12593
	PRINT	*	12553	07	
	PRINT	N	12557	07	
	PRINT	E	12561	07	
	PRINT	G	12565	07	
	PRINT	,	12569	07	
	EDIT	C	12573	06	
	EDIT	C	12577	06	
	EDIT	C	12581	06	
	EDIT	C	12585	06	
	EDIT	C	12589	06	
J1	RETURN		12593	02	
TERM	CALL	FACTR	12595	11	12772
	FALSE	K1	12602	14	12770
K2	CALL		12609	11	00000
	MATCH	*	12616	05	
	FALSE	K3	12620	14	12692
	CALL	FACTR	12627	11	12772
	FALSE	K4	12634	14	12761
	PRINT	*	12641	07	
	PRINT	M	12645	07	
	PRINT	U	12649	07	
	PRINT	L	12653	07	

	PRINT	,	12657	07	
	EDIT	C	12661	06	
	EDIT	C	12665	06	
	EDIT	C	12669	06	
	EDIT	C	12673	06	
	EDIT	C	12677	06	
	EDIT	C	12681	06	
	BRANCH	K4	12685	16	12761
K3	MATCH	/	12692	05	
	FALSE	K4	12696	14	12761
	CALL	FACTR	12703	11	12772
	FALSE	K4	12710	14	12761
	PRINT	*	12717	07	
	PRINT	D	12121	07	
	PRINT	I	12725	07	
	PRINT	V	12729	07	
	PRINT	,	12733	07	
	EDIT	C	12737	06	
	EDIT	C	12741	06	
	EDIT	C	12745	06	
	EDIT	C	12749	06	
	EDIT	C	12753	06	
	EDIT	C	12757	06	
K4	RETURN		12761	02	
	FLAG	K2	12763	15	12609

```
K1      RETURN                  12770   02
FACTR   CALL    PRIMY           12772   11  12884
        FALSE   L1              12779   14  12882
L2      CALL                    12786   11  00000
        MATCH   *               12793   05
        FALSE   L3              12797   14  12873
        MATCH   *               12804   05
        FALSE   L3              12808   14  12873
        CALL    PRIMY           12815   11  12884
        FALSE   L3              12822   14  12873
        PRINT   *               12829   07
        PRINT   E               12833   07
        PRINT   X               12837   07
        PRINT   P               12841   07
        PRINT   ,               12845   07
        EDIT    C               12849   06
        EDIT    C               12853   06
        EDIT    C               12857   06
        EDIT    C               12861   06
        EDIT    C               12865   06
        EDIT    C               12869   06
L3      RETURN                  12873   02
        FLAG    L2              12875   15  12786
L1      RETURN                  12882   02
PRIMY   CALL    VARBL           12884   11  12317
```

```
        FALSE   S2          12891   14  12940
        PRINT   *           12898   07
        PRINT   C           12902   07
        PRINT   L           12906   07
        PRINT   A           12910   07
        PRINT   ,           12914   07
        EDIT    C           12918   06
        EDIT    C           12922   06
        EDIT    C           12926   06
        EDIT    C           12930   06
        EDIT    C           12934   06
        RETURN              12938   02
S2      CALL    NUMBR       12940   11  12170
        TRUE    S1          12947   13  13061
S3      MATCH   (           12954   05
        FALSE   S4          12958   14  12985
        CALL    EXPRN       12965   11  12331
        FALSE   S1          12972   14  13061
        MATCH   )           12979   05
        RETURN              12983   02
S4      MATCH   /           12985   05
        FALSE   S1          12989   14  13061
        CALL    EXPRN       12996   11  12331
        FALSE   S1          13003   14  13061
        MATCH   /           13010   05
        FALSE   S1          13014   14  13061
```

	PRINT	*	13021	07	
	PRINT	A	13025	07	
	PRINT	B	13029	07	
	PRINT	S	13033	07	
	PRINT	,	13037	07	
	EDIT	C	13041	06	
	EDIT	C	13045	06	
	EDIT	C	13049	06	
	EDIT	C	13053	06	
	EDIT	C	13057	06	
S1	RETURN		13061	02	
STMNT	CALL	LBLLS	13063	11	13097
	FALSE	M1	13070	14	13095
	CALL	BSTMT	13077	11	13190
	FALSE	M1	13084	14	13095
	EDIT	C	13091	06	
M1	RETURN		13095	02	
LBLLS	NULL		13097	01	
T2	CALL		13099	11	00000
	CALL	LABEL	13106	11	12324
	FALSE	T3	13113	14	13179
	MATCH	.	13120	05	
	FALSE	T3	13124	14	13179
	PRINT	*	13131	07	

	PRINT	L	13135	07	
	PRINT	A	13139	07	
	PRINT	B	13143	07	
	PRINT	,	13147	07	
	EDIT	C	13151	06	
	EDIT	C	13155	06	
	EDIT	C	13159	06	
	EDIT	C	13163	06	
	EDIT	X	13167	06	
	EDIT	C	13171	06	
	EDIT	C	13175	06	
T3	RETURN		13179	02	
	FLAG	T2	13181	15	13099
T1	RETURN		13188	02	
BSTMT	CALL	IFST	13190	11	13412
	TRUE	N1	13197	13	13225
	CALL	GOTST	13204	11	13312
	TRUE	N1	13211	13	13225
	BRANCH	SETST	13218	16	13227
N1	RETURN		13225	02	
SETST	CALL	VARBL	13227	11	12317
	FALSE	01	13234	14	13310
	MATCH	=	13241	05	
	FALSE	01	13245	14	13310
	CALL	EXPRN	13252	11	12331

	FALSE	01	13259	14	13310
	PRINT	*	13266	07	
	PRINT	S	13270	07	
	PRINT	T	13274	07	
	PRINT	O	13278	07	
	PRINT	,	13282	07	
	EDIT	C	13286	06	
	EDIT	C	13290	06	
	EDIT	C	13294	06	
	EDIT	C	13298	06	
	EDIT	C	13302	06	
	EDIT	C	13306	06	
01	RETURN		13310	02	
GOTST	MATCH	G	13312	05	
	FALSE	P1	13316	14	13410
	MATCH	O	13323	05	
	FALSE	P1	13327	14	13410
	MATCH	T	13334	05	
	FALSE	P1	13338	14	13410
	MATCH	O	13345	05	
	FALSE	P1	13349	14	13410
	CALL	LABEL	13356	11	12324
	FALSE	P1	13363	14	13410
	PRINT	*	13370	07	
	PRINT	T	13374	07	

	PRINT	R	13378	07	
	PRINT	A	13382	07	
	PRINT	,	13386	07	
	EDIT	C	13390	06	
	EDIT	C	13394	06	
	EDIT	C	13398	06	
	EDIT	C	13402	06	
	EDIT	C	13406	06	
P1	RETURN		13410	02	
IFST	MATCH	I	13412	05	
	FALSE	Q1	13416	14	13554
	MATCH	F	13423	05	
	FALSE	Q1	13427	14	13554
	CALL	EXPRN	13434	11	12331
	FALSE	Q1	13441	14	13554
	MATCH	=	13448	05	
	FALSE	Q1	13452	14	13554
	CALL	COND	13459	11	13556
	FALSE	Q1	13466	14	13554
	MATCH	,	13473	05	
	FALSE	Q1	13477	14	13554
	MATCH	G	13484	05	
	FALSE	Q1	13488	14	13554
	MATCH	O	13495	05	
	FALSE	Q1	13499	14	13554

```
        MATCH    T          13506  05
        FALSE    Q1         13510  14  13554
        MATCH    0          13517  05
        FALSE    Q1         13521  14  13554
        CALL     LABEL      13528  11  12324
        FALSE    Q1         13535  14  13554
        EDIT     X          13542  06
        EDIT     C          13546  06
        EDIT     C          13550  06
Q1      RETURN              13554  02
COND    MATCH    0          13556  05
        FALSE    R2         13560  14  13605
        PRINT    *          13567  07
        PRINT    T          13571  07
        PRINT    Z          13575  07
        PRINT    E          13579  07
        PRINT    ,          13583  07
        EDIT     C          13587  06
        EDIT     C          13591  06
        EDIT     C          13595  06
        EDIT     C          13599  06
        RETURN              13603  02
R2      MATCH    +          13605  05
        FALSE    R3         13609  14  13654
        PRINT    *          13616  07
```

	PRINT	T	13620	07	
	PRINT	P	13624	07	
	PRINT	L	13628	07	
	PRINT	,	13632	07	
	EDIT	C	13636	07	
	EDIT	C	13640	07	
	EDIT	C	13644	07	
	EDIT	C	13648	07	
	RETURN		13652	02	
R3	MATCH	-	13654	05	
	FALSE	R1	13658	14	13701
	PRINT	*	13665	07	
	PRINT	T	13669	07	
	PRINT	M	13673	07	
	PRINT	I	13677	07	
	PRINT	,	13681	07	
	EDIT	C	13685	06	
	EDIT	C	13689	06	
	EDIT	C	13693	06	
	EDIT	C	13697	06	
R1	RETURN		13701	02	
DECLN	MATCH	(	13703	05	
	FALSE	W1	13707	14	13732
	CALL	VARLS	13714	11	13734
	FALSE	W1	13721	14	13732

```
        MATCH    )          13728  05
W1      RETURN              13732  02
VARLS   CALL     VARBL      13734  11  12317
        FALSE    X1         13741  14  13881
        PRINT    *          13748  07
        PRINT    V          13752  07
        PRINT    A          13756  07
        PRINT    R          13760  07
        PRINT    ,          13764  07
        EDIT     C          13768  06
        EDIT     C          13772  06
        EDIT     C          13776  06
        EDIT     C          13780  06
        EDIT     X          13784  06
        EDIT     C          13788  06
X2      CALL                13792  11  00000
        MATCH    ,          13799  05
        FALSE    X3         13803  14  13872
        CALL     VARBL      13810  11  12317
        FALSE    X3         13817  14  13872
        PRINT    *          13824  07
        PRINT    V          13828  07
        PRINT    A          13832  07
        PRINT    R          13836  07
        PRINT    ,          13840  07
```

	EDIT	C	13844	06	
	EDIT	C	13848	06	
	EDIT	C	13852	06	
	EDIT	C	13856	06	
	EDIT	X	13860	06	
	EDIT	C	13864	06	
	EDIT	C	13868	06	
X3	RETURN		13872	02	
	FLAG	X2	13874	15	13792
X1	RETURN		13881	02	
DECLS	CALL	DECLN	13883	11	13703
	FALSE	U1	13890	14	13942
U2	CALL		13897	11	00000
	MATCH	⊄	13904	05	
	FALSE	U3	13908	14	13933
	CALL	DECLN	13915	11	13703
	FALSE	U3	13922	14	13933
	EDIT	C	13929	06	
U3	RETURN		13933	02	
	FLAG	U2	13935	15	13897
U1	RETURN		13942	02	
STLST	CALL	STMNT	13944	11	13063
	FALSE	V1	13951	14	14043
V2	CALL		13958	11	00000

```
        MATCH    $          13965  05
        FALSE    V3         13969  14  13994
        CALL     STMNT      13976  11  13063
        FALSE    V3         13983  14  13994
        EDIT     C          13990  06
V3      RETURN              13994  02
        FLAG     V2         13996  15  13958
        PRINT    *          14003  07
        PRINT    H          14007  07
        PRINT    L          14011  07
        PRINT    T          14015  07
        PRINT    ,          14019  07
        EDIT     C          14023  06
        EDIT     C          14027  06
        EDIT     C          14031  06
        EDIT     C          14035  06
        EDIT     C          14039  06
V1      RETURN              14043  02
END
```

0430 CARDS PROCESSED, 02045 CORE POSITIONS REQUIRED

APPENDIX II.

SYNTAX PROGRAM FOR THE SYNTAX CHECKER FOR BASIC

A Formal Definition of Dartmouth Basic†

⟨*alphabetic character*⟩ := A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
⟨*digit*⟩ := 0|1|2|3|4|5|6|7|8|9
⟨*special character*⟩ := +|−|*|/|b|=|(|)|>|<|.|,|;
⟨*integer*⟩ := $\{\langle digit\rangle\}_1^9$
⟨*fraction*⟩ := .⟨*digit*⟩
⟨*decimal number*⟩ := $\{\langle digit\rangle\}_1^{n\leq 9}$. $\{\langle digit\rangle\}_0^{9-n}$

Note: A decimal number could not be defined as

⟨*decimal number*⟩ := ⟨*integer*⟩.⟨*integer*⟩

since (a) no more than nine digits are permitted in a number, whereas the above construct would allow a maximum of 18 and (b) since an ⟨*integer*⟩ must contain at least one digit, the form $\{\langle digit\rangle\}$. is not permitted.

⟨*sign*⟩ := ⟨*null*⟩|−
⟨*exponent*⟩ := E⟨*sign*⟩$\{\langle digit\rangle\}_1^2$
⟨*number*⟩ := ⟨*integer*⟩|⟨*fraction*⟩|⟨*decimal number*⟩|
⟨*number*⟩⟨*exponent*⟩
⟨*signed number*⟩ := ⟨*sign*⟩⟨*number*⟩
⟨*simple variable*⟩ := ⟨*alphabetic character*⟩$\{\langle digit\rangle\}_0^1$
⟨*subscripted variable*⟩ :=
⟨*alphabetic character*⟩(⟨*expression*⟩$\{$,⟨*expression*⟩$\}_0^1$)
⟨*variable*⟩ := ⟨*simple variable*⟩|⟨*subscripted variable*⟩
⟨*function name*⟩ := SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT|RND|
FN⟨*alphabetic character*⟩
⟨*function term*⟩ := ⟨*function name*⟩(⟨*expression*⟩)
⟨*term*⟩ := ⟨*number*⟩|⟨*variable*⟩|⟨*function term*⟩|(⟨*expression*⟩)
⟨*involution factor*⟩ := ⟨*term*⟩|⟨*term*⟩↑⟨*term*⟩
⟨*multiply factor*⟩ := ⟨*involution factor*⟩|
⟨*multiply factor*⟩$\{*|/\}_1^1$⟨*involution factor*⟩
⟨*expression*⟩ := ⟨*multiply factor*⟩|⟨*sign*⟩⟨*expression*⟩|
⟨*expression*⟩$\{+|-\}_1^1$⟨*multiply factor*⟩

† "BASIC" (A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time-Sharing System), Dartmouth College, Jan. 1, 1965.

<assignment statement> := LET<variable>=<expression>
<read list> := <variable>{,<variable>}$_0^\infty$
<READ statement> := READ<read list>
<number list> := <signed number>{,<signed number>}$_0^\infty$
<DATA statement> := DATA<number list>
<message> := "{<alphabetic character>|<digit>|<special character>}$_1^\infty$"
<print item> := <expression>|<message>|<message><expression>
<print list> := <null>|<print item>{,<print item>}$_0^\infty${,}$_0^1$
<PRINT statement> := PRINT<print list>
<line no 1> := {<digit>}$_1^3$
<GO TO statement> := GO{b}$_0^1$TO<line no 1>
<comment> := REM{<alphabet character>|<digit>|<special character>}$_0^\infty$
<relation op> := >=|>|<>|<|<=|=
<IF statement> := IF<expression><relation op><expression>THEN
<line no 1>
<FOR statement> := FOR<simple variable>=<expression>TO<expression>
{STEP<expression>}$_0^1$
<NEXT statement> := NEXT<simple variable>
<END statement> := END
<size> := <integer>{,<integer>}$_0^1$
<dimension variable> := <alphabetic character>(<size>)
<DIMension statement> := DIM<dimension variable>
{,<dimension variable>}$_0^\infty$
<DEFine statement> := DEFbFN<alphabetic character>(<simple variable>)
=<expression>
<GOSUB statement> := GOSUB<line no 1>
<RETURN statement> := RETURN
<statement body> := <assignment statement>|<READ statement>|
<DATA statement>|<PRINT statement>|
<GO TO statement>|<IF statement>|
<FOR statement>|<NEXT statement>|
<DIMension statement>|<DEFine statement>|
<GOSUB statement>|<RETURN statement>|
<comment>
<line number> := {<digit>}$_1^3$b
<BASIC statement> := <line number><statement body>
<BASIC program> := {<BASIC statement>}$_1^\infty$
<line number><END statement>

The following definitions from the formal definition of Basic on pages 137 and 138 have re-orded or altered. The right hand part is the re-ordered or altered part.

<number> ::= {<decimal number> | <integer> | <fraction>} {<exponent>}$_0^1$

<special character> ::= + | - | * | / | = | (|) | . | , | $

<expression> ::= <sign> <multiply factor> $\int$ {+ -}$_1^1$ <multiply factor>

<involution factor> ::= <term> ** <term>

<multiply factor> ::= <involution factor> $\int$ {* | /}$_1^1$ <involution factor>

<relation operator> ::= .GE. | .GT. | .NE. | .LE. | .EQ.

<message> ::= @ {<alphabetic character> | <digit> | <special character>}$_1^\infty$ @

<BASIC statement> ::= <line number> <statement body> $

<variable> ::= <subscripted variable> | <simple variable>

<term> ::= <number> | <function term> | <variable> | (<expression>)

Abbreviations used for goal and sub-goal names in the syntax program are listed below.

PROG	program
BASIC	BASIC program
BSTMT	BASIC statement
LNNO1	line no 1
STBDY	statement body
RETRN	RETURN statement
GOSUB	GOSUB statement
DEFFN	DEFine statement

DIMEN	DIMension statement
DIMUR	dimension variable
SIZE	size
ENDST	END statement
NEXTS	NEXT statement
FORST	FOR statement
IFSTM	IF statement
RELOP	relation operator
COMEN	comment
GOTOS	GO TO statement
PRINT	PRINT statement
MESGE	message
DATAS	DATA statement
NOLST	number list
READS	READ statement
RDLST	read list
ASIGN	assignment statement
EXPRN	expression
MULFC	multiply factor
INVFC	invalution factor
TERM	term
FNTRM	function term
FNAME	function name
VARBL	variable

SIMVR	simple variable
SUBVR	subscripted variable
SGNUM	signed number
NUMBR	number
EXPON	exponent
SIGN	sign
DECNO	decimal number
FRACT	fraction
INTEG	integer
SPCHR	special character

The Syntax Program

```
PROG    CALL    BASIC
        STOP
BASIC   CALL    BSTMT
        FLAG    A2
        ERROR   01
A4      MATCH   ∅
        FLAG    A2
        NEXT
        BRANCH  A4
A2      CALL
        CALL    BSTMT
        FLAG    A3
        CALL    LNNO1
```

```
                FALSE     A5
                CALL      ENDST
                FLAG      A7
A5              ERROR     01
A6              MATCH     $
                FLAG      A3
                NEXT
                BRANCH    A6
A3              RETURN
                FLAG      A2
A7              RETURN
A1              RETURN
LNNO1           DIGIT
                FALSE     B1
                CALL
                DIGIT
                RETURN
                FALSE     B2
                CALL
                DIGIT
                RETURN
B2              FLAG      B1
B1              RETURN
BSTMT           CALL      LNNO1
```

```
                FALSE     C1
                NEXT
                CALL      STBDY
                FALSE     C1
                MATCH     $
C1              RETURN
STBDY           CALL      ASIGN
                TRUE      D1
                CALL      IFSTM
                TRUE      D1
                CALL      FORST
                TRUE      D1
                CALL      GOTOS
                TRUE      D1
                CALL      GOSUB
                TRUE      D1
                CALL      NEXTS
                TRUE      D1
                CALL      COMEN
                TRUE      D1
                CALL      RETRN
                TRUE      D1
                CALL      PRINT
                TRUE      D1
                CALL      READS
```

```
                TRUE        D1
                CALL        DEFFN
                TRUE        D1
                CALL        DATAS
                TRUE        D1
                CALL        DIMEN
D1              RETURN
RETRN           MATCH       R
                FALSE       E1
                MATCH       E
                FALSE       E1
                MATCH       T
                FALSE       E1
                MATCH       U
                FALSE       E1
                MATCH       R
                FALSE       E1
                MATCH       N
E1              RETURN
GOSUB           MATCH       G
                FALSE       F1
                MATCH       O
                FALSE       F1
                MATCH       S
                FALSE       F1
```

```
                MATCH       U
                FALSE       F1
                MATCH       B
                FALSE       F1
                BRANCH      LNNO1
F1              RETURN
DEFFN           MATCH       D
                FALSE       G1
                MATCH       E
                FALSE       G1
                MATCH       F
                FALSE       G1
                NEXT
                MATCH       F
                FALSE       G1
                MATCH       N
                FALSE       G1
                ALPHA
                FALSE       G1
                MATCH       (
                FALSE       G1
                CALL        SIMVR
                FALSE       G1
                MATCH       )
                FALSE       G1
```

```
                MATCH       =
                FALSE       G1
                BRANCH      EXPRN
G1              RETURN
DIMEN           MATCH       D
                FALSE       H1
                MATCH       I
                FALSE       H1
                MATCH       M
                FALSE       H1
                CALL        DIMVR
                FALSE       H1
H2              CALL
                MATCH       ,
                FALSE       H3
                CALL        DIMVR
H3              RETURN
                FLAG        H2
H1              RETURN
DIMVR           ALPHA
                FALSE       I1
                MATCH       (
                FALSE       I1
                CALL        SIZE
                FALSE       I1
```

```
                MATCH       )
I1              RETURN
SIZE            CALL        INTEG
                FALSE       J1
                CALL
                MATCH       ,
                FALSE       J3
                CALL        INTEG
J3              RETURN
                FLAG        J1
J1              RETURN
ENDST           MATCH       E
                FALSE       K1
                MATCH       N
                FALSE       K1
                MATCH       D
K1              RETURN
NEXTS           MATCH       N
                FALSE       L1
                MATCH       E
                FALSE       L1
                MATCH       X
                FALSE       L1
                MATCH       T
                FALSE       L1
```

```
            BRANCH      SIMVR
L1          RETURN
FORST       MATCH       F
            FALSE       M1
            MATCH       O
            FALSE       M1
            MATCH       R
            FALSE       M1
            CALL        SIMVR
            FALSE       M1
            MATCH       =
            FALSE       M1
            CALL        EXPRN
            FALSE       M1
            MATCH       T
            FALSE       M1
            MATCH       O
            FALSE       M1
            CALL        EXPRN
            FALSE       M1
            CALL
            MATCH       S
            FALSE       M2
            MATCH       T
            FALSE       M2
```

```
        MATCH   E
        FALSE   M2
        MATCH   P
        FALSE   M2
        CALL    EXPRN
M2      RETURN
        FLAG    M1
M1      RETURN
IFSTM   MATCH   I
        FALSE   N1
        MATCH   F
        FALSE   N1
        CALL    EXPRN
        FALSE   N1
        CALL    RELOP
        FALSE   N1
        CALL    EXPRN
        FALSE   N1
        MATCH   T
        FALSE   N1
        MATCH   H
        FALSE   N1
        MATCH   E
        FALSE   N1
        MATCH   N
```

```
                BRANCH      LNNO1
N1              RETURN
RELOP           MATCH       .
                FALSE       P1
                MATCH       E
                FALSE       P2
                MATCH       Q
                FALSE       P1
                MATCH       .
                RETURN
P2              MATCH       G
                FALSE       P3
P6              MATCH       E
                FALSE       P4
                BRANCH      P5
P4              MATCH       T
                FALSE       P1
                BRANCH      P5
P3              MATCH       L
                FLAG        P6
                MATCH       N
                FALSE       P1
                MATCH       E
                FALSE       P1
P5              MATCH       .
P1              RETURN
```

```
COMEN   MATCH   R
        FALSE   Q1
        MATCH   E
        FALSE   Q1
        MATCH   M
        FALSE   Q1
Q2      CALL
        ALPHA
        FLAG    Q3
        DIGIT
        FLAG    Q3
        CALL    SPCHR
Q3      RETURN
        FLAG    Q2
Q1      RETURN
GOTOS   MATCH   G
        FALSE   R1
        MATCH   O
        FALSE   R1
        MATCH   T
        FALSE   R1
        MATCH   O
        FALSE   R1
        BRANCH  LNNO1
R1      RETURN
```

```
PRINT   MATCH   P
        FALSE   S1
        MATCH   R
        FALSE   S1
        MATCH   I
        FALSE   S1
        MATCH   N
        FALSE   S1
        MATCH   T
        FALSE   S1
        BRANCH  PRLST
S1      RETURN
PRLST   CALL    PRITM
        FLAG    T1
        RETURN
T1      CALL
        MATCH   ,
        FALSE   T2
        CALL    PRITM
T2      RETURN
        FLAG    T1
        MATCH   ,
        FLAG    T3
T3      RETURN
PRITM   CALL    MESGE
```

```
                TRUE        U2
                BRANCH      EXPRN
U2              CALL        EXPRN
                FLAG        U3
U3              RETURN
MESGE           MATCH       @
                FALSE       V1
                ALPHA
                FLAG        V2
                DIGIT
                FLAG        V2
                CALL        SPCHR
                FALSE       V1
V2              CALL
                ALPHA
                FLAG        V3
                DIGIT
                FLAG        V3
                CALL        SPCHR
V3              RETURN
                FLAG        V2
                MATCH       @
V1              RETURN
DATAS           MATCH       D
                FALSE       W1
```

```
                MATCH       A
                FALSE       W1
                MATCH       T
                FALSE       W1
                MATCH       A
                FALSE       W1
                BRANCH      NOLST
W1              RETURN
NOLST           CALL        SGNUM
                FALSE       X1
X2              CALL
                MATCH       ,
                FALSE       X3
                CALL        SGNUM
X3              RETURN
                FLAG        X2
X1              RETURN
READS           MATCH       R
                FALSE       Y1
                MATCH       E
                FALSE       Y1
                MATCH       A
                FALSE       Y1
                MATCH       D
                FALSE       Y1
                BRANCH      RDLST
```

```
Y1          RETURN
RDLST       CALL        VARBL
            FALSE       Z1
Z2          CALL
            MATCH       ,
            FALSE       Z3
            CALL        VARBL
Z3          RETURN
            FLAG        Z2
Z1          RETURN
ASIGN       MATCH       L
            FALSE       AB
            MATCH       E
            FALSE       AB
            MATCH       T
            FALSE       AB
            CALL        VARBL
            FALSE       AB
            MATCH       =
            FALSE       AB
            BRANCH      EXPRN
AB          RETURN
EXPRN       CALL        SIGN
            CALL        MULFC
```

```
                FALSE       BA
BB              CALL
                MATCH       +
                FLAG        BC
                MATCH       -
                FALSE       BD
BC              CALL        MULFC
BD              RETURN
                FLAG        BB
BA              RETURN
MULFC           CALL        INVFC
                FALSE       CA
CB              CALL
                MATCH       *
                FLAG        CC
                MATCH       /
                FALSE       CD
CC              CALL        INVFC
CD              RETURN
                FLAG        CB
CA              RETURN
INVFC           CALL        TERM
                FALSE       DA
                CALL
                MATCH       *
                FALSE       DB
                MATCH       *
```

```
                FALSE     DB
                CALL      TERM
DB              RETURN
                FLAG      DA
DA              RETURN
TERM            CALL      NUMBR
                TRUE      EA
                CALL      FNTRM
                TRUE      EA
                CALL      VARBL
                TRUE      EA
                MATCH     (
                FALSE     EA
                CALL      EXPRN
                FALSE     EA
                MATCH     )
EA              RETURN
FNTRM           CALL      FNAME
                FALSE     FA
                MATCH     (
                FALSE     FA
                CALL      EXPRN
                FALSE     FA
                MATCH     )
FA              RETURN
```

```
FNAME   MATCH   F
        FALSE   GB
        MATCH   N
        FALSE   GA
        ALPHA
        FALSE   GA
        RETURN
GB      MATCH   A
        FALSE   GSIN
        MATCH   B
        FALSE   GT
GS      MATCH   S
        RETURN
GSIN    MATCH   S
        FALSE   GCOS
        MATCH   I
        FALSE   GSQR
GN      MATCH   N
        RETURN
GT      MATCH   T
        FALSE   GA
        BRANCH  GN
GCOS    MATCH   C
        FALSE   GTAN
        MATCH   O
```

```
                FALSE       GA
                BRANCH      GS
GTAN            MATCH       T
                FALSE       GEXP
                MATCH       A
                FALSE       GA
                BRANCH      GN
GEXP            MATCH       E
                FALSE       GLOG
                MATCH       X
                FALSE       GA
                MATCH       P
                RETURN
GLOG            MATCH       L
                FALSE       GINT
                MATCH       O
                FALSE       GA
                MATCH       G
                RETURN
GSQR            MATCH       Q
                FALSE       GA
                MATCH       R
                RETURN
GINT            MATCH       I
                FALSE       GRND
                MATCH       N
```

```
                FALSE     GA
                MATCH     T
                RETURN
GRND            MATCH     R
                FALSE     GA
                MATCH     N
                FALSE     GA
                MATCH     D
GA              RETURN
VARBL           CALL      SUBVR
                TRUE      HA
                BRANCH    SIMVR
HA              RETURN
SUBVR           ALPHA
                FALSE     JA
                MATCH     (
                FALSE     JA
                CALL      EXPRN
                FALSE     JA
                CALL
                MATCH     ,
                FALSE     JB
                CALL      EXPRN
JB              RETURN
```

```
                FLAG        JC
JC              MATCH       )
JA              RETURN
SIMVR           ALPHA
                FALSE       KA
                DIGIT
                FLAG        KA
KA              RETURN
SGNUM           CALL        SIGN
                FLASE       LA
                BRANCH      NUMBR
LA              RETURN
NUMBR           CALL        DECNO
                FLAG        MB
                CALL        INTEG
                FLAG        MB
                CALL        FRACT
                FALSE       MA
MB              CALL        EXPON
                FLAG        MA
MA              RETURN
EXPON           MATCH       E
                FALSE       NA
                CALL        SIGN
```

```
                FALSE       NA
                DIGIT
                FALSE       NA
                DIGIT
                FLAG        NA
NA              RETURN
SIGN            MATCH       -
                FLAG        PA
PA              RETURN
DECNO           DIGIT
                FALSE       QK
                DIGIT
                FLAG        QB
                MATCH       .
                FALSE       QK
                BRANCH      DEC8
QB              DIGIT
                FLAG        QC
                MATCH       .
                FALSE       QK
                BRANCH      DEC7
QC              DIGIT
                FLAG        QD
                MATCH       .
                FALSE       QK
```

```
            BRANCH      DEC6
QD          DIGIT
            FLAG        QE
            MATCH       .
            FALSE       QK
            BRANCH      DEC5
QE          DIGIT
            FLAG        QF
            MATCH       .
            FALSE       QK
            BRANCH      DEC4
QF          DIGIT
            FLAG        QG
            MATCH       .
            FALSE       QK
            BRANCH      DEC3
QG          DIGIT
            FLAG        QH
            MATCH       .
            FALSE       QK
            BRANCH      DEC2
QH          DIGIT
            FLAG        QI
            MATCH       .
            FALSE       QK
```

```
                BRANCH      DEC1
QI              MATCH       .
                FALSE       QK
                RETURN
DEC8            DIGIT
                FALSE       QJ
DEC7            DIGIT
                FALSE       QJ
DEC6            DIGIT
                FALSE       QJ
DEC5            DIGIT
                FALSE       QJ
DEC4            DIGIT
                FALSE       QJ
DEC3            DIGIT
                FALSE       QJ
DEC2            DIGIT
                FLASE       QJ
DEC1            DIGIT
QJ              FLAG        QK
QK              RETURN
FRACT           MATCH       .
                FALSE       RA
                DIGIT
RA              RETURN
```

```
INTEG    DIGIT
         FALSE     SA
         DIGIT
         FALSE     SB
         DIGIT
         FLASE     SB
         DIGIT
         FALSE     SB
         DIGIT
         FALSE     SB
         DIGIT
         FALSE     SB
         DIGIT
         FALSE     SB
         DIGIT
         FALSE     SB
         DIGIT
SB       FLAG      SA
SA       RETURN
SPCHR    MATCH     +
         FLAG      TA
         MATCH     -
         FLAG      TA
         MATCH     *
         FLAG      TA
```

```
                MATCH   /
                FLAG    TA
                MATCH   =
                FLAG    TA
                MATCH   (
                FLAG    TA
                MATCH   )
                FLAG    TA
                MATCH   .
                FLAG    TA
                MATCH   ?
TA              RETURN
END
```

REFERENCES

A1 METCALFE, H. H. (1964); A parametrized compiler based on mechanical linguistics: Annual Review, Automatic Programming, Volume 4, p. 125. (R. Goodman, Editor) Pergamon Press.

A2 FELDMAN, J. and Gries, D.; Translator writing systems. Comm. ACM, Volume II, no. 2, February 1968.

A3 GRAHAM, R. M.; Bounded context translation. Proc. AFIPS 1964 SJCC, Volume 25, pp. 17-29.

A4 NAUR, P. (Editor); Report on the Algorithmic language ALGOL 60. Comm. ACM, January 1961.

A5 CHEATHAM, T. E. and SATTLEY, K.; Syntax directed compiling. Proc. AFIPS 1964 SJCC, Volume 25, pp. 31-57.

A6 FLOYD, R. W.; The syntax of programming languages - a survey. IEEE Trans. EC13, Volume 4, August 1964, pp. 346-353.

A7 IRONS, E. T.; A syntax directed compiler for ALGOL 60. Comm. ACM, Volume 4, January 1961, pp. 51-55.

A8 DIJKSTRA, E. W.; Recursive programming. Num. Math., Volume 2, no. 5, pp. 312-318. 1960.

A9 WARSHALL, S. and SHAPIRO, R. M.; A general - purpose table - driven compiler. Proc. AFIPS 1964 SJCC, Volume 25, pp. 59-65.

A10 MILLARD, G. E.; A syntax - directed compiling technique. Royal Aircraft Establishment - Technical report no. 66154, June 1966.

A11 KANNER, H., KOSINKI, P., and ROBINSON, C. L.; The structure of yet another ALGOL compiler. Comm. ACM, Volume 8, July 1965, pp. 427-238.

A12 GLENNIE, A. E.; On the syntax machine and the construction of a universal compiler. Computation Centre, Carnegie Institute of Technology, July 1960.

A13 KUNO, S., and OETTINGER, A. G.; Multiple path syntactic analyser. Information Processing 62 (IFIP Congress), Popplewell (Editor). North Holland Publishing Company, Amsterdam, 1962.

A14 REYNOLDS, J. C.; An introduction to the COGENT programming system. Proc. ACM 20th National Conference, 1965, pp. 422-436.

A15 INGERMAN, P. Z.; A syntax orientated translator. Academic Press, Inc., New York, 1966.

A16 IBM 1620 Monitor II system reference manual. File 1520-36, Form C26-5774-0. Revised May 1964.

A17 REEVES, C. M.; Description of a syntax-directed translator. Computer Journal of the British Computer Society; Volume 10, no. 3, November 1967.

A18 A manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing system, Darthmouth College, January, 1965.

A19 LEE, J. A.; Anatomy of a compiler. Reinhold computer science series. 1968

SUPPLEMENTARY BIBLIOGRAPHY

1. FLORES, I.; Computer Software, Prentice Hall, 1965.

INDEX

Index entries in CAPITALS designate similar entries in the text.