https://ntrs.nasa.gov/search.jsp?R=19720005536 2020-03-11T21:50:03+00:00Z

NASA CR- 122384

STAR

Technical Report TR-168
NGR-21-002-197 and
NGL-21-002-008

September 1971

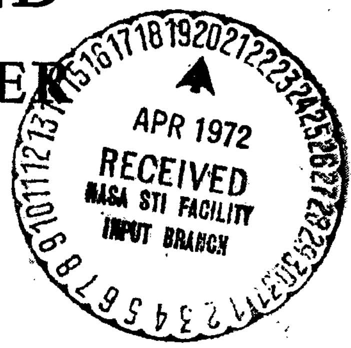
# Architectural Design of an Algol Interpreter

by

Claude K. Jackson

CASE FILE
COPY

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER
COLLEGE PARK, MARYLAND



Technical Report TR-168
NGR-21-002-197 and
NGL-21-002-008

September 1971

# Architectural Design of an Algol Interpreter

by

Claude K. Jackson

This research was supported in part by Grants NGR-21-002-197 and NGL-21-002-008 from the National Aeronautics and Space Administration to the Computer Science Center of the University of Maryland.

## Abstract

This report describes the design of a syntax-directed interpreter for a subset of Algol. It is a conceptual design with sufficient details and completeness but as much independence of implementation as possible. The design includes a detailed description of a scanner, an analyzer described in the Floyd-Evans productions, a hash-coded symbol table, and an executor. Interpretation of sample programs is also provided to show how the interpreter functions.

# Foreword

Programming an Algol interpreter is nothing new, but describing architectural design of an interpreter is. This tutorial paper presents the architectural design of an interpreter for a subset of Algol. In this report, attention is called to the following goals and observations:

(1) to describe the detailed conceptual design of the interpreter, not a listing with plenty of comments;

(2) to show a specific way of presenting software design, particularly the use of such diagrams as those in Figs. 3, 7, and 12;

(3) to bring out an important point: the separation of the architectural design (the creative part) from the programming of the design (the implementation part);

(4) to imply a significant point: the conceptual design is also implementable by hardware. Thus, the architecture of a design is independent of hardware and software.

This report not only presents the architectural design of an interpreter but also hopefully stimulates the reader to recognize the importance of architectural design of software that has been long neglected.

Yaohan Chu

# Table of Contents

Foreward

Abstract

1. Introduction

2. Description of the ALGOL Subset

   2.1 identifiers
   2.2 unsigned number
   2.3 variable
   2.4 label
   2.5 expressions
   2.6 assignment statement
   2.7 goto statement
   2.8 input-output statements
   2.9 declarations
   2.10 conditional statement
   2.11 program, block, and statements

3. Overview of the Interpreter

   3.1 scanner and syntax analysis
   3.2 error indications
   3.3 execution of the postfix string
   3.4 implementation considerations

4. Scanner

   4.1 remove blank
   4.2 recognize identifier or reserved word
   4.3 recognize constant
   4.4 recognize colon or assignment symbol
   4.5 recognize single character symbols
   4.6 procedure GC
   4.7 procedure SEARCH
   4.8 procedure STCK
   4.9 procedures ER1, ER2, ER3

5. Syntax Analyzer

   5.1 Floyd-Evans productions
   5.2 the analyzer described in the Floyd-Evans Productions
   5.3 symbol table
   5.4 table routines

6. Executor

6.1 BEGIN and BEG routines
6.2 END and ENDE routines
6.3 I routine
6.4 C routine
6.5 READ and WRITE routines
6.6 ↑, *, /, +, and - routines
6.7 = and ≠ routines
6.8 := routine
6.9 L routine
6.10 NEG routine
6.11 IF routine
6.12 THEN routine
6.13 TLS routine
6.14 ELSE routine
6.15 GOTO routine
6.16 HALT routine
6.17 FIND procedure
6.18 HASH procedure

7. Interpretations of Sample Programs

7.1 Program 1
7.2 Program 2

8. Acknowledgement

9. References

Appendix A, BNF Description of an Algol Subset

# Architectural Design of an Algol Interpreter

Claude K. Jackson

## 1. Introduction

This report describes the design of an interpreter for a subset of Algol using a syntax'directed technique. Floyd-Evans productions are chosen to describe the syntax analyzer, since they give a formal description of the process with clarity but without dependence on a particular machine or language. The design is given in a general way as it also serves to illustrate the use of the technique.

Besides this introduction, there are six sections. The first section describes the language to be interpreted. The next is an overview which indicates the four major elements of the interpreter together with discussions on their interrelationships and common terms. Then, one section each is devoted to the scanner, analyzer and executor, giving the configurations and the flowcharts of the design. The last section presents interpretations of two sample programs, showing exactly how the interpreter works.

## 2. Description of the Subset of Algol

In order to demonstrate the design of an interpreter, a subset of Algol is chosen. The syntax of this subset is described in Backus Normal Form and shown in Appendix A. As described, the syntax permits integers and declarations statements, arithmetic expressions and assignment statements, operator precedence grammar, boolean expressions and conditional statements, labels and GOTO statements, input and output statements, compound statements and block structure. It does not include the FOR statement, arrays, switches, procedures, and variables other than integers. The syntax and the semantics of the chosen subset follows closely to that in the revised Algol report (6). Symbolic names for the non-terminals in Appendix A are listed in Table 1.

### 2.1 Identifiers

<I>::= A|...|Z|<I>{A|...|Z}|<I>{0|...|9}

An identifier is therefore any sequence of letters or digits which begins with a letter. The interpreter as written will accept an identifier of any length, but it only recognizes and uses the first 12 characters of the identifier. An identifier may be used as either a label or a variable.

Examples:

A10 CONTINUE

BLACK1 T

### 2.2 Unsigned Number

<UN>::= 0|...|9|<UN>{0|...|9}

Table 1 Symbolic names for the non-terminals

| Nonterminals | Symbolic Names |
|---|---|
| arithmetic expression | AE |
| assignment statement | AS |
| basic statement | BS |
| block | B |
| boolean expression | BE |
| compound statement | CPS |
| compound tail | CT |
| conditional statement | CS |
| declaration | D |
| factor | F |
| goto statement | GTS |
| identifier | I |
| label | L |
| primary | P |
| program | PR |
| read statement | RS |
| statement | S |
| term | T |
| type list | TL |
| unconditional statement | US |
| unsigned number | UN |
| variable | V |
| write statement | WS |

An integer is the only type of number accepted by the interpreter. Floating point numbers are not allowed. An integer is only allowed to be 11 digits long.

Examples:

145    2378910

## 2.3 Variable

<V>::=<I>

A variable is an identifier which represents a value. This value may be changed during the execution of the program. The value is assumed to be integer and the variable must be declared before its use in the program.

Examples:

LET    VI

## 2.4 Label

<L>::=<I>

A label is not formally declared as a variable is, but is declared by its use in the program preceding a colon.

Examples:

L1    HERE

## 2.5 Expressions

An expression is a rule for computing a value. That value may be either a number in the case of arithmetic expressions, or TRUE or FALSE in the case of boolean expressions.

### 2.5.1 Arithmetic expressions

<P>::=<UN>|<V>|(<AE>)

<F>::=<P>|<F>↑<P>

<T>::=<F>|<T>{*|/}<F>

<AE>::=<T>|{+|-}<T>|<AE>{+|-}<T>

The above rules give the possible forms of arithmetic expressions. They also describe the order in which a value is to be computed since they describe the precedence of operators. Note that expressions in parentheses are to be evaluated before they are combined. The precedence of the operators as determined by the above rules is:

first ↑

second *,/

third +,-

Operators of the same precedence are evaluated from left to right.

Examples:

| | | | | |
|---|---|---|---|---|
| Primaries: | 85 | LOB | (A+B/2) | (-B) |
| Factors: | LS | TR↑6 | (A*B)↑K | |
| Terms: | M | M*L | M↑A | (-A)/C↑K |
| Arith. exp.: | N | N↑D | A+B-C | (A/B-T) |

### 2.5.2 Boolean Expressions

<BE>::=<AE>{=|≠}<AE>

The values of each of the arithmetic expressions are computed and then compared. If the comparison shows that the values are related in the same way as the logical operator (= or ≠) in the expression then the value of the boolean expression is true. If they are not related in the same way then the value of the boolean expression is false.

Examples:

N * T = B    N ≠ C

## 2.6 Assignment Statement

<AS>::=<V>:=<AE>

The value of the arithmetic expression to the right of the assignment symbol is stored as the value of the variable on the left side of the assignment symbol.

Examples:

A:= C * D + C

C:= C + 1

## 2.7 GOTO Statement

<GTS>::= GOTO<L>

Statements of the program are normally executed in sequential order, but when a GOTO statement is encountered the next statement to be executed

is the one that has the indicated label instead of the following statement.

Examples:

GOTO L1

GOTO CONTINUE

## 2.8 Input-Output Statements

<RS>::= READ(<V>)

<WS>::= WRITE(<V>)

A read statement causes a number to be read off a card and then stored as the value of the variable enclosed in parenthesis. A write statement causes the integer value of the variable enclosed in parentheses to be obtained and then written out as the next line of output.

Examples:

READ(AB)

WRITE(CD)

## 2.9 Declarations

<TL>::=<V>|<V>,<TL>

<D>::= INTEGER<TL>

Any variable used in a block must be declared at the beginning of the block. The declaration holds only for that block as in Algol and the variable is not defined outside of the block. Variables may be redeclared as in Algol. The redeclaration causes the variable to be in effect a different variable from the variable of the same name declared in the outer block.

The variable of the same name declared in the outer block is not defined for this block in which it has been redeclared. At the end of the block when the redeclared variable becomes undefined, the old variable becomes defined again with the value it had when the block was entered. A variable which is declared in a block is given the value zero when that block is entered.

Examples:

INTEGER A

INTEGER LET,NUM,C

## 2.10 Conditional Statement

<CS>::= IF<BE>THEN<US>{ |ELSE<US>}|<L>:<CS>

The boolean expression is evaluated. If its value is true then the unconditional statement after the THEN is executed. The unconditional statement after the ELSE, if it exists, is ignored in this case. If the value of the boolean expression is false then the unconditional statement after the THEN is executed. The unconditional statement after the ELSE, if it exists, is ignored in this case. If the value of the boolean expression is false then the unconditional statement after the THEN is not executed and the unconditional statement after the ELSE is executed. If there is no ELSE statement when the boolean expression is false then the next statement in the program is executed. A conditional statement may have a label.

Examples:

IF K1 = C THEN L:= T + 1 ELSE L:=1

IF K3 ≠ C1 THEN K = K + 1

2.11 Program, Block, and Statements

<BS>::=<AS>|<GTS>|<RS>|<WS>|<L>:<BS>

<US>::=<BS>|<CPS>|<B>

<S>::=<US>|<CS>

<CT>::=<S>|<CT>;<S>

<CPS>::= BEGIN<CT>END|<L>:<CPS>

<B>::= BEGIN<D>;<CT>END|<L>:<B>

<PR>::=<B>|<CPS>

An assignment statment, a GOTO statement, a READ statement, or a WRITE statement may have any number of labels. An unconditional statement is any of the above statements, a block or a compound statement. A statement is an unconditional statement or a conditional statement. A compound statement is any number of statements surrounded by a BEGIN and END symbol. A block has the same form as a compound statement except it must have a declaration before the list of statements. Blocks and compound statements may have labels also.

As mentioned before a variable declared in a block is local to that block. A block may have blocks nested within it and variables can be redeclared in a nested block. No matter what value the redeclared variable takes on in the nested block, when that block is left the value of the variable returns to the value it had before being redeclared, the so-called global value. A label is declared by its use in a block and is local to the

innermost block it is in. It is not possible therefore to transfer from outside a block to a statement in that block.

A program consists of either a block or a compound statement.

Examples:

```
Basic statement:        A:= B+C       GOTO L1     K:A:=1

Compound statement:     BEGIN A:=B*C; WRITE(A)    END

BLOCK:                  BEGIN INTEGER A,B,C; B:=1; C:=1; A:=B*C; WRITE(A) END
```

Figs. 21 and 26 give some additional examples.

## 3. Overview of the Interpreter

As shown in Fig. 1, the interpeter consists of four major elements: the scanner, the syntax analyzer, the executor, and the symbol table. The scanner converts the input program into a string of symbols in the internal code where the code is shown in Table 2. This input string of symbols in the internal code is then processed by the syntax analyzer. The analyzer outputs a postfix string now also in the internal code and generates a symbol table. The postfix string in conjunction with the symbol table is executed by the executor to produce the desired results.

Fig. 2 shows the flow chart for the interpreter. It consists of six blocks, each of which represents a process. These processes except the initialization process are described below.

### 3.1 Scanning and Syntax Analysis

The syntax analyzer is described by the Floyd-Evans productions. The actual set of Floyd-Evans productions which describes the analysis will be described in detail later. These productions perform the syntax analysis part of the interpretation. The scanner is merely a procedure (or a subroutine) called by the analyzer whenever the next symbol of the input stream is needed.

It is important to note the difference between a symbol and a character of the input program. A character is the contents of a single column of a card of the input program. It may be a letter, digit, or any other punctuation such as *. A symbol may however be made up of any number of characters. The analyzer only works with symbols and it is the sole function of the scanner to obtain the next symbol of the input program for

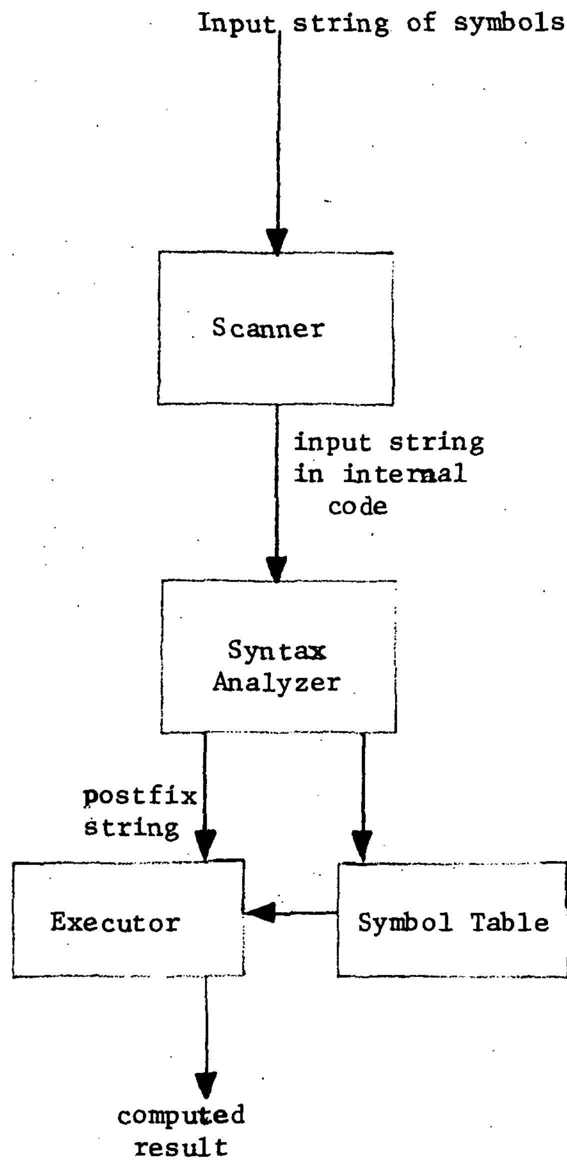



Fig. 1. Configuration of the Interpreter

Table 2 Internal code for the symbols

| SYMBOL | CODE |
|---|---|
| $\uparrow$ | 1 |
| * | 2 |
| / | 3 |
| + | 4 |
| - | 5 |
| = | 6 |
| $\neq$ | 7 |
| $\mapsto$ | 8 |
| , | 9 |
| ; | 10 |
| : | 11 |
| ( | 12 |
| ) | 13 |
| := | 14 |
| I | 15 |
| C | 16 |
| BEGIN | 17 |
| END | 18 |
| INTEGER | 19 |
| READ | 20 |
| WRITE | 21 |
| GOTO | 22 |
| IF | 23 |
| THEN | 24 |
| ELSE | 25 |
| BEG | 26 |
| TLS | 27 |
| HALT | 28 |
| NEG | 29 |
| ENDE | 30 |
| L | 31 |

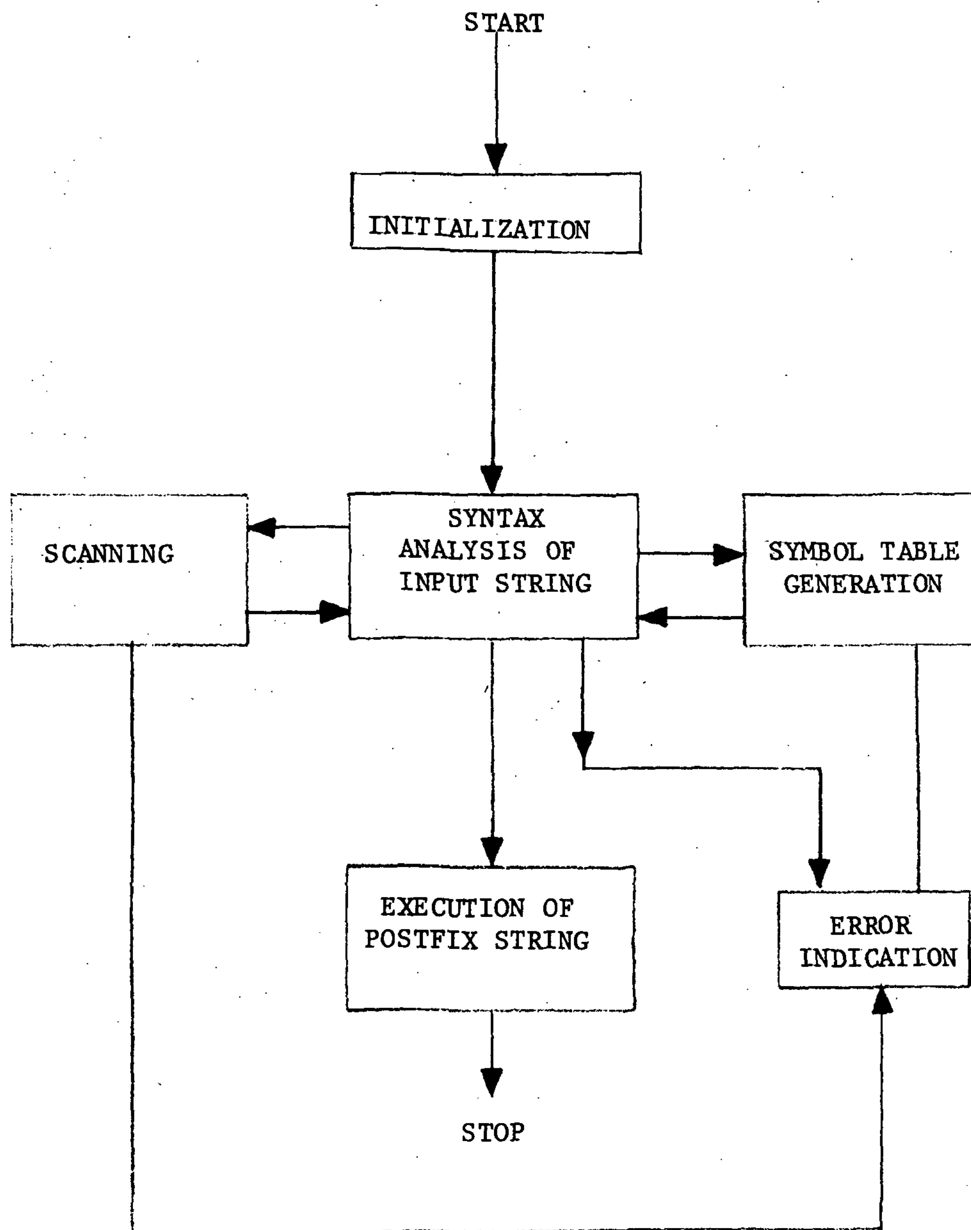



Fig. 2 Flow chart for the Interpreter

the analyzer.

The scanner goes through the input program character by character until the next symbol has been found. As a result, the analyzer can be designed to work with fixed length symbols and does not have to worry about the problems of blanks and lengths of variable names. A list of the symbols for the interpreter is given in Table 2. Next to each symbol is the internal code which is actually used in implementation. The symbols are retained in the description of the Floyd-Evans productions for purposes of readability. If the internal code were used they would be much less comprehensible, but with the symbols they give a very clear description. In short, the only function of the scanner is to pass the next symbol whenever it is needed and also to pass the identifier or constant that the symbol might represent. Since the analyzer assumes that the first symbol has already been obtained when they start, the scanner is called once before syntax analysis of the input string begins. The symbol table is also formed during syntax analysis by those procedures called table routines.

## 3.2 Error indications

Error routines may be called in during any part of the program. An error message is printed out and interpretation stops in most cases. In a few cases, there is some change made to try to solve the error, and interpretation continues, but for the most part the interpreter does not have any error recovery capability.

As mentioned above the result of syntax analysis is a postfix string of symbols or, in actual implementation, a postfix string of internal code. There is a parallel string to the above string which will hold semantics of the postfix string. It will hold the actual identifier which

an I symbol represents and the actual constant for a C symbol.

## 3.3 Execution of the Postfix String

When syntax analysis is done the entire postfix string will have been produced. The execution part of the interpreter then begins to operate using the above two strings produced by the syntax analyzer and also the symbol table that has been produced. The execution part processes the postfix string performing certain actions depending on what particular symbol in the postfix string is presently being processed. The details of this process are left for the description of the execution part.

## 3.4 Implementation

Algol is assumed to be used as implementation language. Although this study does not include the implementation, some remarks about the implementation are made below.

The interpreter uses procedures which function as Algol procedures. Variable names and arrays used by the interpreter are assumed to be all global and thus accessible to all parts of the program. By this means, one part of the interpreter may store something in a global variable for use by another part of the interpreter later. This is how the scanner passes internal code to the syntax analyzer and how the syntax analyzer passes the postfix string to the executioner. The Floyd-Evans productions and flowcharts are written so that the interpreter could be implemented extremely simply without complications in UNIVAC 1108 Algol. This implementation would not be particularly efficient since efficiency has been sacrificed for simplicity or clarity wherever possible. In particular 1108 Algol allows string arrays which hold 12 characters per array element. Since identifiers can have up to twelve

characters, the interpreter as designed uses string arrays with 12 characters per array element which is very inefficient but makes the description of the design simple. The design also allots a full array element for things such as flags which require only a single bit. By using a single array element for each quantity stored the design is much clearer than if two words or part of a word were used to store particular quantities. Since the design has been made in the simplest form it would be easy to change it or to actually implement the interpreter not only more efficiently but also in another language or on another machine.

The idea here is that a type of layered approach to a practical interpreter has been taken. The initial idea was to produce an interpreter. The first thing done was to decide a three part interpreter should be made. The form of each part as mentioned earlier was decided upon next. Then a detailed but flexible design of the interpreter has been made and described herein. This design can be implemented easily and tested in Algol. The final step or layer of the design is the "practical" implementation of the interpreter. This step requires working out certain details and making minor changes so that all the requirements of the particular implementation are met. An example is that if the interpeter were finally implemented in UNIVAC 1108 Fortran it would be necessary to store 12 characters in two words so some changes would have to be made in flowcharts and data structures but they would all be straightforward and require no major design changes. A major advantage of this approach is that the problems should be met and solved at the appropriate time during design. Debugging and design changes will hopefully be less random and more control will be maintained at all stages of design.

## 4. Scanner

The configuration of the scanner is shown in the block diagram of Fig. 3. There are five buffers and one stack. Input buffer INPUT with pointer C1 can contain 72 characters. Buffer CH is a single character buffer. Buffer N with pointer C can hold 12 characters, and buffer N1 also with pointer C can hold 11 decimal digits. Code buffer T stores the internal code of a symbol. Stack FE with pointer C2 is the place where the symbol produced by the scanner is put.

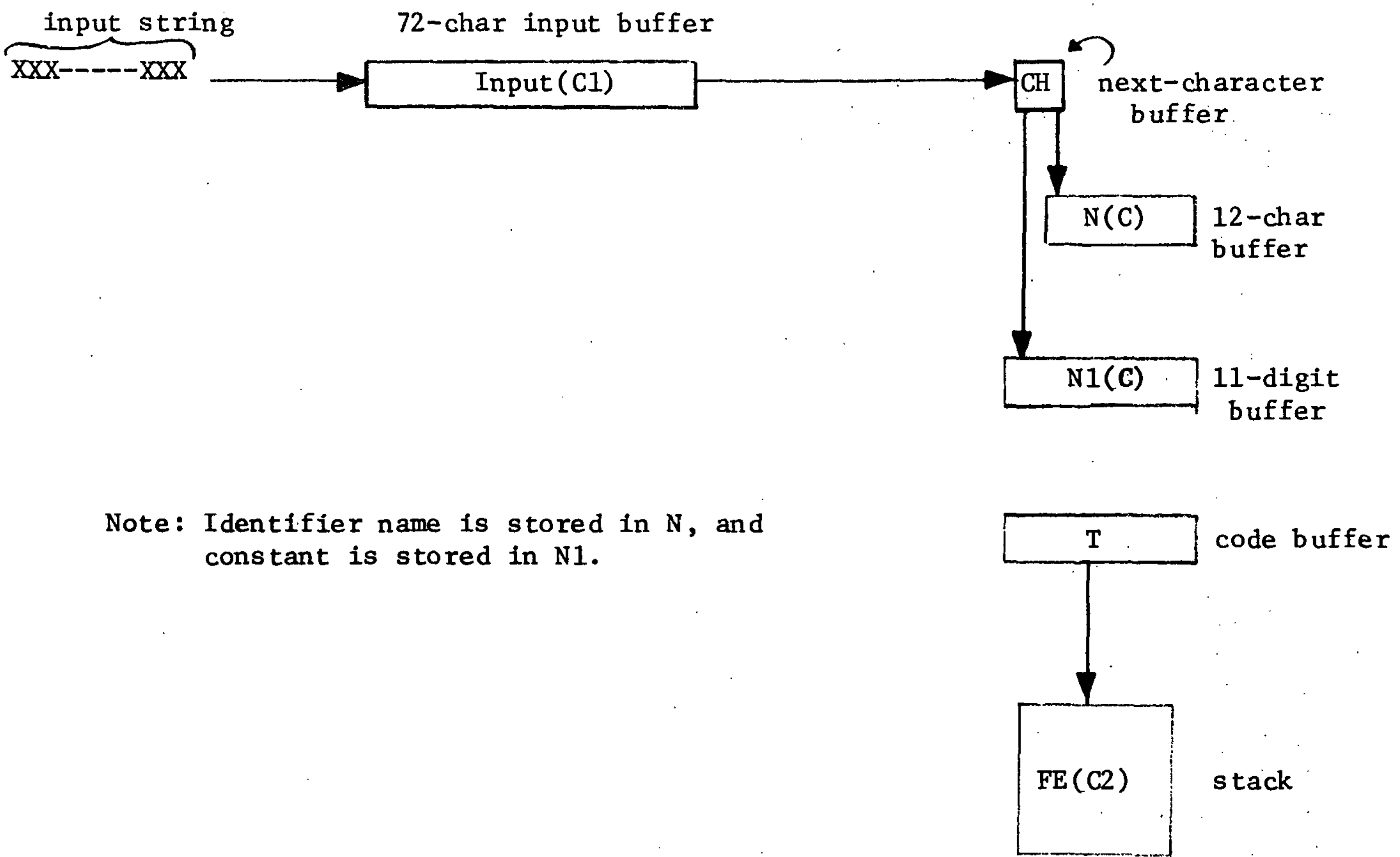
The scanner is called into action by the syntax analyzer whenever the analyzer needs the next symbol. When called, the scanner places the next symbol at the top of stack FE. As mentioned before, the symbol placed in the stack is not the original characters in the input string, but the internal code shown in Table 2.

The operation of the scanner is described in the flow chart of Fig. 4 and the terms in the flow chart are explained in Table 3. A procedure called GC is employed by the scanner to obtain the next character from the input string and place it in buffer CH.

Since the scanner always starts with the assumption that the next character is already in buffer CH procedure GC is called once during initialization of the interpretation to place the 1st character in CH. Thereafter, the scanner will leave the next character in buffer CH.

### 4.1 Remove Blank

The first part of the scanner checks for a blank in buffer CH and if found it calls procedure GC to put a new character in CH. This process removes any blank characters before a symbol. Blanks may not be used in iden-




Note: Identifier name is stored in N, and constant is stored in N1.

Fig. 3. Configuration of the scanner

Table 3, Designation of the Terms used in the Scanner

| Term | Designation |
|---|---|
| CH | a buffer for the next character in the input string |
| N | a buffer for storing a string of 12 characters |
| INPUT | a buffer for 72 characters |
| C | a pointer for N and N1 |
| C1 | a pointer for INPUT |
| C2 | a pointer for FE |
| FE | a stack for syntax analysis |
| SEARCH | a procedure to determine if a given identifier is a reserved word |
| N1 | a buffer for storing a string of digits |
| GC | a procedure to botain the next character in CH from the input string |
| STCK | a procedure to stack the internal code of the symbol in stack FE |
| T | a buffer for storing an internal code |
| ER1 | a procedure to indicate an illegal character has been encountered |
| ER2 | a procedure to indicate that an end-of-program marker is encountered and a complete program has yet to be processed |
| ER3 | a procedure to indicate a constant having more than 11 digits |

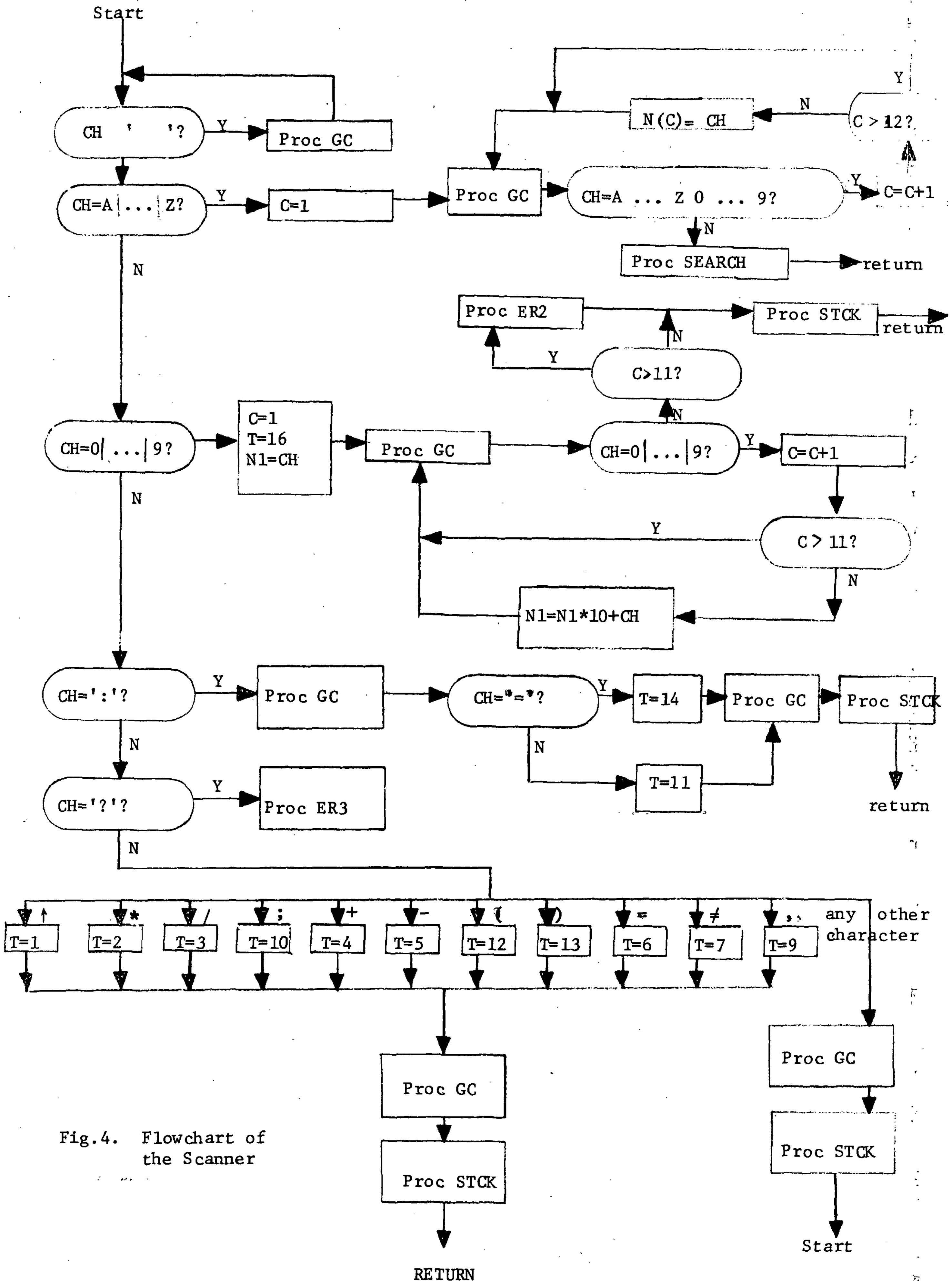



Fig.4. Flowchart of the Scanner

tifiers or constants so they mark the end of an identifier or constant. In any other place blanks are ignored. The scanner then proceeds to check what the first non-blank character is and enters the appropriate branch to finish processing the symbol.

4.2 <u>Recognize Identifier or Reserved Word</u>

If the first nonblank character is a letter then the symbol must be an identifier or reserved word. The first twelve characters of the identifier or reserved word are stored in the buffer N and the counter C is used to count how many characters the identifier or reserved word has. The scanner enters a loop to do this. Note that after the first character, the identifier may have any digit or letter and it terminates only when neither a digit or letter is the next character. The next character is thus in buffer CH when the symbol has been processed and the character is left there for the next call of the scanner. Only the first 12 characters found are saved in N. If the identifier is longer then the characters are merely ignored and not stored in N. When the entire identifier has been obtained, then the procedure SEARCH is called to check to see if the identifier is in fact a reserved word. The appropriate symbol (internal code) is placed on the stack FE of the analyzer. Either the symbol for the reserved word matched or the symbol for identifier if there is no match is placed on the stack. Stacking on FE is always done in two steps. The internal code for the symbol is stored in T and then just before the scanner is exited procedure STCK is called to put the value in T onto stack FE.

4.3 <u>Recognize Constant</u>

If the first nonblank character is a digit, then the scanner pro-

ceeds into another loop which processes a constant of up to 11 digits. The constant is stored in buffer N1 and is terminated when the first non-digit character is encountered. If the constant is over 11 digits long then the leftmost 11 digits are used and an error message is printed out by procedure ER2 before returning to the syntax analyzer. Again a character is left in buffer CH for the next call of the scanner. The internal code 16 is placed on stack FE.

### 4.4 Recognize Colon or Assignment Symbol

If a : is the first non-blank character then the symbol must be either := or just :. Therefore, the next character is obtained by procedure GC and checked to see if it is =. If it is = then the code 16 is placed on stack FE and the next character must be obtained by calling procedure GC so that the next character will be in buffer CH for the next call of the scanner. If the next character was not = then the internal code 11 is placed on stack FE and procedure GC is not called since the next character is already in buffer CH.

### 4.5 Recognize Single Character Symbols

The last section of the scanner checks for single character symbols. When a symbol is matched, then the appropriate internal code is stacked on stack FE. The next character must be placed in buffer CH by calling procedure GC and then the program returns to the syntax analyzer. If the single character is ? (a character placed at the end of all input programs), then the end of the input string has been reached without a complete program having been processed. This is an error condition so procedure ER3 is called to print out an error message and end the interpretation. If the single character

does not match any of the legal characters, then an error message is printed out by procedure ER1. The illegal character is then skipped by calling GC to get the next character and then returning to the start of the scanner.

### 4.6 Procedure GC

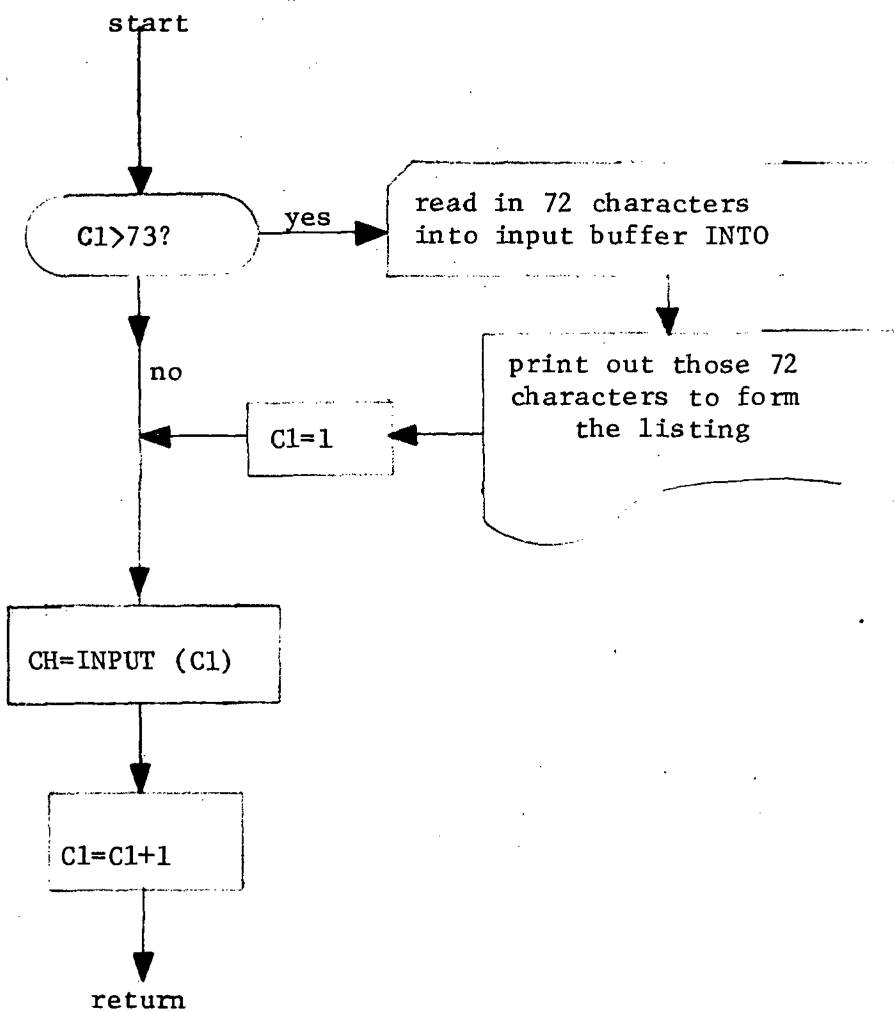
Fig. 5(a) shows the flow chart for procedure GC. As mentioned earlier, procedure GC obtains the next character of the input string. When the interpretation of a new program begins, the pointer C1 is initialized to 73. Whenever procedure GC is called and the pointer C1 is 73, then 72 characters are read in off the next card and placed in the string INPUT. Pointer C1 is then initialized to 1 to point to the first character. Note the 72 characters read in are printed out to form a listing of the program.

After reading in 72 new characters or immediately if pointer C1 is less than 73, the character pointed to by pointer C1 is stored in buffer CH, pointer C1 is incremented, and then the procedure ends.

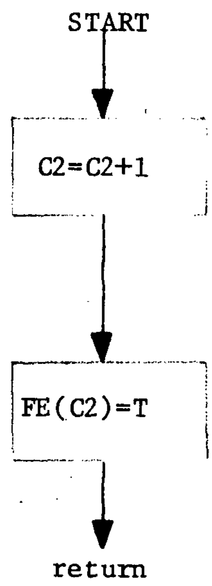
### 4.7 Procedure Search

The flow chart for procedure SEARCH is shown in Fig. 6. Procedure SEARCH tests an identifier to see if it is a reserved word. If it has more than 7 characters then it cannot be a reserved word and we can immediately process it as an identifier. This consists of placing the internal code for identifier, 15, into buffer T, stacking it on FE, and returning. The actual identifier name is saved in buffer N.

According to the number of characters of the identifier, it is tested character by character for the possible reserved words. After a number of tests it will be known either that the identifier cannot be a reserved word in which case it can be processed as an identifier as described above,




(a)Procedure GC




(b)Procedure STCK

Fig. 5. Flowcharts for Procedures GC and STCK

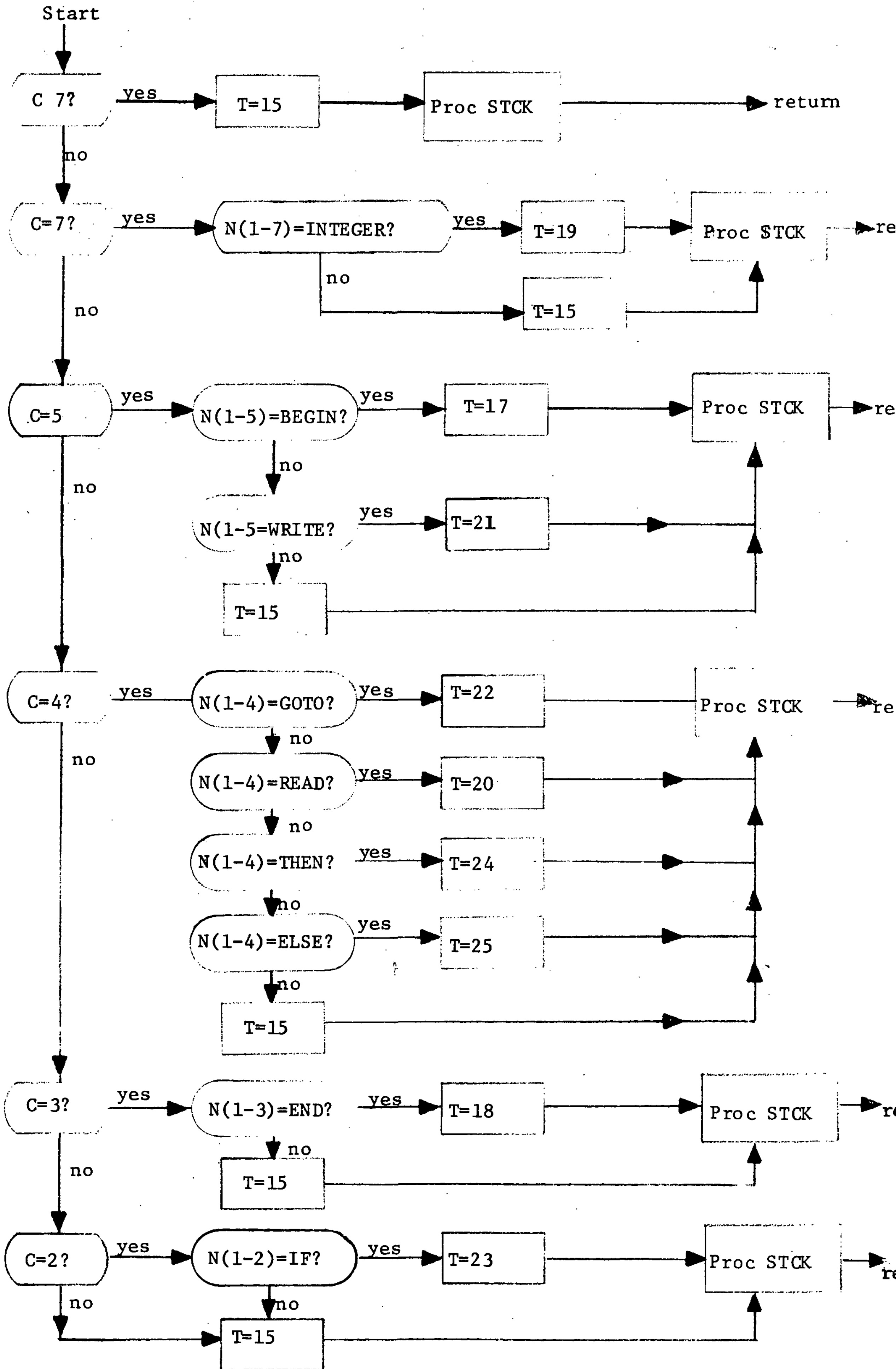



Fig. 6 Flow chart for the Procedure SEARCH

or it is a reserved word. In the latter case, it will have matched at all tests for characters of a particular reserved word. The appropriate internal code will be placed on stack FE and the procedure ends. After processing by this procedure reserved words have been changed to internal code and are handled in coded form hereafter. They are never entered in the symbol table.

### 4.8 Procedure STCK

Fig. 5(b) shows the flow chart of procedure STCK. Procedure STCK handles the stacking of the internal code of a symbol which is always first stored in buffer T. There are two operations. First, the stack pointer C2 must be incremented since it always points to the top element of stack FE, and second, the code is then placed on the top of stack FE.

### 4.9 Procedures ER1, ER2, ER3

The three procedures ER1, ER2, and ER3 print-out error messages. ER2 also caused an end of interpretation. The flow charts for these procedures are not provided.

## 5. Syntax Analyzer

The syntax analyzer analyzes the input string, produces the postfix string, and constructs the symbol table. The configuration of the syntax analyzer is shown in Fig. 7. As shown, there are five buffers, one stack, and a symbol table. Buffers N, N1, and T and stack FE have been introduced when the scanner was described. Buffers PS store the postfix string of symbols, while buffer PS1 stores the postfix string of constant values and identifier names. Thus, symbols I and C in buffer PS represent identifiers and constant respectively, while their names and values are located in the corresponding positions in buffer PS1. Table SYM is the symbol table of the input program. The details of the process of syntax analysis are now described below.

### 5.1 Floyd-Evans Productions

The syntax analysis is described rather precisely and concisely by the Floyd-Evans productions. The particular form of the productions adapted in this report is the same as the form used by Evans (3) but with the addition of table routines for the purpose of constructing the symbol table. Since the form of the productions varies with their use by each new author, one must make sure that the particular form of the productions is understood. The key element here as in most types of syntactic analysis is the stack. The stack starts with two occurrences of the special symbol $\vdash$ (internal code 8) as its top element. The reason for doing so is given later.

The productions can be considered as a programming language since they can easily be interpreted themselves. Each production can be considered an instruction to be executed. It is extremely simple and straightforward to

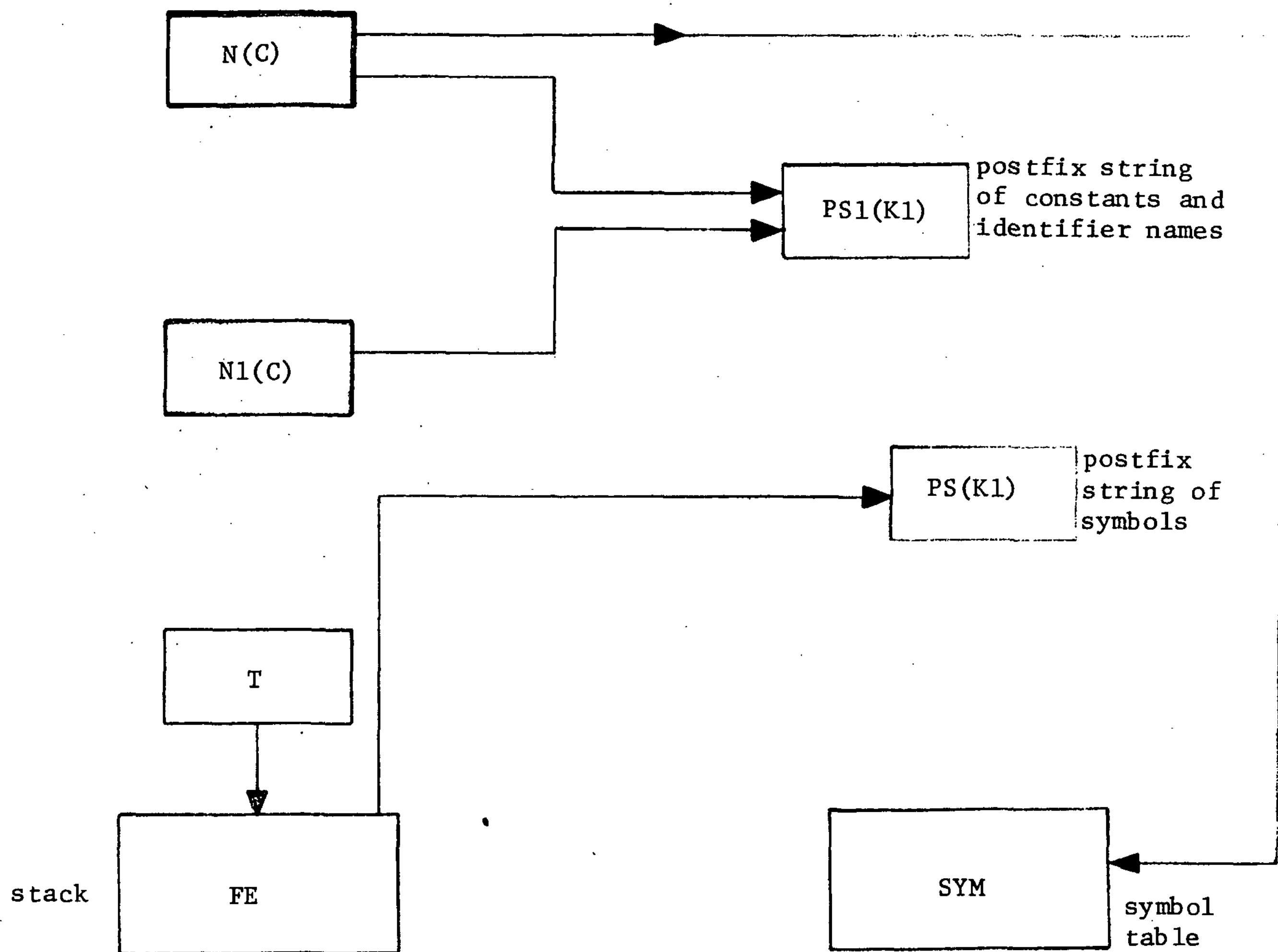



Fig. 7. Configuration of the syntax analyzer

implement a syntax analyzer described using these productions and as a result no flow charts are given to describe the syntax analysis but only the table of Floyd-Evans productions is given. As will be seen these productions provide a machine and language independent description of syntax analysis.

Each production has six columns. The first column may contain a label which is used for branching as in any programming language. The second column contains symbols that are to be compared to the top symbols of the stack. Once again we remark that we will speak of symbols and use the symbols shown in Table 2 but in actual implementation the internal code will be used instead. If there is a match between the symbols in the second column and the top symbols of the stack then the actions indicated in the remaining columns are performed. If there is no match then the next production is performed next.

If there has been a match and there is a → in the third column then the symbols that were matched in the stack are replaced by the symbols to the right of the →. If there is no → in the third column then the stack is left unchanged. A special symbol <SG> means that any symbol in that position of the stack is matched.

Since the purpose of the syntax analysis is to produce a postfix string the fourth column specifies symbols that are to be placed in the postfix string if there was a match. If an I or C symbol is to be placed in the postfix string then the corresponding identifier (saved in N) or constant (saved in N1) is also placed in the parallel postfix semantic string.

Sometimes in column four there occurs the word COMPILE or COMPILE followed by a symbol. In these cases these symbols are not outputted. Instead, in the first case the precedence of the top symbol of the stack is compared to the precedence of the second symbol on the stack and if the precedence

of the second symbol is greater than or equal to the precedence of the top symbol then the second symbol is removed from the stack and placed in the postfix string. This process is repeated until the precedence of the top stack element becomes greater than the precedence of the second stack element.

If a symbol follows the word COMPILE, then instead of comparing the top stack element to the second stack element, the precedence of the symbol after word COMPILE is compared with the precedence of the second stack element. The word COMPILE is used in order to implement the technique of operator precedence in parts of the syntax analysis.

The fourth column may contain a call to a table routine. These routines are procedures which perform some part of the construction of the symbol table depending on which symbol has just been matched on the stack. If a BEGIN followed by an INTEGER, an END, an I or a C symbol is matched then the appropriate table routine is called. A thorough description of the symbol table and the table routines along with flowcharts are given later in this section of the report.

If the last column has a * in it then the scanner is to be called to obtain the next symbol of the input program and place it on top of the stack. The scanner takes care of recognizing identifiers and changing them to the symbol I before placing them on the stack. Also in the last column is the label of the next production to be executed or the instruction HALT to stop execution of the productions.

## 5.2 The Analyzer Described by Floyd-Evans Productions

The algorithm for the syntax analyzer is described by the Floyd-Evans productions shown in Table 4, where the following metacharacters are adopted:

<RL> is $= | \neq$

and <OP> is $+ | - | * | / | \uparrow$ .

The construction of the table is based upon checking for various syntactical constructs as defined in Appendix A until a legal one is found in the input program. The checking is a so-called left-to-right bottom-up analysis and continues until a complete program has been recognized. At each testing of neighboring symbols, there is only a finite number of syntactical constructs to check. If the symbols on the stack do not match one of the tests, then the input program is in error. In such a case, the last test checks <SG> which means any symbol and causes an error routine to be executed. If a match is found on the stack, the appropriate change in the stack is made and the analyzer proceeds to check the next set of possible syntactical constructs.

Two markers are needed: the end-of-stack marker and the beginning-of-statement marker. The former is to indicate that the stack is empty and the latter is to mark where the statement currently being analyzed began. The symbol $\mapsto$ (internal code 8) is used for both markers.

### 5.2.1 Labels S0, L1, D1, DC1, DC2

In the production labeled S0 a check for the outermost block which must start with symbol BEGIN is made as shown in Table 4. There might be a

Table 4, Syntax Analyzer described in Floyd-Evans Productions

| Label | Stack Before | Stack After | Output to Post-fix string PS | Table Routine | Next Production |
|---|---|---|---|---|---|
| S0 | BEGIN | →1→1→ | | EXEC1 | *D1 |
| | I | → | L | EXEC5 | *L1 |
| | <SG> | | | | ER1 |
| L1 | : | → | | | *S0 |
| | <SG> | | | | ER2 |
| D1 | 1→ INTEGER | →BEGIN 1→INTEGER | BEGIN | EXEC2 | *DC1 |
| | 1→ <SG> | →BEG1→<SG> | BEG | | S1 |
| DC1 | INTEGER I | →INTEGER | | EXEC4 | *DC2 |
| | <SG> | | | | ER3 |
| DC2 | , | → | | | *DC1 |
| | INTEGER; | → | | | *S1 |
| | <SG> | | | | ER4 |
| S1 | IF | | | | *EX1 |
| S1B | I | → | | | *S2 |
| | GOTO | → | | | *G1 |
| | READ | → | | | *R1 |
| | WRITE | → | | | *W1 |
| | BEGIN | → | | | *D1 |
| EX1 | I | → | I | | *EX3 |
| | C | → | C | | *EX3 |
| | ( | | | | *EX4 |
| | + | → | | | *EX4 |
| | - | →NEG | | | *EX4 |
| | <SG> | | | | ER5 |

| Label | Stack Before | Stack After | Output to Post-fix Strings PS | Table Routines | Next Production |
|---|---|---|---|---|---|
| E1 | 1→ IF THEN | →THEN 1→ | IF | | *S1B |
| | <SG> | | | | ER6 |
| E2 | THEN 1→ ELSE | → ELSE 1→ | THEN | | *S1B |
| | <SG> | | | | ER22 |
| E3 | THEN 1→ END | → 1→ END | TLS | | E3 |
| | ELSE 1→ END | → 1→ END | ELSE | | E3 |
| | BEGIN1→END | → 1→ | END | EXEC3 | E4 |
| | BEG 1→ END | → 1→ | ENDE | | E4 |
| | <SG> | | | | ER7 |
| E4 | 1-> 1-> | | HALT | | HALT |
| | <SG> | | | | *E5 |
| E5 | END | | | | E3 |
| | ; | | | | E6 |
| | ELSE | | | | E2 |
| | <SG> | | | | ER24 |
| S2 | : | → | L | EXEC6 | *S1 |
| | := | | I | | *EX4 |
| | <SG> | | | | ER8 |
| EX1 | I | → | I | | *EX2 |
| | C | → | C | | *EX2 |
| | ( | →( | | | *EX1 |
| | + | → | | | *EX1 |
| | - | →NEG | | | *EX1 |
| | <SG> | | | | ER9 |
| EX2 | <OP> | | COMPILE | | *EX1 |
| | <RL> | | COMPILE | | *EX4 |
| | ) | | COMPILE | | P1 |
| | <SG> | | | | ER10 |

| Label | Stack Before | Stack After | Output to Postfix String PS | Table Routines | Next Production |
|---|---|---|---|---|---|
| EX3 | <OP> | | COMPILE | | *EX4 |
| | ) | | COMPILE | | P2 |
| | ; | | COMPILE ←1 | | E6 |
| | THEN | | COMPILE ←1 | | E1 |
| | ELSE | | COMPILE ←1 | | E2 |
| | END | | COMPILE ←1 | | E3 |
| | <SG> | | | | ER11 |
| E6 | THEN 1→ ; | → 1→ | TLS | | *S1 |
| | ELSE 1→ ; | → 1→ | ELSE | | *S1 |
| | 1→ ; | 1→ | | | *S1 |
| | <SG> | | | | ER12 |
| P1 | ( ) | → | | | *EX2 |
| | <SG> | | | | ER13 |
| P2 | ( ) | → | | | *EX3 |
| | <SG> | | | | ER14 |
| G1 | I | → | I GOTO | | *E5 |
| | <SG> | | | | ER15 |
| R1 | ( | → | | | *R3 |
| | <SG> | | | | ER16 |
| R2 | I | → | I | | *R3 |
| | <SG> | | | | ER17 |
| R3 | ) | → | READ | | *E5 |
| | <SG> | | | | ER18 |
| W1 | ( | → | | | *W2 |
| | <SG> | | | | ER19 |
| W2 | I | → | I | | *W3 |
| | <SG> | | | | ER20 |
| W3 | ) | → | WRITE | | *E5 |
| | <SG> | | | | ER21 |

Table 5 Precedence Table

| Precedence | Symbols | Remark |
|---|---|---|
| 1 | $\uparrow$ | HIGHEST |
| 2 | *, / | |
| 3 | +, -, NEG | |
| 4 | ) | |
| 5 | =, $\neq$ | |
| 6 | := | |
| 7 | ELSE | |
| 8 | $\longleftarrow\!\!\!\dashv$ | |
| 9 | IF | |
| 10 | THEN | |
| 11 | ( | |
| 12 | $\longmapsto$ | LOWEST |

label, so that possibility is also checked. If neither symbol BEGIN nor symbol I is matched, then there is an error and error routine ER1 is executed next. If symbol I is found, then the production labeled L1 is the next production to be executed. L1 will check to see if I is indeed a label which means I must be followed by a :. If there is no : then there is an error. If it is a label, an L symbol (code 31) is placed in the postfix string PS. Table routine EXEC6 is called to put the label in the symbol table. The production labeled SO is then executed again to check for the outermost block. When the BEGIN symbol of the outermost block is encountered the symbols ⊢→⊢→ are placed on top of the stack FE of the Floyd-Evans productions.

Table routine EXEC1 is called to initialize variables needed to form block heads. Block heads are formed for each block and are described in the section on the symbol table and table routines. The production labeled D1 is the next production and it processes the beginning of blocks by checking for declarations. Two different types of BEGIN symbols are placed in the postfix string PS depending on whether or not there is a declaration. If there is no declaration then a compound statement is being encountered and the symbol BEG is placed in the postfix string. If there is a declaration this is a block and a BEGIN symbol is placed in the postfix string. Table routine EXEC2 is called to form a block head for this block. The production starting at DC1 and DC2 process the declaration. Note that a variable which is declared is not placed in the postfix string but table routine EXEC4 is called to place the variable in the symbol table. The productions starting at DC2 checks for the end of the declaration and no trace of the declaration is placed in the postfix string PS.

### 5.2.2 Labels S1, S2

After the beginning of a block or compound statement has been processed there must be a statement so production S1 is next. It checks for all possible forms of a statement. No blank statement is allowed so the productions starting at S1 check all the possible starting symbols for a statement. If an I symbol starts a statement it may be a label or the beginning of an assignment statement. The program goes to the productions starting at S2 which check for these possibilities. If the I symbol is followed by a : symbol then this is a label so an L symbol is placed in the postfix string PS. Table routine EXEC7 is called to place the label in the symbol table and then the program returns to production S1 to check for a statement.

If the I is followed by a := symbol then this statement must be an assignment statement. The I symbol is placed in the postfix string immediately. It is not left on the stack FE as operators are sometimes. Operators are only placed in the postfix string at the appropriate time to produce proper postfix notation. The rest of the assignment statement must be an arithmetic expression so the program will proceed to production EX4 which processes arithmetic expressions.

If an IF symbol, GOTO symbol, READ symbol or WRITE symbol is encountered at the beginning of a statement then the program proceeds to appropriate productions to check to see if the statement has the correct form. A statement may also be a block or compound statement so it may start with a BEGIN in which case the program proceeds to production D1 to process the beginning of the block or compound statement.

### 5.2.3 Labels G1, R1, W1

The processing of the GOTO, READ, and WRITE symbols by the productions

starting at G1, R1, and W1 respectively is very straightforward since there is a standard form with no possible variation. Only one I symbol is permitted in each of these statements. The I is outputted into the postfix string immediately and if the whole statement is correct the appropraite symbol (GOTO, READ, or WRITE) is placed after the I. In all three cases the next production to be executed is E5 which processes the end of statements.

### 5.2.4 Conditional Statements and Assignment Statements

A conditional statement requires substantial checking. Immediately after the IF symbol must come a boolean expression which is two arithmetic expressions with a relational operator (= or ≠) between them. Upon recognizing an IF the program proceeds to production EX1 which processes an arithmetic expression using operator precedence using the precedences given in Table 5. The program goes back and forth between the productions starting at EX1 and EX2 processing operators and operands. Operands (I or C) are placed in the postfix string immediately while operators are placed in the postfix string only when operation COMPILE indicates that they should be. Unary operators and parentheses are also processed. When unary minus is encountered it is changed to the symbol NEG (code 29) so that when it is eventually placed in the postfix string there will be no ambiguity between unary and binary minus.

When a ) symbol is encountered then the operation COMPILE is executed. Next the program proceeds to production P1 to check if there is a matching parenthesis. If there is a matching parenthesis then all operators up to the matching ( will have been placed in the postfix string by the operation COMPILE. If there is not a ( symbol then there is no matching parenthesis so there will be no match at production P1. The production after P1

will match and the error routine ER13 will be executed. If there was a match the program returns to production EX2 to continue processing.

When the relational operator is found then there must be another arithmetic expression so the program goes to production EX4 which as mentioned earlier processes arithmetic expressions. Thus assignment statements and conditional statement both use the productions at EX3 and EX4. As for the productions at EX1 and EX2 the program goes back and forth between EX3 and EX4 processing operands and operators. The program stops this processing, not when it finds a relational operator as before, but when it encounters a ; symbol, a THEN symbol, an ELSE symbol, or an END symbol. After encountering one of these symbols the program performs the COMPILE ←1 operation which removes all operators from the stack. The program then goes to the appropriate productions depending on which symbol was encountered. All of the above described productions have been constructed so that only proper constructions of assignment and conditional statements are accepted.

The productions at EX1, EX2, EX3, and EX4 are constructed and precedences given to symbols so that illegal boolean expressions are not accepted. This is an important consideration since operator precedence is being used to speed analysis and also since certain productions (EX3 and EX4) are used to recognize two different constructs. There are certain errors which it was necessary to be careful not to accept. Boolean expressions without a relational operator, arithmetic expressions with relation operators, and parentheses which do not match or surround a relational operator were some of these errors. The first of these errors is detected by checking specifically for a relational operator at the proper place in the expression. If none occurs then eventually a match is made which causes an error routine to be executed. The second error is caught because a relational operator will not match anything in the productions at EX4 and will thus cause an error routine to be exe-

cuted.

The problems with parentheses were detected by making the precedences of all symbols such that only proper constructs would be unstacked properly. A ) symbol to the right of a relational operator without a matching ( or with a ( on the other side of the relational operator causes everything only up to the relational operator to be unstacked. Therefore there is no match at P2 which indicates the error. A ( without a ) will never be unstacked because of its low precedence so it will cause no match at some point and result in an error indication.

If in the productions at EX3 a THEN symbol is encountered then the program proceeds to production E1. A THEN must have been proceeded by an IF symbol so if a THEN follows an assignment statement there will be no match. Only a proper conditional statement will be accepted. At this point the IF symbol is placed in the postfix string to maintain postfix notation. The program next proceeds to production S1B to look for an unconditional statement. Note that the program goes to production S1B and not production S1. Therefore, if a conditional statement follows there will be no match and the error will be detected.

If an ELSE symbol is encountered in the productions at EX3 after an assignment statement or in the productions at E5 after any other legal statement then the program proceeds to production E2 to check if the ELSE is preceeded on the stack by the symbols THEN 1→. If it is, then the proper form of a conditional statement has been followed so far. The THEN will only be in the stack if earlier an IF was processed properly. The THEN symbol is placed in the postfix string and the program proceeds to S1B again to look for an unconditional statement.

If in the productions at EX3 an END symbol or a ; symbol is encountered then the program proceeds to production E3 or production E6 depending on the symbol to check if a statement has just ended properly. All operators are unstacked before going to productions at E3 or E6 by using the operation COMPILE $\leftarrow$1. If an END or ; is encountered after a boolean expression the IF symbol still on the stack will cause no matches to occur in the productions at E3 or E6 and thus result in the error being detected.

Before discussing the productions at E3 and E6 it is desirable to mention the productions at E5 which is where the program goes after accepting a READ, WRITE, or GOTO statement. After accepting one of these statements there must follow an ELSE, if a conditional statement is being processed, a ; which means another statement follows, or an END which means the end of a block or compound statement has been reached. These are the only possible symbols which can follow a statement.

If a ; symbol was encountered then it either ended a conditional statement which means a TLS or ELSE must be placed in the postfix string after the statement just processed or the ; merely ended an unconditional statement. All three cases are checked for in the productions at E6 and in each case there must be another statement after the semicolon so the next production to be executed is at S1. Note the ; is deleted immediately upon recognition and is never placed in the postfix string. Since only one unconditional statement may follow a THEN or ELSE there is no ambiguity as to whether or not the ; ends the conditional statement. According to the grammar the ; must end the conditional statement.

In the productions at E3 an END following a statement is processed. The END may end a conditional statement by ending a block or compound statement in which the conditional statement is the last statement. The THEN or ELSE must be outputted into the postfix string and then the program proceeds

to productions at E3 to process the end of a block or compound statement. Here and after the ; is encountered the symbol TLS is outputted into the postfix string instead of THEN so that during execution it is known that no ELSE statement follows. The next possibility is that the END ends a block or compound statement which is appropriately processed. Blocks and compound statements are unconditional statements and are thus reduced to a $\vdash\!\!\!\rightarrow$ symbol in the stack. The program proceeds to productions at E4 which check for $\vdash\!\!\!\rightarrow\vdash\!\!\!\rightarrow$ on the top of the stack which means the end of the original block has been processed and syntax analysis is done. If this is not the case then another symbol must be obtained from the scanner and the program proceeds to productions at E5 to see how this statement (the block or compound statement) just processed has ended.

### 5.2.5 Error Routines

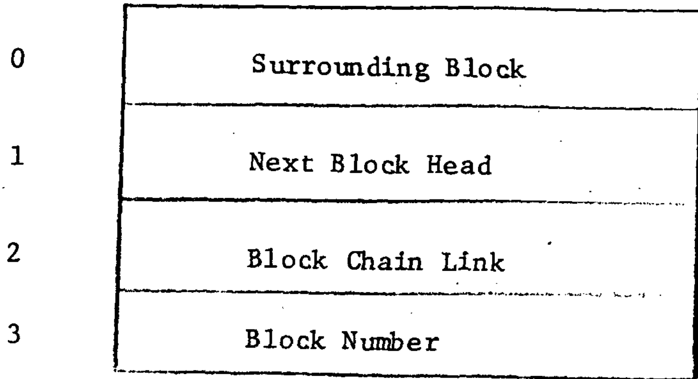
Whenever there is no match of the possible legal symbols then an error routine is called. Any label in the last column starting with ER is the label of an error routine. These routines are not described because all they do is print out the label of the error routine as the error message and then interpretation is stopped.

## 5.3 Symbol Table

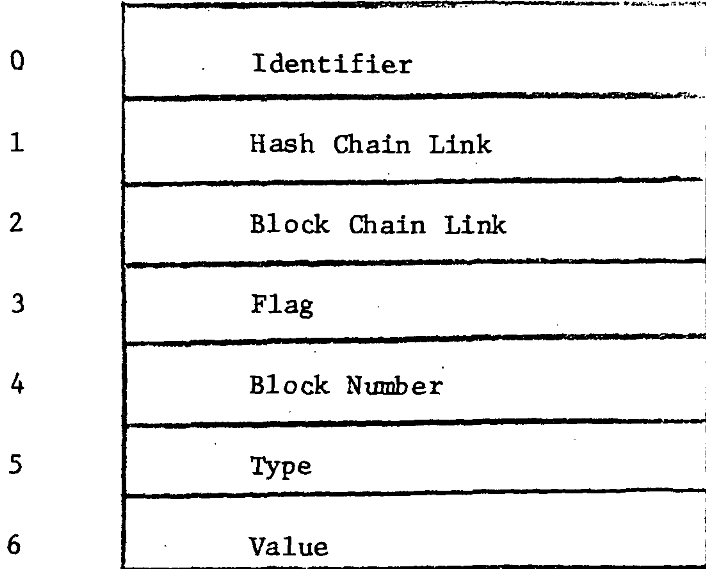
The symbol table consists of two parts. There is a hash table of 128 locations to which an identifier can be hashed. There is also a chain of block heads. Each block has a block head to which all identifiers declared in that block are linked.

Each location or bucket in the hash table is a pointer to a chain of the identifiers that hashed to that bucket. Each entry of an identifier consists of seven items or fields of information as seen in Fig. 8(b). An entry of an identifier is not in the hash table itself but only linked to the table through the hash chain starting at the bucket the identifier hashed to. Each item of information in an entry does not require the same amount of space. In this design in order to maintain clarity and simplicity each item of information is stored in the same amount of space. In a practical implementation the structure of an entry will be more complicated to make efficient utilization of space. The hash function will be described in the section on execution. It only hashes the identifier name so identifiers from different blocks with the same name are stored by chaining them in decreasing order of their block numbers.

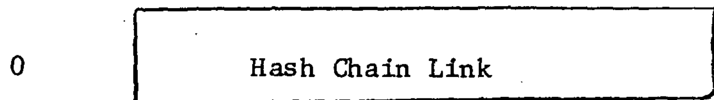
Since the subset is like Algol it is necessary to keep track of global and local variables. During execution the flag field (field 3) of an identifier entry is used to keep track of whether the particular identifier is presently defined. Therefore, whenever a block is entered all variables and labels declared in that block must have their flag field set to 1. Correspondingly whenever a block is exited the appropriate flag fields must be set to 0. To facilitate the above processes all the variables and labels of each block are linked together using field 2. Each block has a block head




(a) format of block head entry




(b) format of identifier entry




(c) format of the table bucket entry

Fig. 8 Formats of the symbol table

(see Fig. 8(a) for format) which has a link to the chain of identifier symbol table entries for its block. The block head also contains the address of the surrounding block, the block number, and the address of the next block head of the chain of block heads.

Another case to take care of is the redeclaring in a subblock of an identifier already declared. As in Algol this identifier should be a new location and should be used until the block is exited at which time the old declaration and location should become active again. The natural way to take care of this is some type of a stack. All identical identifiers are linked together in the hash chain in decreasing order of block numbers. Since blocks are numbered in order of their occurrence in the program this means that the most recent declaration of the identifier will be found first as one goes through the hash chain looking for the identifier. The value of an identifier willl be set to zero whenever a block is entered.

## 5.4 Table Routines

The basic structure and concepts of the symbol table have just been given. The construction of the symbol table is done by what is called table routines. During syntax analysis whenever certain symbols or constructs are recognized, a table routine is called to update the symbol table according to what was just recognized.

If a BEGIN is encountered followed by a declaration then a new block head must be formed. If BEGIN END is on the stack of the analyzer then the end of a block has been found and the appropriate processing is done. Whenever a

variable or label is declared then it must be entered in the symbol table.

At the end of syntax analysis the entire symbol table has been formed. During execution all that need be done is to keep the variables and labels up to date as execution proceeds. Whenever a block is entered the flag field for all variables and labels in that block can easily be set to 1 by going through the chain for that block. Upon exiting a block the flag fields can be set to 0.

Before syntax analysis various things had to be initialized. AV must be initialized to 129 in order to reserve space for the hash table. AV is the pointer to the next available space in the available space array SP. SP is a string array which holds 12 characters per array element. This array SP and the form of the symbol table are very simple and clear, and would have to be changed for efficiency in practical implementation. Every bucket of the hash table (the first 128 locations of SP) must be initialized to 0 since no hash chains exist at the start of interpretation.

### 5.4.1 Routine EXEC1

The flow chart for table routine EXEC1 is shown in Fig. 9(a). When the first BEGIN symbol of an Algol program is encountered this table routine is called to initialize BLKNO which will be used to keep track of the block numbers and also P2 which is a stack pointer.

### 5.4.2 Routine EXEC2

The flow chart for table routine EXEC2 is shown in Fig. 9(b). Table routine EXEC2 is called when it is definite that a block and not a compound statement is being recognized. It is definite when an INTEGER sym-

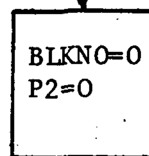
Table 6 Symbol Table Routines

| Table routine | Function description |
|---|---|
| EXEC1 | initialization of BLKNO and P2 |
| EXEC2 | constructs a block head |
| EXEC3 | pop-up stacks ST2A and ST2N |
| EXEC4 | enters variable in symbol table SYM |
| EXEC5 | enters label of outermost block in symbol table SYM |
| EXEC6 | enters label in symbol table |

Table 7 Description of Names Used in Table Routines

| TERM | DESCRIPTION |
|---|---|
| AV | a pointer to next available location in SP |
| SP | an array of available space |
| CRB | a pointer to first location in current block head |
| BLKNO | a number of the latest block encountered |
| ST2N | a stack for the block numbers |
| ST2A | a stack for the addresses of block heads |
| P2 | a pointer for ST2N and ST2A |
| OB | a pointer to first location in last block head |
| K1 | a pointer for PS |
| TT | a buffer in which the address of successive items of a hash chain are stored |
| N | a buffer containing a string of 12 characters placed there by the scanner |
| TR | a buffer in which the address of the last item processed in the hash chain is stored |
| ER9 | a procedure to indicate an identifier declared twice in one block |

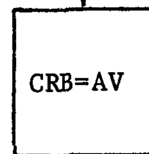
start




return

(a) table routine EXEC1

start




AV=AV+4
BLKNO=BLKNO+1

BLKNO=1?

Y

N

SP(CRB)=ST2A(P2)
AP(OB+1)=CRB

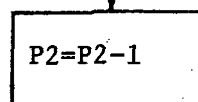
SP(CRB+2)=0
P2=P2+1

ST2N(P2)=BLKNO
ST2A(P2)=CRB
SP(CRB+3)=BLKNO
OB=CRB

return

(b) table routine EXEC2

start




return

(c) table routine EXEC3

Fig 9. Flowcharts for table routines EXEC1, EXEC2, and EXEC3

bol is found following a BEGIN symbol. This routine constructs a block head for this block and puts the appropriate values in the various parts of the block head. Fig. 8(a) should be referenced again at this point.

CRB will always point to the first location of the current block head so it is set to AV the next available location in the available space array SP. Four is added to AV next to reserve four locations as needed for a block head.

BLKNO is then incremented to obtain the number of this block. Remember BLKNO is initially zero so that addition gives the correct number for all blocks including the first one. Note that BLKNO is never decremented since it is only used to number the blocks in the order they are encountered in a single pass through the program. During execution BLKNO will be used and it will be decremented then since GOTO's cause blocks to be entered many times.

For all blocks except the first block the address of the surrounding block in ST2A(P2) is stored in SP(CRB), the surrounding block field. Also to maintain the chain of block heads the address of the present block head in CRB is stored in SP(OB+1) which is the block head chain link field of the last block head. Note that OB is used to save the address (subscript) of the last block head. Since none of this need or can be done for the first block head there must be a test for the first block head so that the above actions are not performed for that block head.

The block number and location of this block head are stored in the parallel stacks ST2N and ST2A. The single pointer P2 is used for both stacks. The reason for stacking this information is that when any subblocks are exited any labels encountered must be placed in the block chain for the outer block. Therefore, information on the outer block must be saved. BLKNO is also stored

in SP(CRB+3), the block number field.

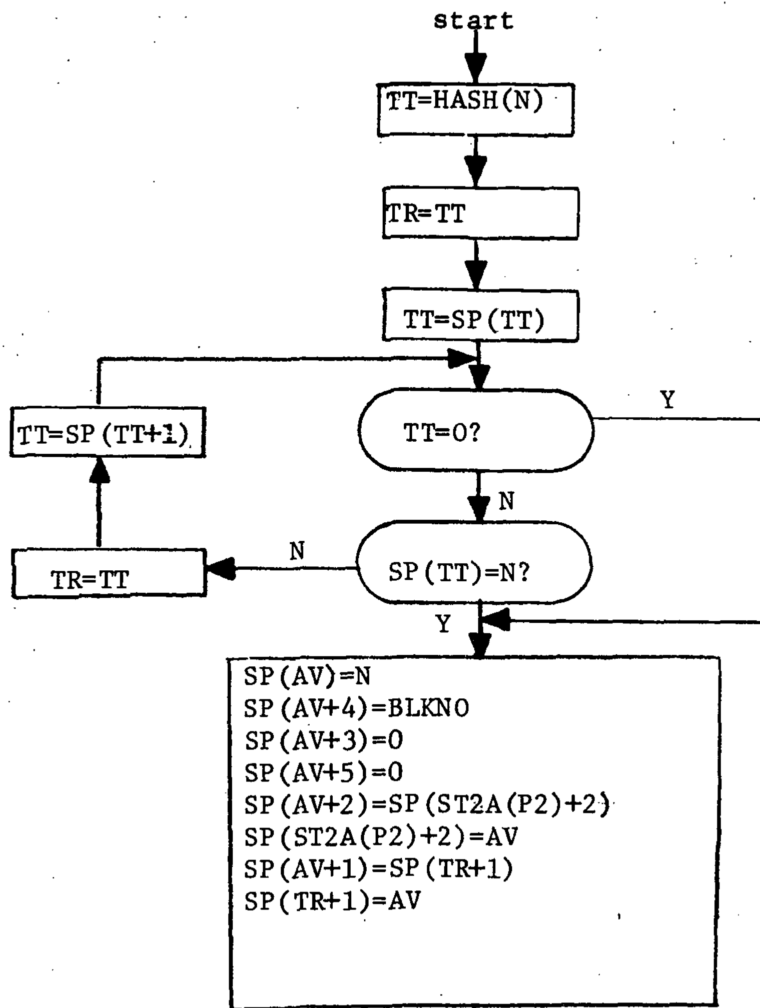
Since no identifiers have been declared in this block yet, zero is placed in SP(CRB+2) which is the block chain link. Note that CRB is saved in OB so that the next block head can be linked to this block head to maintain the block head chain.

### 5.4.3 Routine EXEC3

The flow chart for table routine EXEC3 is shown in Fig. 9(c). When the three symbols BEGIN 1→ END are found on the stack FE then the end of the current block has been reached and all that need be done is pop the top elements of the stacks ST2A and ST2N by decrementing P2. This makes the block number and starting location of the block head of the surrounding block available again as is necessary.

### 5.4.4 Routine EXEC4

The flow chart for table routine EXEC4 is shown in Fig. 10(a). Whenever a variable is declared EXEC4 is called to enter the variable in the correct part of the symbol table. The identifier is hashed and the resultant location in the hash table is placed in TT. SP(TT) points to the hash chain for this location of the hash table. If TT becomes zero at any point then the end of this hash chain has been reached without finding the variable since zero marks the end of a hash chain. It should then be entered at the end of the hash chain. If the identifier is encountered in the hash chain then it should be linked into the chain just before its first occurrence. Note that each declaration in a different block requires a different symbol table entry. Whenever another declaration of a variable occurs in a new block, that block has a higher block number than any previous declarations. Therefore, since




AV=AV+7

return

(a) table routine EXEC4

start

T=HASH (N)

SP(AV)=N
SP(AV+4)=0
SP(AV+5)=1
SP(AV+3)=1
SP(AV+1)=0
SP(T+1)=AV

AV=AV+7

return

(b) table routine EXEC5

Fig. 10. Flowcharts for table routines EXEC4 and EXEC5

identifiers are kept in decreasing order of block numbers the variable can be linked before the first occurrence as mentioned above.

When the variable is entered into the hash chain in either of the above cases several additional items of information must be entered. The variable name is stored in SP(AV). The block number is BLKNO since the block has just been entered and BLKNO has just been updated. It is stored in SP(AV+4). The type field, (SP(AV+5)), is set to zero meaning a variable is stored in this entry. SP(AV+3), the flag, is set to zero since it stays zero during execution except during execution of its particular block. The entry must also be linked into the block chain for the current block. The block chain link of the current block, (SP(ST2A(P2)+2)) always has the address of the identifier most recently declared in the block. This address is stored in SP(AV+2). The address of this entry is then stored in SP(ST2A(P2)+2) to finish linking this entry into the block chain. The linking into the hash chain is done using the variable TR which contains the address of the entry that is to be before the current entry. Finally AV is updated to complete the routine.

### 5.4.5 Routine EXEC5

The flow chart for table routine EXEC5 is shown in Fig. 10(b). It is possible that the outermost block has a label which will be global to the entire program. It is entered in the hash table and is put in hypothetical block 0. It will be left active at all times and has the lowest possible block number.

### 5.4.6 Routine EXEC6

The flow chart for table routine EXEC6 is shown in Fig. 1. Table routine EXEC6 is called whenever a label is encountered during syntax analysis. The entry of a label into the symbol table is similar to the entry of a varia-

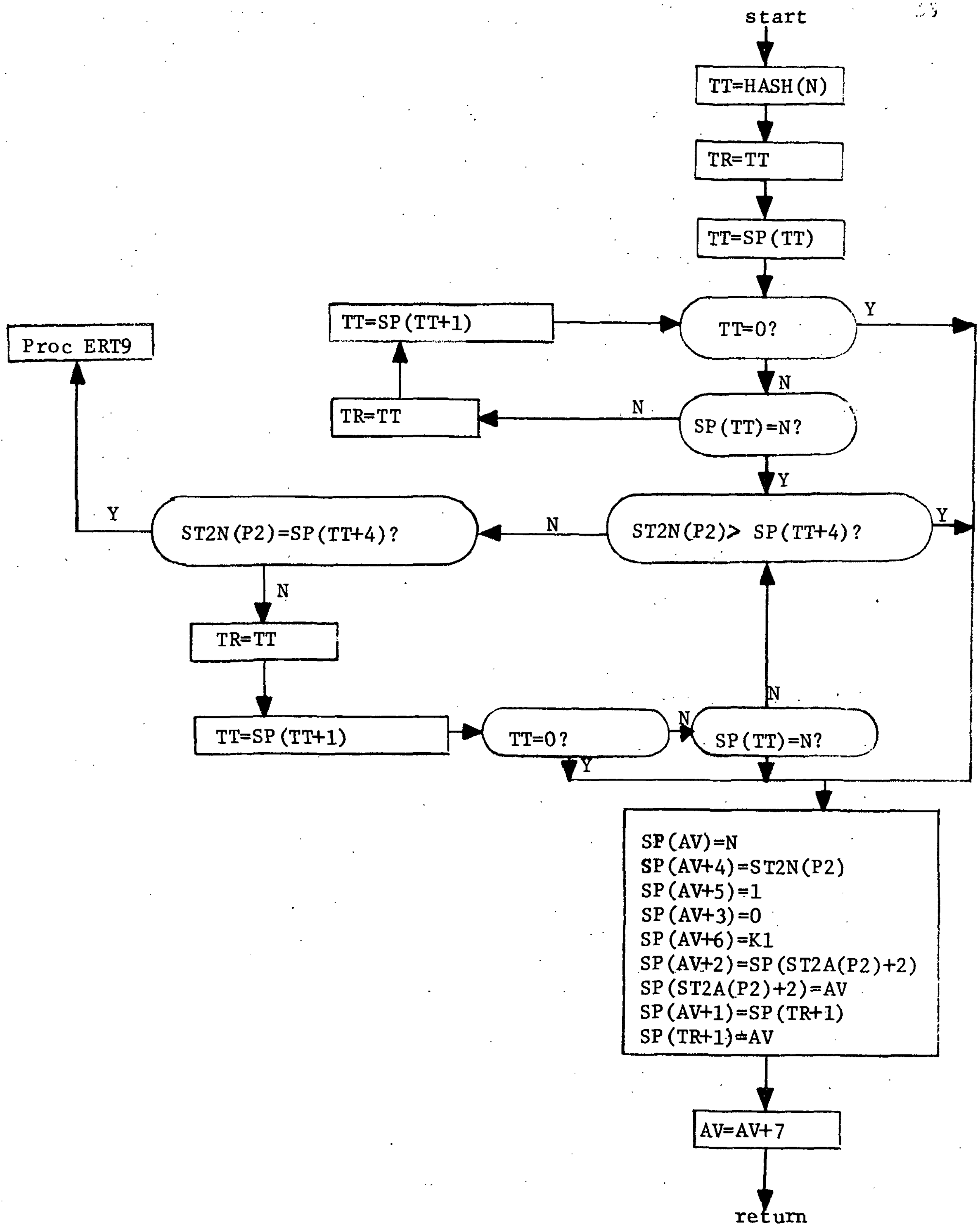



Fig. 11. Flowchart for table routine EXEC6

ble. The label is hash-coded and then the search through the appropriate hash chain is performed. If the end of the hash chain is reached (TT=0) then the label is linked onto the end of the chain. If the label is found as a label or variable then the current block number is compared to the block number of the entry just found. Note that the current block number must be obtained from stack ST2N. This is necessary since labels may be found anywhere in a block which means the given label may occur after some subblocks and thus BLKNO no longer contains the number of the current block. Since a label may occur and thus be declared anywhere in a block, its block number may be greater or less than any of the previous entries with the same name. A series of tests must be made to determine exactly where among the series of entries with identical names the label must be linked. A test is also made to make sure the label has not already been declared in the same block and also that no variable has been declared with the same name in this block. This is done by merely checking that no entry has the same name and same block number as the label now being entered. If this occurs then error procedure ERL9 is called into action. It prints out an error message and stops interpretation.

When the label is linked into the hash chain the process is the same as it was for variables except for the following changes. The type field, (SP(AV+5)), has one stored in it. The value field has the location of the label in the postfix string, PS, placed in it. K1 still points to the label so K1 is stored in the value field. The block number is obtained from the top of the stack ST2N and placed in SP(AV+4), the block number field.

## 6. Executor

The last part of the interpreter is the execution of the postfix string which the syntax analyzer produced. The configuration of the Executor is shown in the block diagram of Fig. 12. Remember that there are actually two parallel postfix strings as shown in Fig. 12. Buffer PS contains the symbols (internal code) outputted by the analyzer and buffer PS1 contains the semantics of those symbols. In the semantic string are the values of the constants and the names of the identifiers. Buffer PS is an integer array. Buffer PS1 is a string array; each of its elements holds 12 characters.

Execution is performed with the aid of a stack for the operands. A pointer, K1, is used to go through the postfix string, PS, one symbol at a time. In general an operand is stacked whenever encountered while an operator causes some operation on the top elements of the stack. The stack has two fields for each entry. It has a value field and it has a kind field which tells whether the value is the location of an identifier or it is the actual value of an operand. When an identifier is placed on the stack its location and not its name is placed on the stack. Therefore, the value field can be integer instead of string which is more efficient in general. The two fields of the operand stack are implemented by using two parallel stacks. The value field of an operand is stored in stack FE and the kind field is stored in stack FE1. Both stacks use the same pointer K.

After initialization of K1, BLKNO, CRB, and K a loop is entered as seen in Fig. 13. This loop consists of going to a section of the executor program depending on what symbol in PS K1 points to. That section performs the appropriate execution for the symbol pointed at and then returns to repeat the loop. K1 is kept updated by the individual sections of the execution of

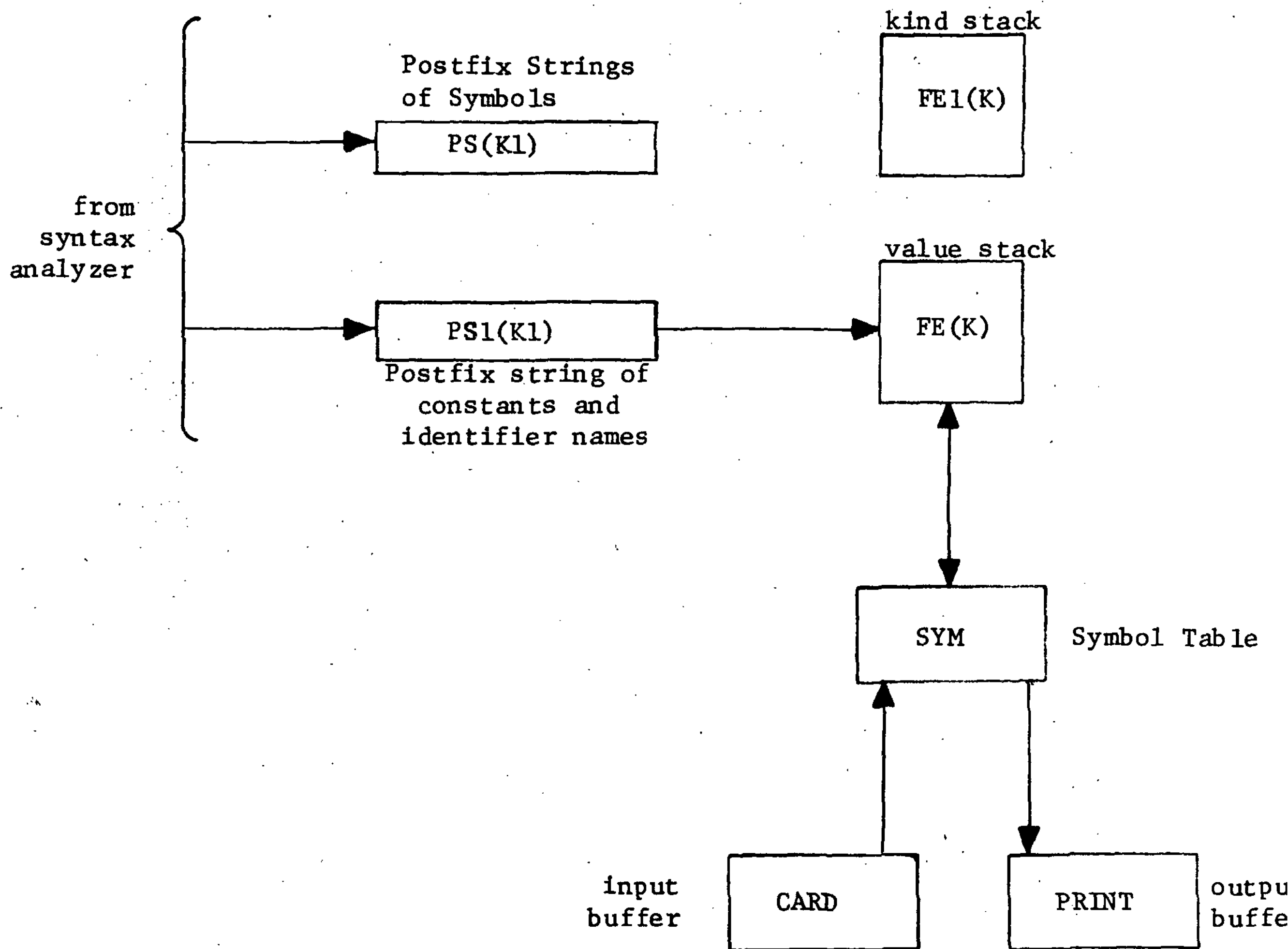



Fig.12. Configuration of the Executor

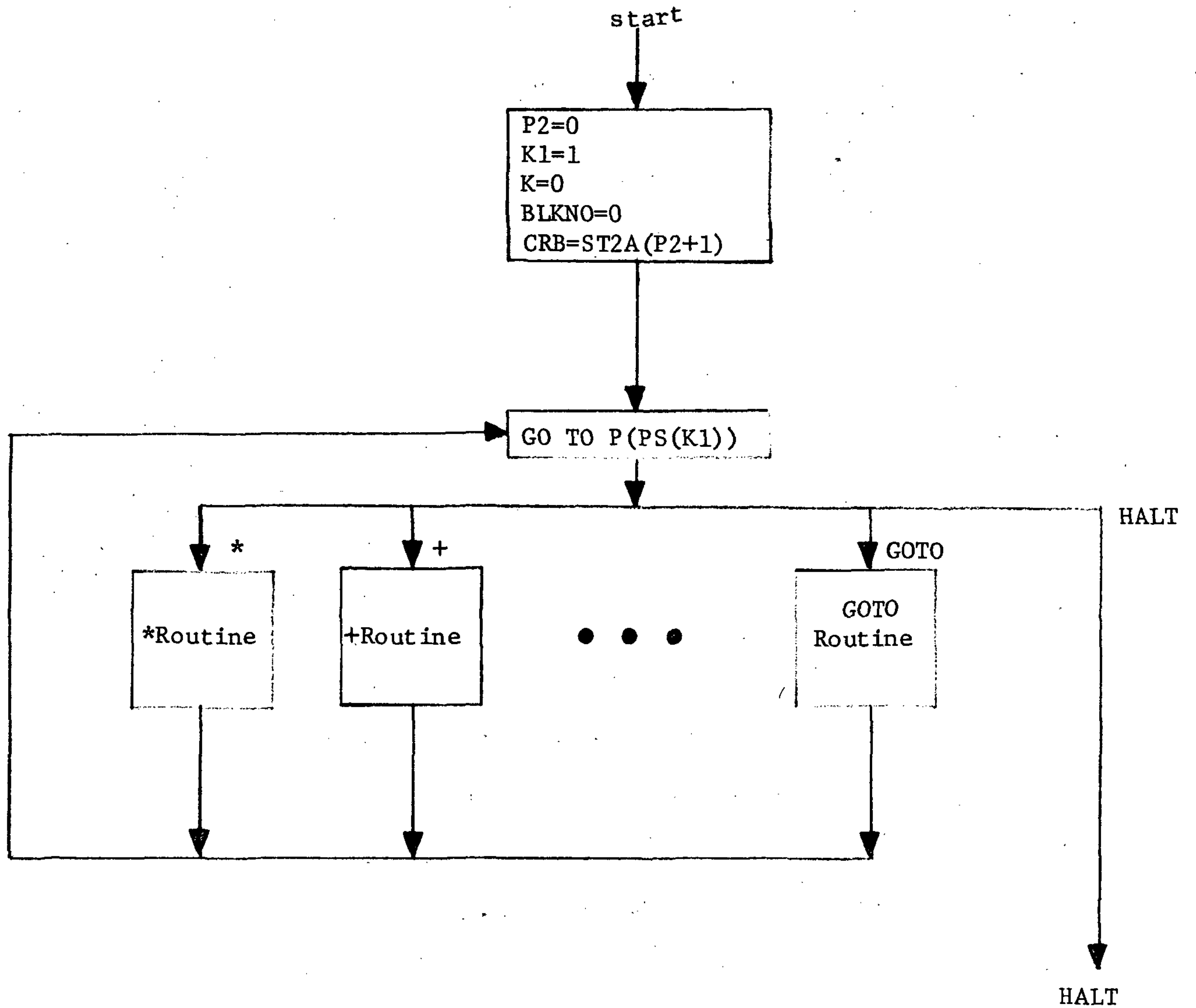



Fig.13 Flowchart of actual execution

the execution program. An error or the reaching of the symbol HALT causes the stopping of the executor and of the entire interpreter.

## 6.1 <u>BEGIN and BEG Routines</u> (Figs. 14 and 17)

Whenever a block is entered it is necessary to set the flag fields of all variables and labels declared in that block. When a BEGIN symbol is encountered a block is being entered. When a BEG symbol is encountered a compound statement is being entered and nothing need be done to the symbol table. The BEG routine merely increments K1 by one and ends.

As in the table routines BLKNO will keep track of block numbers so that whenever a block is entered it is only necessary to add one to BLKNO to get the number of the block being entered. Since there are GOTO's to be executed it will be more difficult to keep BLKNO correct. In particular during the execution of a GOTO symbol much work must be done to keep BLKNO correct. CRB will point to the block head for the block being executed.

The routine for the BEGIN symbol starts by adding one to BLKNO to obtain the correct block number and then going through the block head chain to find the block head for this block. It is only necessary to go in the one direction to find the block head becuase of the order in which the block heads were formed and linked together. A subblock is always formed and linked somewhere after the block it is contained in.

Next the routine proceeds to activate (set to one) the flag fields of all labels and variables of this block and also intializes all variables to zero. This is done easily since they are all linked together with the block head. Finally the pointer K1 is incremented so that it points to the next symbol and the routine ends.

Table 8 Names Used in Execution Routine

| Name | Description |
|---|---|
| BLKNO | the number of the next block to be entered minus one |
| FE | value operand stack |
| FE1 | kind operand stack |
| PS | postfix string of symbols |
| PS1 | postfix string of identifier names and constants |
| K | pointer for FE and FE1 |
| K1 | pointer for PS and PS1 |
| CRB | address of block head of current block |
| T | temporary variable |
| FIND | integer procedure which finds address of identifier in symbol table |
| HASH | integer procedure which hashes identifier |
| C3 | a counter |

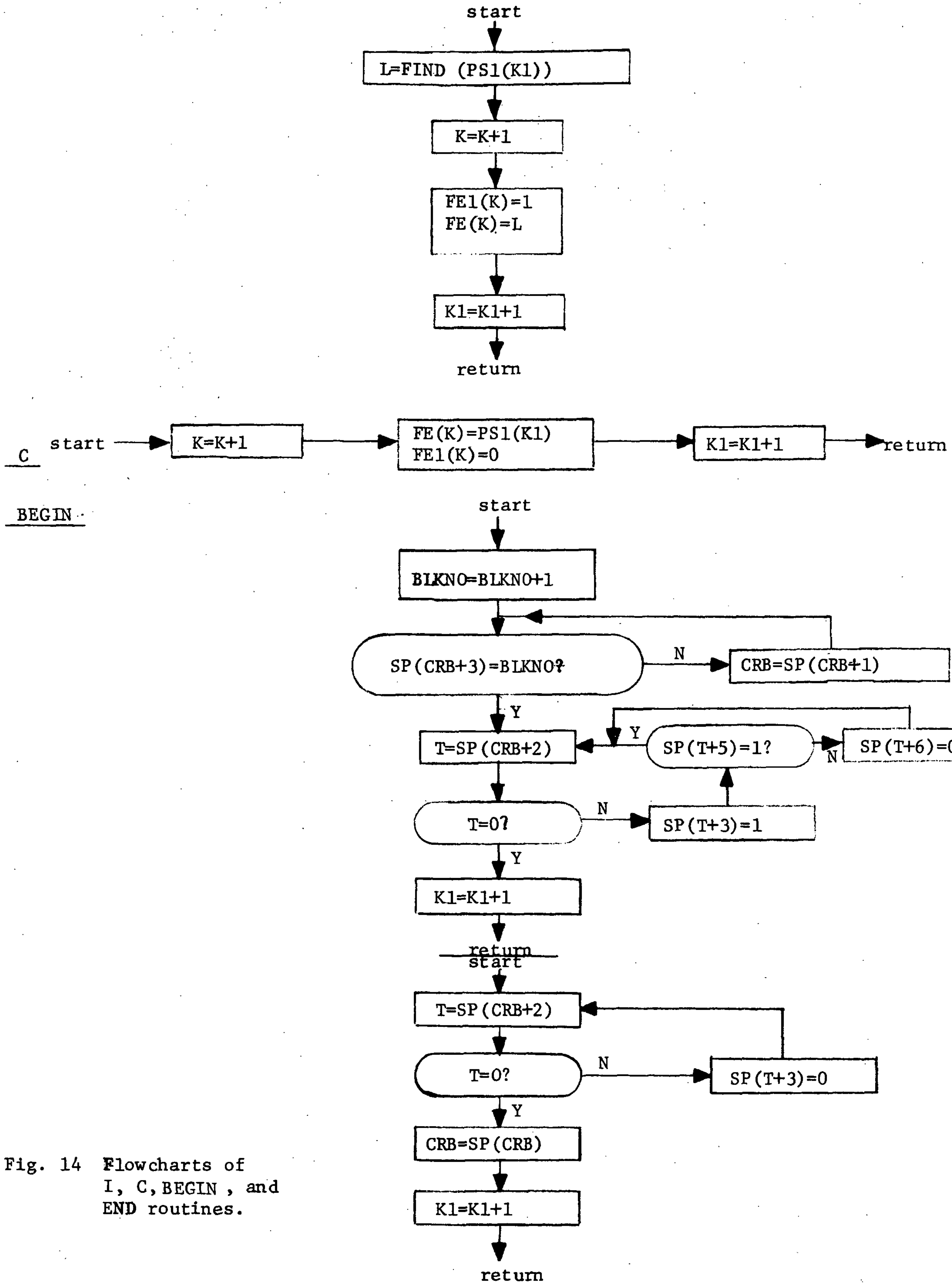



Fig. 14 Flowcharts of I, C, BEGIN, and END routines.

## 6.2 END and ENDE Routines (Figs. 14 and 17)

When an ENDE symbol is encountered the end of a compound statement is reached and nothing need be done to the symbol table. The ENDE routine increments K1 by one and ends. The END routine must update the symbol table since a block is being exited. First all variables and labels declared in the block being exited are deactivated by setting their flag fields to zero. Then the address of the surrounding block is obtained from SP(CRB), the surrounding block field of the block being left, and stored in CRB so that CRB will contain the address of the block head of the block which is once again the current block. Pointer K1 is incremented by one so that it points to the next symbol and then the routine ends.

## 6.3 I Routine (Fig. 14)

When the symbol for an identifier is encountered a simple routine proceeds to place its address (subscript in SP) in the operand stack. The first action is the calling of the integer procedure FIND which obtains the address (subscript in SP) of the identifier stored in PS1(K1). That address is stored in L. It is then stacked on the operand stack FE which is the value field. A one is also stored in the parallel operand stack FE1 which is the kind field. A one in FE1 means there is the address of an operand in FE while a zero means that the actual value of the operand is in FE. Pointer K1 is then incremented by one and the routine ends.

## 6.4 C Routine (Fig. 14)

When the C symbol is encountered in PS(K1) then its value stored in PS1(K1) is stacked on the operand stack FE. A zero is stored in FE1 to signify that the actual value of an operand is in FE. K1 is incremented by one and the routine ends.

6.5 READ and WRITE Routines (Fig. 15)

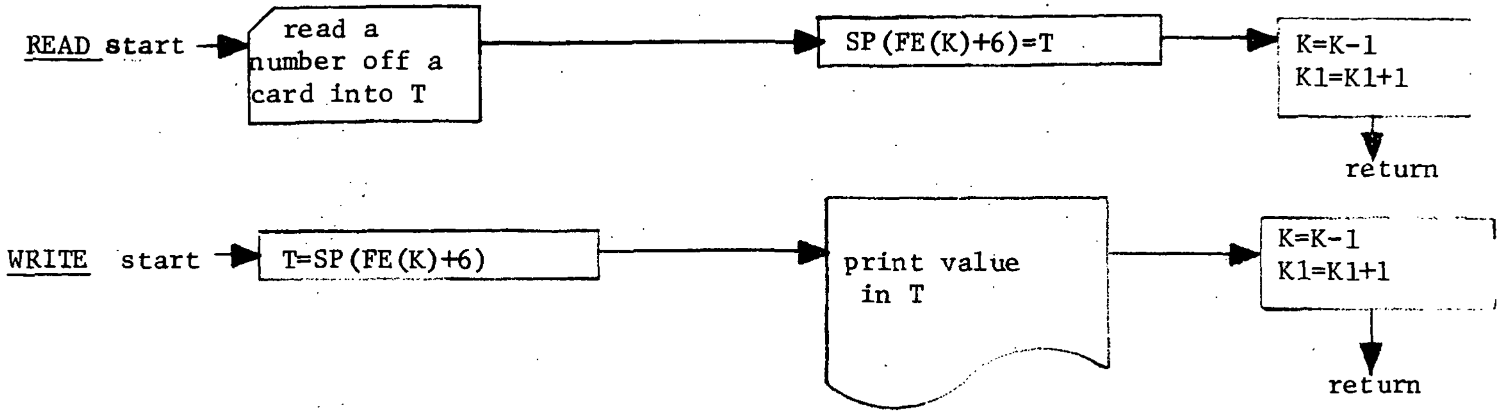
The READ routine reads a number off a card and then stores that number in the symbol table value field of the variable whose address (subscript) is on top of the operand stack. The WRITE routine on the other hand obtains the value of the variable whose address is on top of the operand stack and writes out that value. At the end of both routines the top element on the operand stack is removed by decrementing by one K. K1 is incremented by one and the routine ends.

6.6 ↑, *, /, +, and - Routines (Figs. 16 and 17)

↑, *, /, +, and - are all binary operators and the routines that are called when they are encountered are almost identical. In all five routines the first action is to check the top two operands of the operand stack to see what is the kind of their values. If the kind field of either is one, meaning the value is the address of a variable, then the value of the variable is obtained and the address is replaced with the actual value of the variable. The two operands are then combined as the particular operator in question requires and the result is placed in the location below the top of the stack (second stack location). The kind of this location of the stack is set to zero since an actual value has just been stored. The stack pointer K is decremented by one so that the result is on top of the stack and then finally K1 is incremented by one.

6.7 = and ≠ Routines (Fig. 16)

= and ≠ can be considered binary operators. As for the other binary operators the first action is to make sure that the top two operands on the operand stack are actual values. These two values are then compared. If they are the same then the result is one (true) if it is the = routine and




FIND

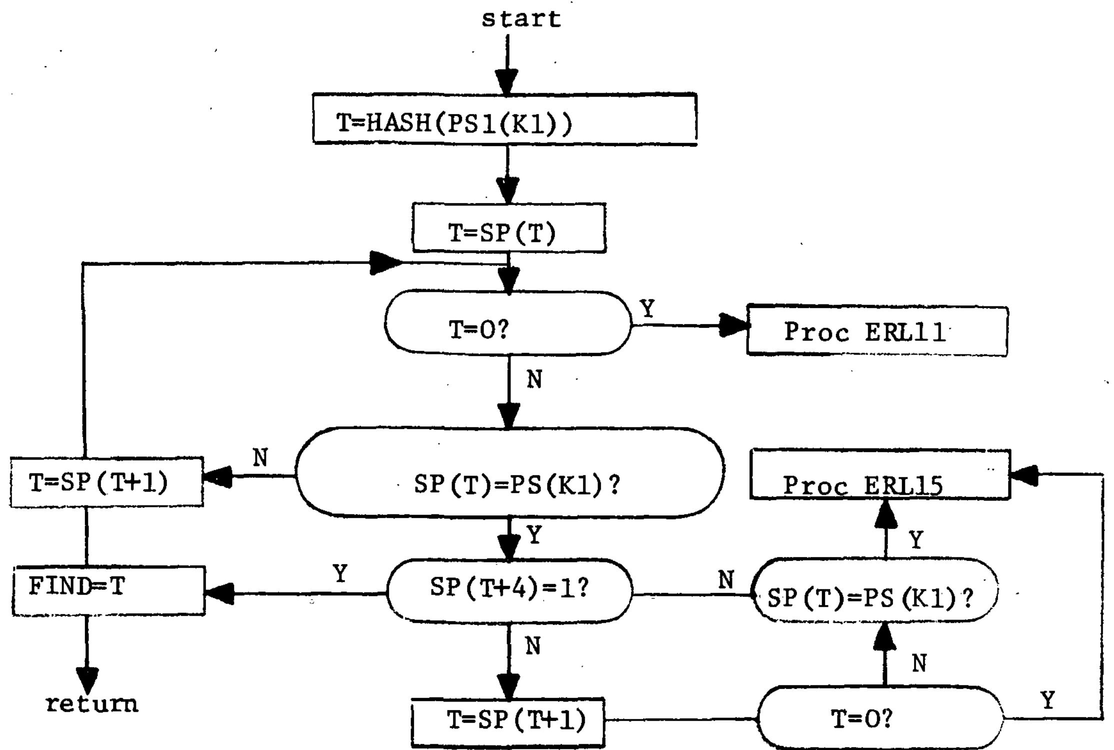



Fig. 15. Flowcharts of READ and WRITE routines and of procedure FIND

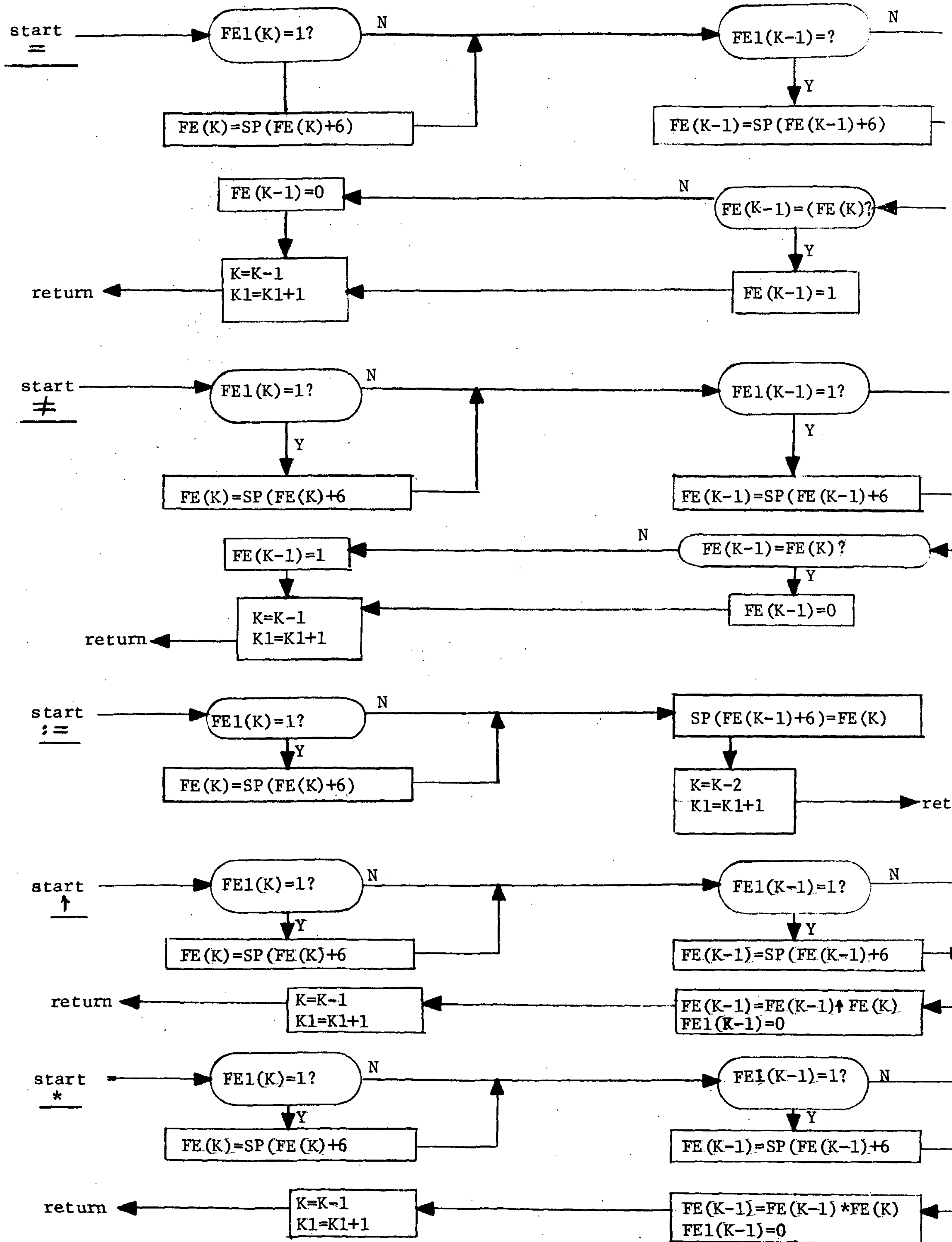
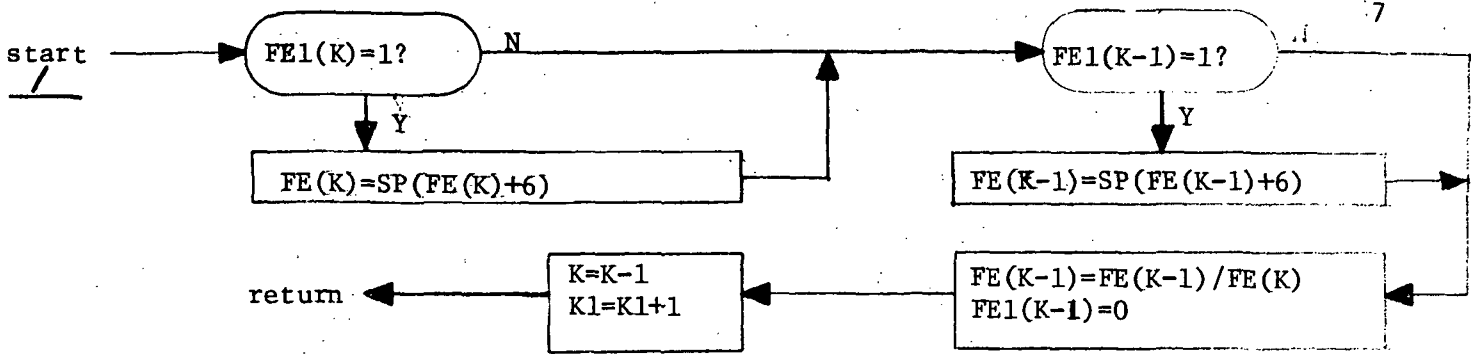
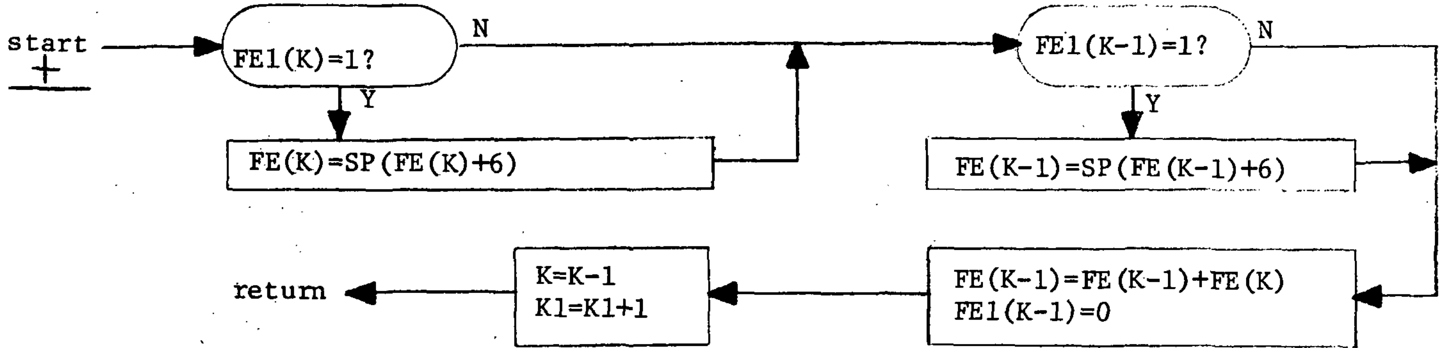
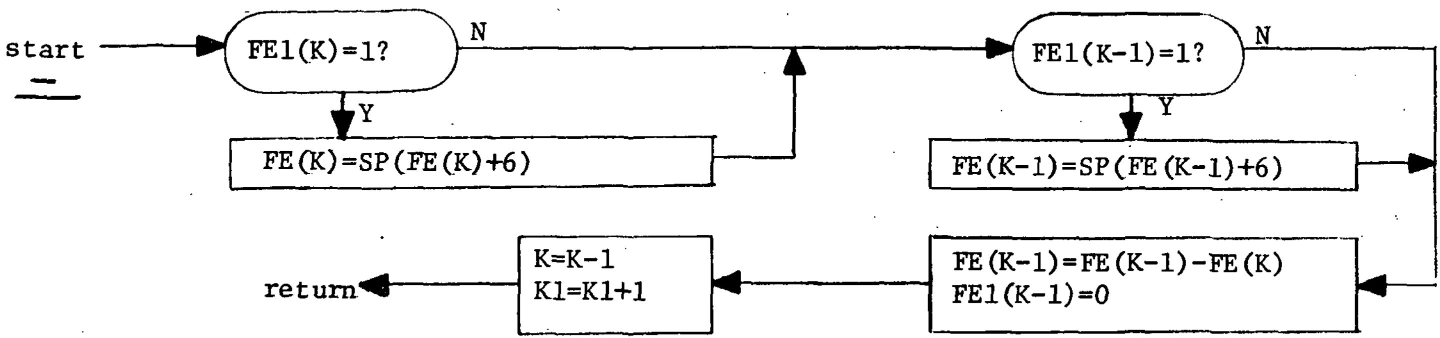


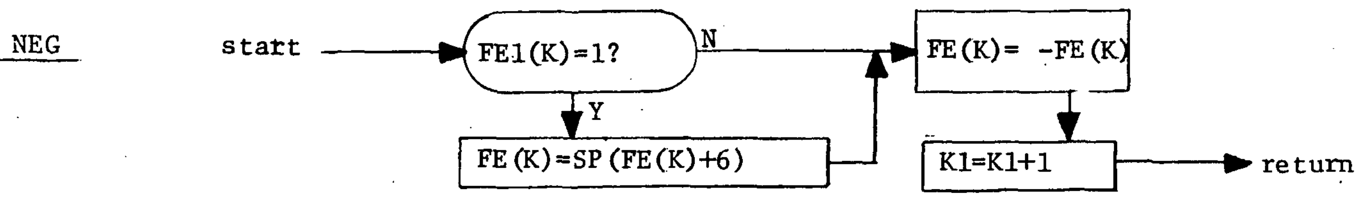

Fig. 16 Flowcharts of =, ≠, :=, ↑, and * routine

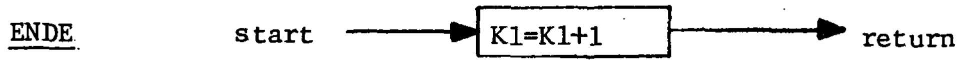


L start K1=K1+1 return





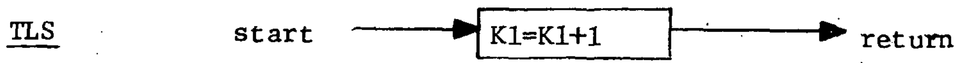



Figure 17
Flowcharts of /,+,−, HALT, L, NEG, BEG,ENDE ELSE, and TLS routines

zero (false) if it is the ≠ routine. If the two values are different then the result is zero if it is the = routine and one if it is the ≠ routine. The result is then placed in the second location of the stack. K is decremented by one and K1 is incremented by one.

### 6.8 := Routine (Fig. 16)

:= means that the value of the top operand of the stack should be stored in the symbol table value field of the variable whose address is in the second location of the operand stack. The first action is thus to make sure there is an actual value on top of the operand stack. That value is then stored in the appropriate value field. Two is then subtracted from K since neither operand is needed any longer. K1 is decremented by one as usual.

### 6.9 L Routine (Fig. 17)

When an L symbol is encountered a label is being encountered and nothing need be done. The : has been eliminated so all that need be done is increment K1 by one and then the routine ends.

### 6.10 NEG Routine (Fig. 17)

NEG is a unary operator. The top operand is made an actual value and then the negative of that value is stored in its place on top of the stack. K1 is incremented by one.

### 6.11 IF Routine (Fig. 18)

The important thing to remember with conditional statements is that the syntax analyzer has already checked for all errors and therefore this execution part can assume the correct symbols will be where expected. In particu-

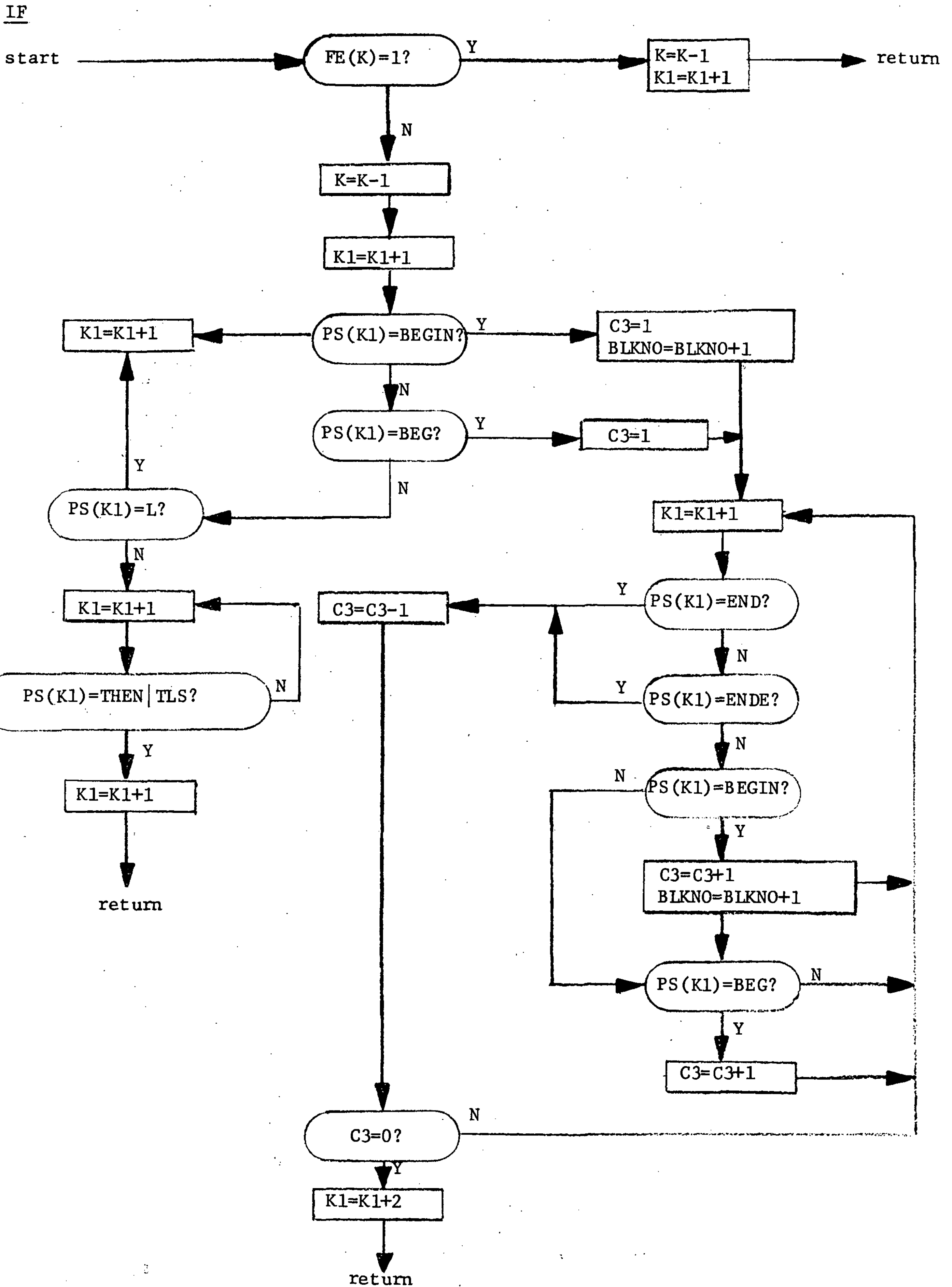



Fig. 18, Flowchart of the IF routine

lar the THEN and ELSE for a given conditional statement will be in their proper positions.

When an IF symbol is encountered the top operand on the stack is a value, one or zero. If it is a one then the boolean expression was true and the THEN statement is to be executed. If it is zero then the ELSE statement is to be executed. In either case after it is decided which statement is to be executed the operand is removed from the top of the stack by decrementing K by 1. If it was true then the statement immediately following the IF symbol in PS is the THEN statement so all that need be done is increment K1 by one and end the routine. When the THEN symbol is encountered then the THEN statement is over and at that point the skipping of the ELSE statement will be taken care of.

If the boolean expression is false then the THEN statement must be skipped so that the ELSE statement is the next statement executed. It is not possible to just go through PS until a THEN symbol is encountered. Since a THEN statement may be a block or compound statement which may itself have conditional statements, blocks or compound statements it necessary to keep track of how many BEGIN, BEG, END, and ENDE symbols are skipped. The block or compound statement may also have labels so the checking for a block (a BEGIN symbol) or compound statement (a BEG symbol) must continue if there are labels (L symbols) until the first symbol which is not an L symbol is reached. Then if there is not a BEGIN or BEG symbol all that need be done is to keep skipping symbols until a THEN or TLE symbol is reached. K1 is incremented by one and the routine ends.

If there is a BEGIN or BEG symbol then it is necessary to continue skipping symbols until the end of the block or compound statement is reached. To do this it is necessary as mentioned above to keep count of the BEGIN, END, BEG, and ENDE symbols encountered. Only when the matching END for the BEGIN

or matching ENDE for the BEG is encountered has the whole block or compound statement been skipped. Only the BEGIN, BEG, END, and ENDE symbols must be checked for. This is done by counting the BEGIN and BEG symbols in the counter C3. When an END or ENDE is encountered one is subtracted from C3. Therefore, when C3 reaches zero the entire block or compound statement has been skipped. The syntax analysis has made sure that all BEGIN symbols are matched by END symbols and all BEG symbols have matching ENDE symbols. It is no longer necessary to deal with such problems here. K1 is then incremented by two so that the statement after the THEN symbol is executed next. Note that it is assumed that a THEN symbol follows the block or compound statement since it was the job of the syntax analyzer to produce only proper postfix code.

One other action is performed when skipping blocks. BLKNO must be incremented by one for every block skipped so that it has the proper value the next time it is needed.

6.12 THEN Routine (Fig. 19)

When a THEN symbol is encountered then the end of a THEN statement has been reached and it is necessary to skip an ELSE statement. This process is identical to the process just described for skipping a THEN statement except an ELSE symbol instead of a THEN symbol marks the end of the statement being skipped.

It should be noted here that a GOTO branch into a conditional statement is handled properly as it is in standard Algol. A transfer to the THEN statement is legal and at the end of the execution of the statement the ELSE statement is skipped because a THEN symbol is encountered which results in the skipping of the ELSE statement as mentioned above.

THEN

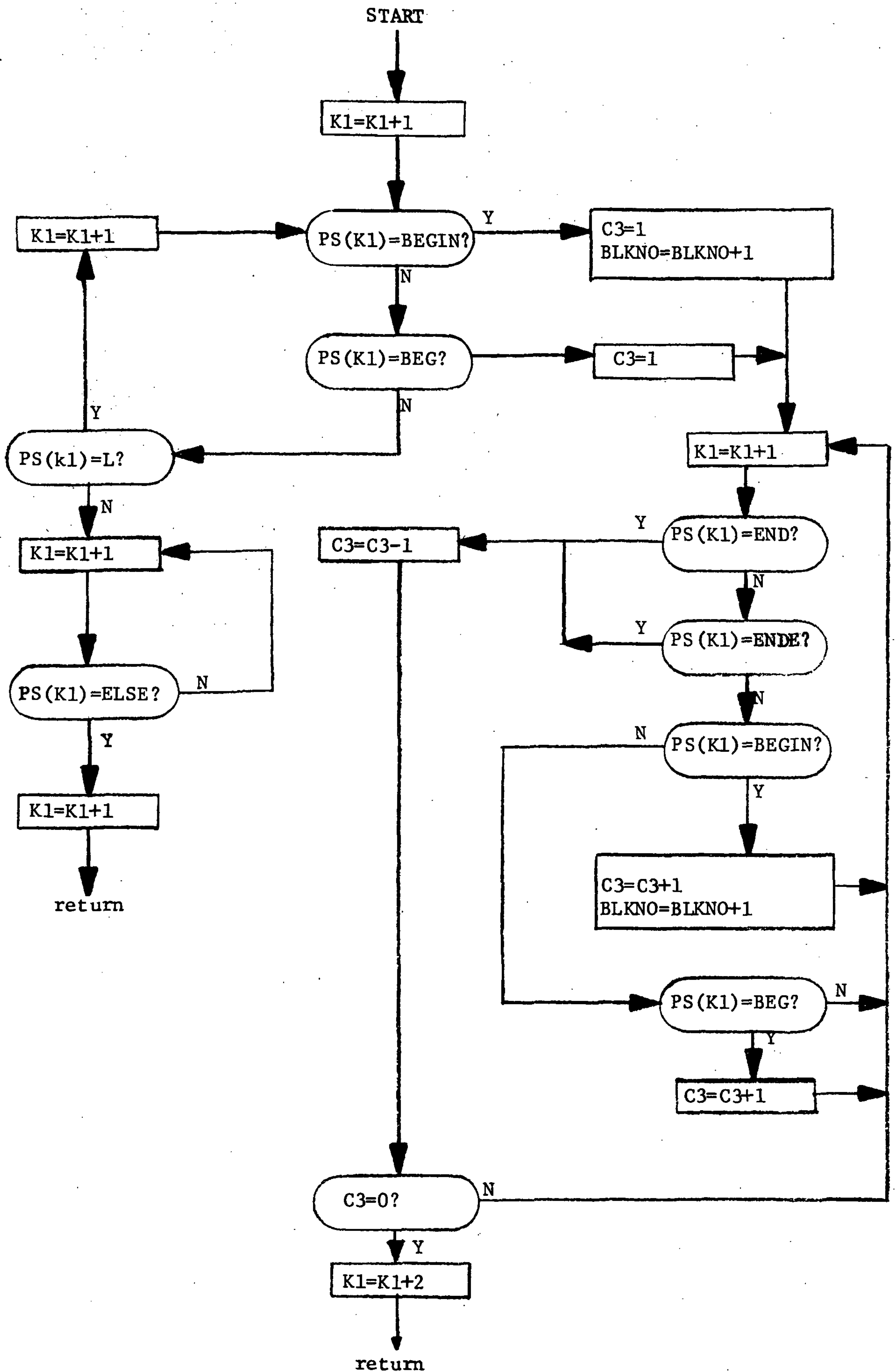



Fig. 19 Flowchart of the THEN routine

6.13 TLS Routine (Fig. 17)

The TLS symbol is encountered at the end of the execution of the THEN statement of a conditional statement which has no ELSE statement. All that need be done is increment K1 by one so that the first symbol of the next statement is pointed to and then the routine ends.

6.14 ELSE Routine (Fig. 17)

The ELSE symbol is encountered at the end of the execution of an ELSE statement and all that need be done is increment K1 by one so that first symbol of the next statement is pointed to.

6.15 GOTO Routine (Fig. 20)

This routine is the most complicated execution routine because a jump across or out of blocks requires updating the symbol table and several block variables. Nothing need be done for compound statements so BEG and ENDE symbols are ignored.

The first action is to check if the top operand on the operand stack is in fact a label. If it is not then there is an error and procedure ERL 40 is called to print out an error message and end interpretation. Otherwise the value of the label is obtained and stored in T. Note that an undefined label will result in an error even before the GOTO routine is reached. The address in the symbol table of the label is fetched and placed on the operand stack by the I routine before the GOTO symbol is encountered. If the label is undefined in this block then procedure FIND of the I routine will not find the label in the symbol table and will cause error termination of the interpretation. The value of the label is the location in PS of the label. A test is then made

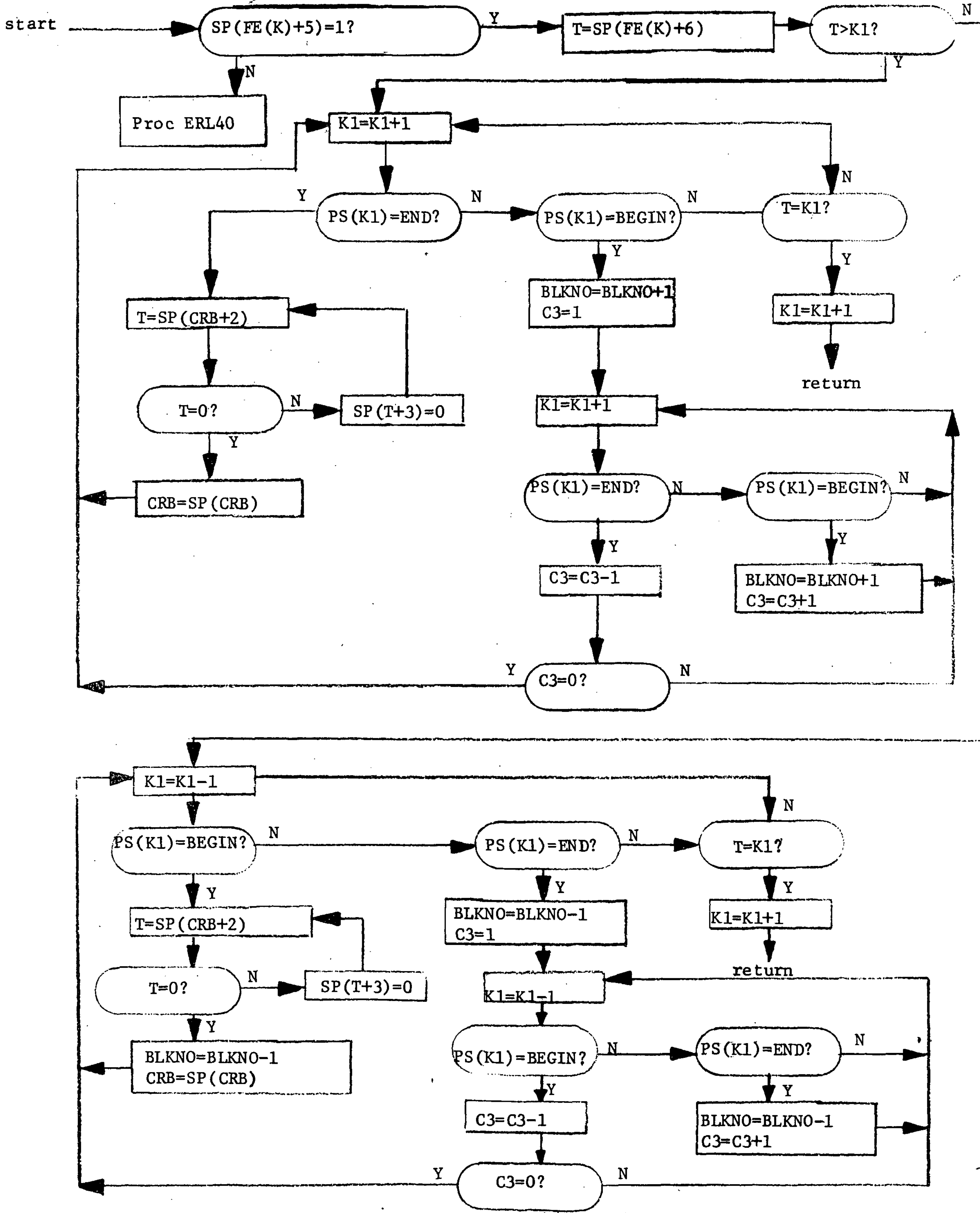



Fig. 20, Flowchart of the GOTO routine

to see whether this GOTO branch is a forward or backward branch. The branch is then processed in one of two ways depending on the result of the test.

If T is greater than K1 then it is a forward branch since K1 points to the present position in PS. The routine then proceeds to go through PS symbol by symbol until it reaches the Tth symbol. Whenever an END symbol is encountered then a block is being exited and it is necessary to deactivate all the variables and labels declared in the block being exited. This is done easily as usual since all entries for that block are linked together. CRB is loaded with the address of the surrounding block which is now becoming the innermost block. The address of the surrounding block is in SP(CRB), the surrounding block field of the block head of the block being exited.

If a BEGIN symbol is encountered then the routine continues toward position T of PS, but it keep tracks of the blocks being passed by. These blocks are not active since they were not global or local at the start of the execution of the GOTO. The label cannot therefore lie in these blocks. If it had been in one of these blocks there would already have been error termination since the label was not active. Nothing need be done to the variables and labels in these blocks since they will not be active at the end of this routine. Therefore, these blocks are passed through without doing anything except incrementing BLKNO by one each time a block is passed through (each time a BEGIN symbol is encountered). When K1 becomes equal to T the label has been reached so K1 is incremented by one and the routine ends.

If T is less than K1 then the branch is backward and a slightly different process is followed. Again the routine proceeds through PS this time decrementing K1 and processing the symbols encountered.

This time if a BEGIN symbol is encountered a block is being exited and the labels and variables declared therein must be deactivated. The routine goes through the chain for that block deactivating the entries. CRB is loaded

with the address of the block head of the surrounding block which is now becoming the innermost block. This address is obtained from SP(CRB). BLKNO must be decremented by one to keep it updated. If one remembers how block numbers are assigned then it is clear that BLKNO must be decremented by one here.

If an END symbol is encountered then a situation similar to that in the forward branch occurs. The END means the end of a block which is not active is being encountered. This block and any of its subblocks are to be skipped since the branch cannot be into any of these blocks. Also no changes are to be made to any variables and labels declared in these blocks. BLKNO must be kept updated again by being decremented by one for every block passed through (for every END symbol encountered). When the matching BEGIN for the END that started this block is encountered then the routine continues going through PS until T equals K1. Any END or BEGIN symbols cause the above procedures to be followed. Again when K1 reaches T, K1 is incremented by one and the routine ends.

### 6.16 HALT Routine (Fig. 17)

The last symbol in PS should be the HALT symbol. It causes the execution of the program to terminate and the end of the whole interpretation of this program also.

### 6.17 FIND Procedure (Fig. 15)

FIND is an integer procedure which takes the identifier stored in PS1(K1) and produces as its result the address (subscript) of the symbol table entry of that identifier. The first action is to apply the integer procedure HASH to the identifier which produces a subscript (bucket) in the hash table. The procedure then proceeds through the hash chain starting at this subscript (or bucket) until the identifier in question is found. If the iden-

tifier is not declared anywhere in the program then it will not be found in the chain and error procedure ERL11 is called to print out an error message and end interpretation.

If the identifier is found the procedure continues through the chain to find the first entry of the identifier that is active. The subscript of this entry is returned as the result of procedure FIND. If none of the entries of this identifier are active then the error procedure ERL15 is called to print out an error message and end interpretation.

## 6.18 HASH Procedure

The procedure HASH is an integer procedure which takes an identifier name and converts it to an integer between 1 and 128. Each identifier is considered to be twelve characters long. If the identifier is less than twelve characters long then blanks are filled in at the end of the identifier. The characters are then converted to binary using the Univac 1108 character code (8). This produces a 72 bit result. The EXCLUSIVE OR operation is performed on the first and last 36 bits producing a 36 bit result. The first five seven-bit sections of the 36-bit result then have the EXCLUSIVE OR operation performed on them in succession to produce a seven bit result. One is added to this result to produce a number between 1 and 128 and this number is the result of the procedure HASH. This number is the subscript (or bucket) of the given identifier in the hash table. The hash table is the first 128 locations in the available space array SP.

## 7. Interpretations of Sample Programs

In order to illustrate how the interpreter functions, interpretation of two programs are described below.

### 7.1 Program 1

Figs. 1 and 2 should be consulted to aid in the following of the interpretation of program 1 shown in Fig. 21. Program 1 computes the factorial of the number read into NUM and prints out the result.

#### 7.1.1 Initialization, Scanner, and Syntax Analysis

The space for the hash table is reserved first and the buckets are all set to zero. C1 and C2 are initialized and then procedure GC is called to load the first character into buffer CH. The analyzer then starts and calls the scanner to place the first symbol on stack FE.

The scanner does the converting of the input program into the symbols shown in Table 2, but it does not do this all at once. It obtains the next symbol only when a request to do this is made by the syntax analyzer. In Fig. 22 are the symbols that are passed to the syntax analyzer by the scanner. The symbols are in the order they are passed, but not in the actual form they are passed in. As has been done throughout this report the symbols are given, in place of the internal code which is actually used in an interpreter. For I and C symbols the actual identifier or constant is passed also and is given below the symbol in Fig. 22.

The syntax analyzer also controls the building of the symbol table by calling table routines at appropriate times. Since program 1 in Fig. 21 has only one block only a single block head is formed. The variables NUM, I,

```
BEGIN     INTEGER NUM, I, TEMP, NFACT;
                  READ(NUM);
                  TEMP:=1;
                  I:=1;
         CONTINUE I:=I+1;
                  TEMP:=TEMP*I;
                  IF I=NUM THEN NFACT:=TEMP ELSE GOTO CONTINUE;
                  WRITE(NFACT)
END
```

Fig. 21 Sample Program 1 (Calculation of N factorial)

```
BEGIN  INTEGER NUM,I;
       I=1;
       BEGIN INTEGER TEMP,NUM;
             TEMP=1;
             NUM=2;
             BEGIN INTEGER NFACT,NUM;
                   NFACT=TEMP;
                   NUM=NFACT
             END
       END
END
```

Fig. 26 Sample Program 2

| [illegible] | [illegible] | I NUM | , | I I | , | I TEMP | , |
|---|---|---|---|---|---|---|---|
| I NFACT | ; | READ | ( | I NUM | ) | ; | I TEMP |
| := | C 1 | ; | I I | := | C 1 | ; | I CONTINUE |
| : | I I | := | I I | + | C 1 | ; | I TEMP |
| := | I TEMP | * | I I | ; | IF | I I | = |
| I NUM | THEN | I NFACT | := | I TEMP | ELSE | GOTO | I CONTINUE |
| ; | WRITE | ( | I NFACT | ) | END | | |

Fig. 22 Symbols and corresponding identifiers or constants in the order they are passed by the scanner to the syntax analyzer for Program 1

TEMP, and NFACT are placed in the hash table and linked together in a chain from the single block head. The block head is formed when the BEGIN symbol is encountered. The variabels are placed in the symbol table when they are encountered in the declaration. CONTINUE is entered in the symbol table as a label when it is encountered. It is linked in the block chain also.

The contents of the symbol table at the end of syntax analysis is given in Fig. 23. It should be compared with Fig. 8 to see what has been entered in the fields of the various entries. Note that the hash table which is locations 1 through 128 contains zeroes except where the identifiers have been linked to buckets. Only one variable or label has been linked to each bucket so all the hash chains consists of only one entry. Since all the variables and labels are declared in the same block they are all linked together through the third location (field two) of each entry. The location of the label CONTINUE in the postfix string PS is stored in the value field of CONTINUE in the symbol table.

At the end of syntax analysis the entire input program has been processed and converted to the postfix strings PS and PS1. The postfix strings produced for program 1 are shown in Fig. 24. Again the symbols and not the internal code are shown in the Fig. 24. The variable names and constants saved in the parallel string PS1 are also shown.

7.1.2 Execution

The postfix string PS and the parallel string PS1 are executed by the interpreter next. As mentioned in the description of the execution part of the interpreter the two parallel stacks FE and FE1 are used to perform execution of the postfix strings PS and PS1. The first thing done during execution is the initialization of the pointers K and K1. For keeping track of

| Location Number | Hash table Locations 1-128 |
|---|---|
| 23 | 140 |
| 64 | 147 |
| 72 | 161 |
| 83 | 154 |
| 106 | 133 |

(All other locations of hash table contain 0)

| Location Number | Block head 1 |
|---|---|
| 129 | |
| 130 | |
| 131 | 161 |
| 132 | 1 |

| Location Number | Contents |
|---|---|
| 133 | NUM |
| 134 | 0 |
| 135 | 0 |
| 136 | 0 |
| 137 | 1 |
| 138 | 0 |
| 139 | |

| Location Number | Contents |
|---|---|
| 140 | I |
| 141 | 0 |
| 142 | 133 |
| 143 | 0 |
| 144 | 1 |
| 145 | 0 |
| 146 | |

| Location Number | Contents |
|---|---|
| 147 | TEMP |
| 148 | 0 |
| 149 | 140 |
| 150 | 0 |
| 151 | 1 |
| 152 | 0 |
| 153 | |

| Location Number | Contents |
|---|---|
| 154 | NFACT |
| 155 | 0 |
| 156 | 147 |
| 157 | 0 |
| 158 | 1 |
| 159 | 0 |
| 160 | |

| Location Number | Contents |
|---|---|
| 161 | CONTINUE |
| 162 | 0 |
| 163 | 154 |
| 164 | 0 |
| 165 | 1 |
| 166 | 1 |
| 167 | 10 |

Fig. 23 The contents of the symbol table at the end of syntax analysis for Program 1

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| PS | BEGIN | I NUM | READ | I TEMP | C 1 | := | I I | C 1 |

| 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|
| := | L CONTINUE | I I | I I | C I | + | := | I TEMP |

| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|
| I TEMP | I I | * | := | I I | I NUM | = | IF |

| 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|
| I NFACT | I TEMP | := | THEN | I CONTINUE | GOTO | ELSE | I NFACT |

| 33 | 34 | 35 |
|---|---|---|
| WRITE | END | HALT |

Fig. 24 Postfix Strings PS and PS1 produced for Program 1

the block structure BLKNO and CRB are initialized. The loop which performs the execution is now entered using the symbol pointed to by K1 to decide which execution routine is to be entered.

The appropriate routine is branched to by the use of a GOTO and a switch (the equivalent of a computed GOTO in FORTRAN) with PS(K1) as the argument.

The first symbol is a BEGIN symbol so the BEGIN routine is entered. At this point the operand stacks FE and FE1 are empty. BLKNO is incremented by one and the first and only block head is found. By proceeding through the block chain the flags of the entries NUM, I, TEMP, CONTINUE, and NFACT are set to one. The value parts (field 6) of the variables are also set to zero while the value part of the label CONTINUE is left unchanged. At the end of the routine the stacks FE and FE1 are still empty. In Fig. 25(1) the states of FE, FE1, and K1 after executing the BEGIN routine are given. Note that only the first three locations of the stacks FE and FE1 are shown since no more than three elements are ever on these stacks during the execution of program 1. Since PS and PS1 are never altered during exeuction only the pointer K1 is given and Fig. 24 should be consulted to see what symbol K1 is pointing to.

The next symbol is I and the I routine causes the address of the identifier NUM stored in PS1 to be placed on stack FE and 1 to be placed on stack FE1. Fig. 25(2) shows the state of the stacks and pointer K1 after the I routine has been executed.

Next a READ symbol is encountered and its routine causes a number to be read off a data card. It is assumed that the number four is on the data card. The number is stored in the value field of the symbol table entry of the variable NUM. The stacks FE and FE1 are then emptied (Fig. 25(3)). The figure specified here and those specified throughout the rest of this section

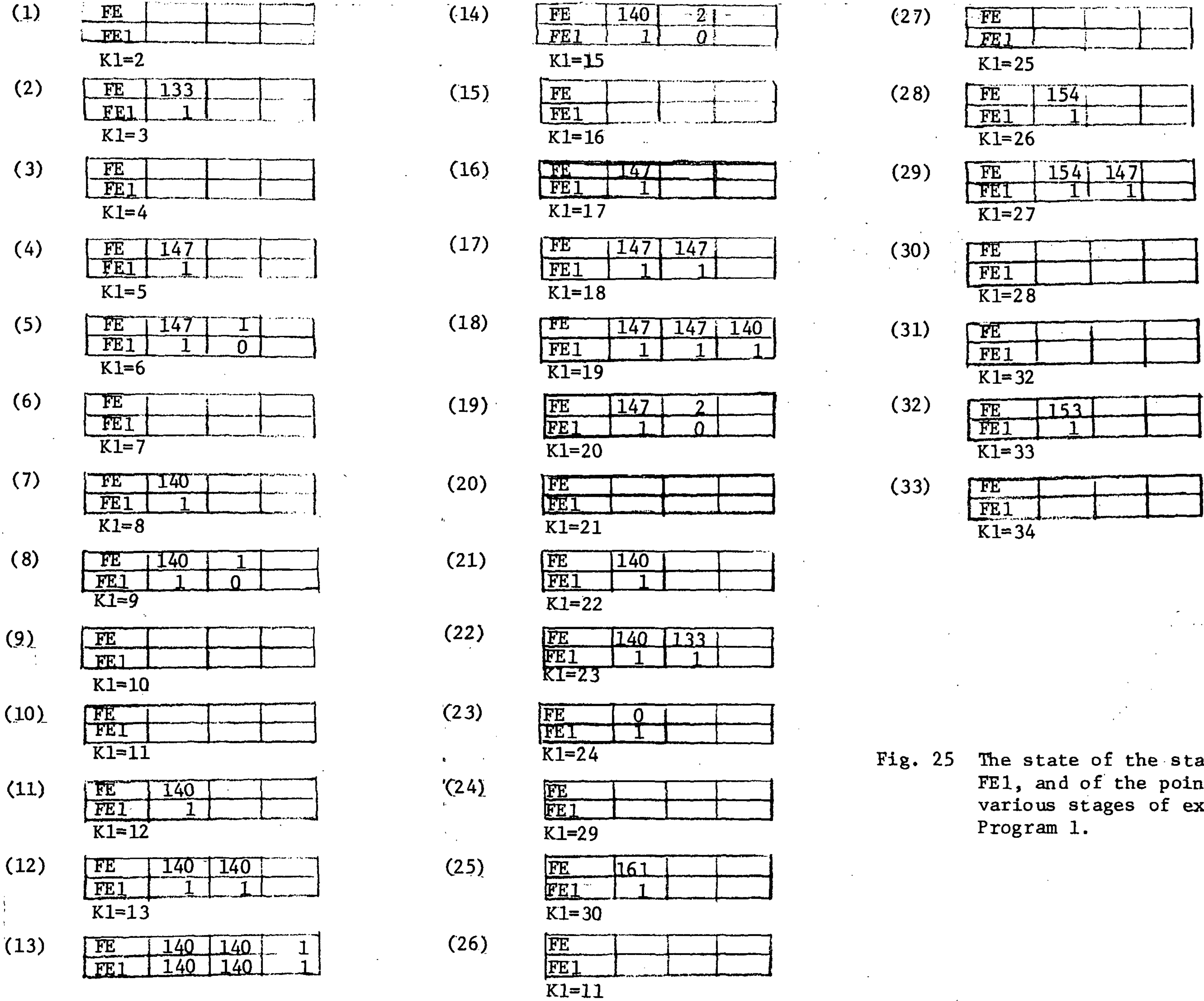



Fig. 25 The state of the stacks FE and FE1, and of the pointer K1 at various stages of execution for Program 1.

give the states of the stacks and of pointer K1 at the end of the routine described.

In Figs. 25(4) and 25(5) it can be seen that the address of the variable TEMP and the value of the constant one have been stored on the operand stack by the appropriate routines. Each figure shows the results of the execution of a routine.

Next a := symbol is encountered and the routine causes the constant one to be stored in the value field of the symbol table entry of the variable TEMP. The stack is then emptied (Fig. 25(6)).

The same process is performed next on the variable I. The constant one is stored finally in the value field of the symbol table entry of I. Figs. 25(7), 25(8), and 25(9) show the result of the three routines executed.

An L symbol is encountered so all that is done by the L routine is to increment K1 so that the next symbol is processed (Fig. 25(10)).

In Figs. 25(11), 25(12), and 25(13), it can be seen that the addresses of the variables I and I are stored on the operand stack and the value of the constant one is stored on the operand stack since the appropriate symbols are encountered.

The symbol + is encountered next. The top operand stack element is value (FE1(K)=0) so it is not changed, but the second stack element is an address (FE1(K-1)=1) so the value of the variable is obtained and replaces the address. The top two stack elements are then added producing the result of 2. This value then replaces the second stack element and the kind field in FE1 is set to zero. The pointer K is decremented so that the result is now on top of the stack (Fig. 25(14)).

A := symbol is encountered and since the top stack element is value that value is stored in the value field of the symbol table entry of the variable I whose address is the second stack element. The stack is again emptied (Fig. 25(15)).

In Figs. 25(16), 25(17), and 25(18) the addresses of the variables TEMP, TEMP, and I are stacked since the symbol I is encountered three times in succession. Then the symbol * is encountered. The values of the two top operands are obtained easily since their addresses in the symbol table are on the stack. The two values are multiplied together and the result, 2, replaces the second stack element. The kind field is set to zero. K is decremented by one so that the result is on top of the stack (Fig. 25(19)).

A := symbol is encountered so the value 2 which is the top operand stack element is stored in the value field of the symbol table entry of the variable TEMP whose address is the second stack element. FE and FE1 are again emptied (Fig. 25(20)).

Two consecutive I symbols cause the addresses of the variables I and NUM to be placed on the operand stack (Figs. 25(21) and 25(22)).

The symbol = causes the values of the top two operands to replace their addresses. The two values are then compared. Since the value are different a zero replaces the second stack element which is promptly made the top stack element by decrementing K by one (Fig. 25(23)). The values of I and NUM have just been compared. The value of I is 2 and of NUM is 4. NUM contains the number the factorial of which program 1 is computing. When I and NUM have the same value then TEMP will contain the desired result. A conditional statement is being processed and the top value of the operand stack now indicates which part of the conditional statement is to be executed.

The next symbol is an IF symbol. Since the top stack element is a zero, the THEN statement of this conditional statement must be skipped. The stacks FE and FE1 are emptied. The IF routine then proceeds to go through PS symbol by symbol looking for the ELSE statement. Since the THEN statement is not a block or compound statement the routine continues incrementing K1 until a THEN or TLS symbol is encountered. When K1 becomes 28 a THEN symbol

is encountered. K1 is incremented by one and the routine ends (Fig. 25(24)).

An I symbol is encountered next so the address of the identifier CONTINUE is placed on the operand stack FE (Fig. 25(25)). Next a GOTO symbol is encountered. The GOTO routine obtains the location in PS of label CONTINUE whose address in the symbol table is the top operand stack element. Since the label is earlier in PS than K1 presently points the routine proceeds to back up by decrementing K1 until K1 reaches the label which is in PS(10). The routine checks for either a BEGIN or END symbol each time before it decrements K1 by one, but no such symbols are encountered. Eventually K1 becomes 10 so K1 is incremented and the routine ends with K1 equal to 11 and the stack empty (Fig. 25(26)).

Execution continues but it is processing the same symbols again with new values for some of the variables. Eventually the conditional statement is reached again and since I is three this time it will still not be equal to NUM so the GOTO will be reached and the loop will be repeated again. TEMP will have value 6 after the second pass through the loop. The third time the conditional statement is reached I will have value 4 so it will be equal to NUM and a different part of the IF routine is entered. All this part does is empty the operand stack which had a 1 (true) on it and increment K1 by one so that the THEN statement is executed instead of skipped. (Fig. 25(27)).

Two consecutive I symbols are encountered so the address of the variables, NFACT and TEMP, are placed on the operand stack (Figs. 25(28) and 24(29)). A := symbol is encountered so the value of the top operand is obtained and replaces the address on top of the stack. This value is the value of TEMP which is 24 after the last time through the loop. It is the desired result i.e., NUM!. The value is then stored in the value field of the symbol table entry of NFACT whose address is the second stack element. The stack is emptied and K1 is incremented by one to 29 (Fig. 25(30)).

A THEN symbol is encountered which means an ELSE statement is next and must be skipped. This is done by incrementing K1 until the ELSE symbol is reached. The THEN routine must check for a block or compound statement but since none is found all that need be done is increment K1 until the ELSE symbol is reached. K1 is incremented by one so that the first symbol after the ELSE symbol will be the next to be executed (Fig. 25(31)).

An I symbol is encountered so the address of the variable NFACT is stacked (Fig. 25(32)). A WRITE symbol is encountered next so the value, 24, of the variable NFACT is written out. The stack is emptied one last time (Fig. 25(33)).

An END symbol is next so the END routine proceeds to deactivate the variables and labels declared in this block. Finally the HALT symbol is reached. The execution and the interpretation of the program are ended.

### 7.2 Program 2

Fig. 26 shows program 2. The program does not compute anything of interest, but it is given as a simple example of a program with a more complicated symbol table than the last example. Fig. 27 shows the symbols in the order they are passed by the scanner to the syntax analyzer. It also shows the identifiers and constants which are passed along with the symbols I and C. In Fig. 28 the output of the syntax analysis is shown. The execution of this program is even simpler than that for the last program except that there are more BEGIN and END symbols in order to activate or deactivate identifier at the proper times. The execution of program 2 is not described, but Fig. 29 should be consulted since the symbol table for program 2 is given there.

For program 2 there are three block heads. Each block head has a chain of the identifiers declared in that block. This chain is utilized during the execution to activate all the identifiers for that block when it is

| BEGIN | INTEGER | I<br>NUM | , | I<br>I | ; | I<br>I | C<br>1 |
|---|---|---|---|---|---|---|---|
| ; | BEGIN | INTEGER | I<br>TEMP | , | I<br>NUM | ; | I<br>TEMP |
| = | C<br>1 | ; | I<br>NUM | = | C<br>2 | ; | BEGIN |
| INTEGER | I<br>NFACT | , | I<br>NUM | ; | I<br>NFACT | = | I<br>TEMP |
| ; | I<br>NUM | = | I<br>NFACT | END | END | END | |

Fig. 27 Symbols in the order they are passed by the scanner to the syntax analyzer for program 2

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PS | BEGIN | I | C | = | BEGIN | I | C |
| PS1 | | I | 1 | | | TEMP | 1 |

| | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|
| PS | = | I | C | = | BEGIN | I | I |
| PS1 | | NUM | 2 | | | NFACT | TEMP |

| | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|
| PS | = | I | I | = | END | END | END |
| PS1 | | NUM | NFACT | | | | |

Fig. 28 Postfix strings PS and PS1 produced for Program 2

| Location Number | Hash table Locations 1-128 |
|---|---|
| 23 | 140 |
| 64 | 151 |
| 83 | 169 |
| 106 | 176 |

(All other locations of hash table contain 0)

Block head 1

| Location Number | Contents |
|---|---|
| 129 | |
| 130 | 147 |
| 131 | 140 |
| 132 | 1 |

| Location Number | Contents |
|---|---|
| 133 | NUM |
| 134 | 0 |
| 135 | 0 |
| 136 | 0 |
| 137 | 1 |
| 138 | 0 |
| 139 | |

| Location Number | Contents |
|---|---|
| 140 | I |
| 141 | 0 |
| 142 | 133 |
| 143 | 0 |
| 144 | 1 |
| 145 | 0 |
| 146 | |

Block head 2

| Location Number | Contents |
|---|---|
| 147 | 129 |
| 148 | 165 |
| 149 | 158 |
| 150 | 2 |

| Location Number | Contents |
|---|---|
| 151 | TEMP |
| 152 | 0 |
| 153 | 0 |
| 154 | 0 |
| 155 | 2 |
| 156 | 0 |
| 157 | |

| Location Number | Contents |
|---|---|
| 158 | NUM |
| 159 | 133 |
| 160 | 151 |
| 161 | 0 |
| 162 | 2 |
| 163 | 0 |
| 164 | |

Block head 3

| Location Number | Contents |
|---|---|
| 165 | 147 |
| 166 | |
| 167 | 176 |
| 168 | 3 |

| Location Number | Contents |
|---|---|
| 169 | NFACT |
| 170 | 0 |
| 171 | 0 |
| 172 | 0 |
| 173 | 3 |
| 174 | 0 |
| 175 | |

| Location Number | Contents |
|---|---|
| 176 | NUM |
| 177 | 158 |
| 178 | 169 |
| 179 | 0 |
| 180 | 3 |
| 181 | 0 |
| 182 | |

Fig. 29 State of the symbol table at the end of syntax analysis for Program 2

entered and to deactivate them when it is left.

The bucket in the hash table at location 106 has a chain with more than one entry. These entries happen to be the same identifier which has been redeclared several times in different blocks and there is an entry for each declaration. Note that these entries are kept in order according to block numbers in decreasing order. The linking together of block heads can be seen in Fig. 29.

## 8. Acknowledgment

The author wishes to express his thanks to Professor Yaohan Chu for his guidance in the architectural design and his assistance in preparing the manuscript, to Leonard S. Haynes for his assistance in syntax analysis, and to Nancy Nowell for her typing.

## 9. References

1. P. Baumann, et al., "Introduction to Algol", Prentice-Hall, Inc., 1964.
2. H. Bloom, "Design and Simulation of an Algol Computer", Tech. Report 70-118, Computer Science Center, University of Maryland, June, 1970.
3. A. Evans, "An Algol 60 Compiler", Annual Review in Automatic Programming 4, Pergamon Press, 1964, pp. 87-124.
4. D. Gries, "Compiler Construction for Digital Computers", John Wiley & Sons, Inc., 1971.
5. J. Icchbiah and S. P. Morse, "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars", Comm. of the ACM, August, 1970.
6. P. Naur, Editor, "Revised Report on the Algorithmic Language Algol 60", Comm. of the ACM, January 1963.
7. T. Signiski, "Design of an Algol Machine", Tech. Report 70-131, Computer Science Center, University of Maryland, September, 1970.
8. Processor and STorage Programmers Reference Manual for Univac 1108, Document No. UP-4053, Rev. 1, Sperry Rand Corporation, 1966, 1970.

## Appendix A, BNF Description of an Algol Subset

1. <I>::=A|...|Z|<I>{A|...|Z}|<I>{0|...|9}
2. <UN>::=0|...|9|<UN>{0|...|9}
3. <V>::=<I>
4. <L>::=<I>
5. <P>::=<UN>|<V>|(<AE>)
6. <F>::=<P>|<F>↑<P>
7. <T>::=<F>|<T>{X|/}<F>
8. <AE>::=<T>|{+|-}<T>|<AE>{+|-}<T>
9. <BE>::=<AE>{=|≠}<AE>
10. <AS>::=<V>:=<AE>
11. <GTS>::=goto<L>
12. <RS>::=read(<V>)
13. <WS>::=write(<V>)
14. <TL>::=<V>|<V>,<TL>
15. <D>::=integer<TL>
16. <CS>::=if<BE>then<US>{Λ|else<US>}|<L>:<CS>
17. <BS>::=<AS>|<GTS>|<RS>|<WS>|<L>:<BS>
18. <US>::=<BS>|<CPS>|<B>
19. <S>::=<US>|<CS>
20. <CT>::=<S>|<CT>;<S>
21. <CPS>::=begin<CT>end|<L>:<CPS>
22. <B>::=begin<D>;<CT>end|<L>:<B>
23. <PR>::=<B>|<CPS>