To the Graduate Council:

I am submitting herewith a thesis written by Karen S. Gregg entitled "An Interpretive System for the Programmed Inquiry, Learning or Teaching (PILOT) Language." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Charles E. Hughes
Charles E. Hughes
Major Professor

We have read this thesis and recommend its acceptance:

Ronald P. Leinius

James R. Pinkert

Accepted for the Council:

Hilton A. Smith
Vice Chancellor
Graduate Studies and Research

AN INTERPRETIVE SYSTEM FOR THE PROGRAMMED INQUIRY, LEARNING OR TEACHING (PILOT) LANGUAGE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee

Karen S. Gregg

June 1975

1220518

THIS THESIS IS DEDICATED

TO MY PARENTS

whose love, encouragement, and understanding

are the real treasures

of my life

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Charles Hughes for his patient guidance in the preparation of this thesis and to Dr. Ronald Leinius and Dr. James Pinkert for their services on the Master's degree committee.

Special thanks go to Mr. Tim MacKenzie and Mr. Joe Morris for their suggestions and assistance.

ABSTRACT

The purpose of this thesis is to develop an efficient interpretive system for the Programmed Inquiry, Learning Or Teaching (PILOT) programming language. PILOT is a programming system for controlling interactive conversation in which little or no restriction is imposed on the replies of the user. YORK APL was used as the host language; use of most of its features is permitted in this version of PILOT.

A short description of the language is given in Chapter II. The actual implementation is explained in Chapter III. Chapter IV is devoted to a discussion of the MATCH function, which is at the heart of PILOT. As this instruction is used quite frequently in PILOT programs, a fast, efficient algorithm should be utilized. Several pattern-matching algorithms are examined in order to determine which provides the most efficient performance, primarily in terms of time required. Copies of the program listings, flowcharts, and sample PILOT programs are included in the Appendices.

TABLE OF CONTENTS

CHAPTER		PAGE
I.	INTRODUCTION	1
II.	BACKGROUND	4
III.	SYSTEM DEVELOPMENT	12
IV.	THE MATCH STATEMENT	39
V.	SUMMARY AND CONCLUSIONS	64
BIBLIOGRAPHY		67
APPENDICES		69
Appendix A		70
Appendix B		84
Appendix C		111
VITA		115

LIST OF FIGURES

FIGURE		PAGE
1.	General System Design of the Interpretive System	13
2.	Flowchart for Simple Sequential Table Search	19
3.	Flowchart of Modified Table Search Algorithm	22
4.	Flowchart of Simple Sequential Search	41
5.	Finite State Diagram of Failure-Link Array for Pattern = "ABCABCAB"	48
6.	Flowchart for LEX	71
7.	Flowchart for Subroutine LUKUP	76
8.	Flowchart for Subroutine IN	77
9.	Flowchart of Subroutine MATCH	79
10.	Flowchart for Subroutine ACCEPT	81
11.	Flowchart for Subroutine CHEK	82
12.	Flowchart for Subroutine BLANK	83

CHAPTER I

INTRODUCTION

Computer-aided instruction (CAI) is an educational concept in which a student is placed in a conversational mode via a terminal to a computer which executes a programmed course, or study plan. This technique permits each student to progress at an individual pace since each of these programmed courses may be designed to choose the next topic of study based on his previous responses.

General-purpose programming languages such as PL/1 have been used to implement CAI systems, however, much time and effort are usually required to write a program for a course of substantial length. Usage of such languages also requires either that the teacher become familiar with the language or that a programmer act as an interface between the teacher and the language. A simple means of avoiding this problem is to simplify the computer language and provide easy access to the computer system. In this way, both student and teacher are able to concentrate on the content of the course, rather than its construction.

Since most instruction is performed through utilization of concepts that are expressed in English sentences, a CAI language must be geared primarily toward efficient text manipulation. One language which suits this requirement is

Programmed Inquiry, Learning Or Teaching (PILOT), which is designed to support conversational programs in which minimal restriction is placed on student responses. It is simple enough that a person who has no previous experience with computers may begin to write interesting programs almost immediately, but very complicated programs may also be written.

It is the purpose of this thesis to design and implement an efficient interpretive system for the PILOT language using YORK APL as the host language. In making the selection of the host language, the suitable timesharing languages available at the University of Tennessee Computing Center—CALL-OS, PL/1, BASIC, and APL—were considered. YORK APL was particularly appealing because of its Error Trap and Unquote features, which have been used heavily in the implementation. Another attractive feature was the use of the I-beam functions which facilitate timing the search routines and provide convenient access to information concerning the amount of space remaining in the library and the active workspace.

In Chapter II, a brief background of the development of the PILOT language is provided. A description of the interpretive system is contained in Chapter III. Chapter IV contains a detailed explanation of the MATCH function, which is at the core of the PILOT language. A summary and

conclusions are presented in Chapter V. The appendices include flowcharts, program listings, and sample PILOT programs.

CHAPTER II

BACKGROUND

In response to the need for a minimal computer language for instructional usage, John Starkweather (13) of the University of California Medical Center developed the PILOT language in 1969. One of the attractive features of this language is its simple configuration. PILOT is designed to deal with conversational programs which impose minimal or no restriction on the replies of the student. Usually in ordinary conversation, only a few words in a sentence convey most of its meaning. In a similar manner, PILOT is capable of searching a response for substrings consisting of words, portions of words, or combinations thereof as indicated by the PILOT programmer, or the author. This capability allows the author to accept conversational replies to questions and to build an array of word components to be used in the recognition of the responses. Rubin (10) pointed out that one important advantage of such a feature is that, when used properly, it can create a sense of individualization for the student. This affects the attitude of the student toward the computer by demonstrating that the computer actually does acknowledge his presence.

Starkweather's version of PILOT was written in assembly language for an SDS-940 computer. In 1971, Rubin (9) of the Stanford Research Institute replaced this version with a smaller one in the form of an interpreter written in BASIC. PYLON, as this version was called, attracted interest from experimenters in CAI, especially since it was easily adaptable to other systems. By late 1972, Stanford University, the Northern California Regional Computer Network, consisting of approximately eighteen colleges, and Lawrence Hall of Science at the University of California at Berkeley had developed versions of PYLON for their use on a minicomputer with timesharing capability. In 1973, Starkweather (12) developed a smaller version for use by the University of California Medical Center. It was executed on a Datapoint 2200, with a 12K byte memory and no central computer connection requirement.

An increasing need for standardization was satisfied in January 1973, when a group of users of the various versions of PILOT met and agreed on specifications for a standard "core" PILOT, called PILOT 73. In this document, all references to PILOT will actually refer to PILOT 73.

A set of one-character instructions is used by PILOT. Any extensions or modifications to PILOT should consist of two or more characters in order that new or modified instructions may be added to a version without destruction

of the interchangeability of the core language. PILOT does not have any pre-defined structure; blanks may be used as desired in most instances to improve readability.

PILOT statements are of the general form:

{label} instruction {condition} : object

or, for a continuation statement,

: object

The label is optional and consists of an asterisk followed by a maximum of eight contiguous YORK APL characters. If a label appears, then the asterisk is required to be the first nonblank character in the statement. All references to the label must include the asterisk.

The valid PILOT instructions are T, R, J, A, M, Y, N, C, E, and U. In addition, this implementation includes an extension to the MATCH (M) instruction, the MATCH VARIABLE (MV) instruction. They are described later in this chapter.

The optional conditional may consist of a single letter "Y" for "yes" or "N" for "no," or it may consist of a logical expression which is enclosed in parentheses, e.g., (3=2). The action taken in the latter case resembles that of a logical "IF." For the case of a conditional of "Y," the results of the last MATCH (M) statement are examined. If there was a match, the match-key equals one ("yes"), and the current statement is then executed. Otherwise, execution

continues with the next statement. Exactly the reverse is true for a conditional of "N."

Any PILOT instruction may be made conditional. In fact, a shorthand representation of TY (TYPE if YES) exists in the form of the YES (Y) instruction. Similarly, TN (TYPE if NO) may alternately be written as the NO (N) instruction.

A colon is required for all PILOT statements regardless of whether or not an object is present.

The presence and form of the object are dependent on the type of statement. It may consist of a label, a text list, text, a variable, a program name, or it may not be present at all (e.g., the E: statement).

Two forms of continuation of PILOT statements are available. If the first nonblank character of a statement is a colon, the previous line is checked to see if it contained a colon. If it did not, the object of the current statement is merely catenated onto the previous statement. However, if a colon was present in the preceding line, its instruction code and conditional also become those of the current line. Note that the latter case is technically not a continuation.

Variable names in PILOT may be from one to eight characters in length. They are recognized by a special character prefix—a "$" for character and a "#" for numeric variable names. When a variable name is encountered in an

ANSWER statement, its name is entered in the label table and the pointer to its corresponding location in either the numeric or character variable value table, dependent upon the type of variable name, is placed in the parsed statement table. When the statement is executed, the input value is placed in the position indicated by the pointer. Note that two distinct tables for values are necessary since YORK APL does not facilitate mixing of numeric with literal values in an array. This pointer is also used if a variable name appears in a TYPE, YES, NO, or COMPUTE statement so that the variable may be defined and used during the interpretive phase.

During the initialization of the parsing phase, all numeric variable values are set to zero and all character variable values are cleared with blanks. If a variable is not defined by the program prior to its use, its initialization value is used. This action is a standard procedure in PILOT. In this implementation, however, the user will be issued a warning message for any variables which are undefined.

As an introduction to the PILOT language, let us first consider the four basic instructions—T, A, J, and M. Most PILOT programs may be written using exclusively these instructions. The TYPE (T) instruction causes the object of

the statement to be printed. A blank line is printed if no object is present.

The ANSWER or ACCEPT (A) statement causes the terminal to type a question mark (?), then to wait for input. If the object contains a variable name, the information which is input is placed at that location. Otherwise, the information is stored in an area reserved for the most recent user response for potential usage in later statements. If the object of a TYPE statement contains variables which have been defined in a previous answer statement, these variable names are replaced by their respective values when the text is printed.

The JUMP (J) statement results in a transfer of execution to the statement whose label is the same as the object of that JUMP statement. In all cases, it is imperative that the asterisk be included as the first character of the label.

Recognition of user responses is made possible through the use of the MATCH (M) statement. It performs a search of the most recent response for specific character strings that comprise the object. Each of the strings, which are separated by commas, is taken sequentially and compared with the most recent response. If at least one of the strings is contained in the response, the match-key is set to a value of one ("yes"). Conversely, if none are elements of the

response, the match-key is set to zero ("no"). Each time a MATCH statement is executed, the match-key is reset. If the last response was stored in a variable, the routine is designed to copy this variable value into the area reserved for the response to the most recent ANSWER statement. Numeric variables are also copied, but the copy is converted into a literal string prior to the comparison. Chapter V contains a detailed explanation of the matching function.

The MATCH VARIABLE (MV) statement is very similar to the MATCH, except that a previously defined variable name is specified as the first element of the object. The value of this variable is copied into the location reserved for the most recent user response. The elements of the object following the variable name are then compared in the usual manner to the new contents of that location.

Remarks are permitted through the use of the REMARK (R) statement. This statement is ignored by the interpreter and thus provides a means of program documentation.

Calculation of any valid YORK APL expression may be accomplished by the use of the COMPUTE (C) instruction. This statement provides an alternate method of definition of variables.

Subroutines may be accessed through the USE (U) statement. This instruction causes a temporary transfer of control to a subroutine, where the subroutine name is the

object. At the end of the subroutine, execution returns to the statement immediately following the U-statement.

A PILOT subroutine begins with a labeled statement and must be terminated by an END statement. It must be included with the main program following the U-statement that refers that particular subroutine. Since it refers to a label in the input, a star must be the first nonblank character in the object. YORK APL programs may also be specified by means of the object, but the first nonblank character in this case must not be a star. Details concerning the operation of the USE statement are discussed in Chapter III.

The END (E) statement causes the termination of the subroutine, or termination of program execution if the program is at zero level. Since control automatically returns to the statement following a U-statement, no object is required. The object field of the E-statement is not examined by the interpreter; it may be used for comments.

CHAPTER III

SYSTEM DEVELOPMENT

As was stated previously, the purpose of this thesis is to design and implement an efficient interpretive system for the PILOT programming language at the University of Tennessee Computing Center using YORK APL as the host language. Approximately 70K bytes of library space and 32K bytes of workspace are available, so efficiency is of the utmost importance.

It will be necessary to use the interpretive approach since user interaction with the system will be required. In general, an interpreter executes somewhat slower than a compiler since it converts source statements into an intermediate form which is executed instead of converting code into machine language as is the case of a compiler. This implementation does not use a pure interpreter; whenever possible it makes use of several features of YORK APL in an attempt to optimize resources. One example is the use of the UNQUOTE function, which is the subject of a later discussion.

The general system design in using the interpretive approach is shown in Figure 1. By observation we see that the system may be easily divided into two phases. The first involves accepting a PILOT source statement from the user,

checking it for errors, and converting it to the intermediate form. The second step consists of processing the statements which are expressed in the intermediate form.

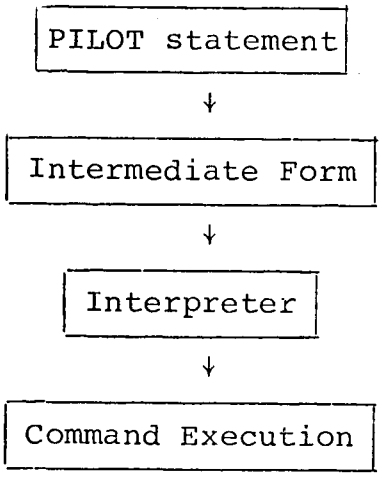


Figure 1. General System Design of the Interpretive System

Separation of the two phases facilitates the use of planned overlay in which only those routines necessary for the parsing phase will be loaded into the active workspace. During the interpretive phase any routines that are no longer needed will be removed from working storage, making room for the routines needed for this phase to be loaded. This is accomplished using the UNQUOTE function with the quoted APL expression. For instance, execution of the APL expression ⍎ ')*LOAD ABC*' results in the program named ABC being loaded into the active workspace. A subroutine called

UNQUOTE has been defined to perform this action of removing quotes from a character string and executing it.

We will refer to the first of the two phases as the parsing phase, and the second phase will be called the interpretive phase. These are distinct stages in the interpretation process. The intermediate forms of the statements will be stored in a table called the parsed statement table. It is this table that serves as an interface for the two phases. Auxiliary tables which contain variable and label names and their values will also be accessible from both phases in order to complete the interface.

In the interest of efficiency, statements should be parsed in a way that is most suitable for interpretation. Most importantly, the op code should be placed at the beginning of the parsed statement. The reason is that although the system only parses each statement once, the parsed statement may be interpreted many times in the execution of the program. The general idea is that as much of the workload as possible should be transferred to the parsing phase in order to increase total system efficiency.

Since each PILOT statement is input as a character string, a means of determining the statement type and extracting the language components must be found in order to begin the parsing phase. Then these independent segments

must be examined in order to determine how they are linked and whether the statement is indeed valid. This syntactic analysis, or parsing, may be accomplished through the use of either a generalized parsing routine or by a specific analyzer called a specialized sieve (7). One advantage to the use of a general routine is that if a definition of a language is input, the same analyzer can be used for several languages. Although this approach is generally useful, it is inefficient when compared to the use of an analyzer specially designed for a specific language. Ordinarily, when a statement is parsed, the entire string must be scanned in order to determine the statement type and to check for validity without regard to redundancy. However, if only the op code, the first nonblank character is any unlabeled PILOT statement, need be examined to determine the statement type, the error detection could occur as a spinoff of the process of extracting the components of the statement. A specialized sieve designed in this manner would speed up the process by branching directly to the routine corresponding to the statement type for each sentence. We will now design a specialized sieve for the PILOT language.

As was mentioned in Chapter II, the general form of PILOT statements is:

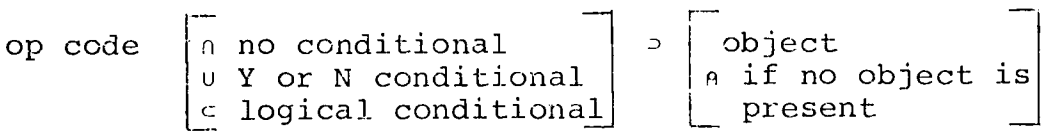
{label} instruction {condition} : object

or, for a continuation statement:

: object

It should be evident at this point that examination of the first nonblank character reveals the type of the statement, provided that we consider continuation statements and labeled statements as special cases. Continuation statements may be contoured to fit this generalized form if we consider the colon to also be the operation code, with no conditional permitted. For the case of labeled statements, after the label has been resolved we may treat the remaining portion of the statement as the general case. Thus with these slight liberties we now have a general form to which every valid PILOT statement must conform. It should also be observed that in this general form, all statements are coded in such a way that the statement type is specified as a unique operation code which is available to the parsing phase immediately. Similarly, this op code may then be placed in the first position of the corresponding parsed statement.

The general form of the parsed statement is:



For example, the program

```
*TY T: THIS IS A PILOT STATEMENT.
T: THIS IS ANOTHER.
J: *TY
E:
```

would be entered into the parsed statement table as follows:

```
T∩⊃' THIS IS A PILOT STATEMENT.'
T∩⊃' THIS IS ANOTHER.'
J∩⊃1
E∩⊃A
```

In this form, the statement type is readily available to the interpretive phase in the form of a unique op code. Thus, we have adhered to one means of attaining greater efficiency in the system.

The type of conditional is specified by a prefix symbol which is either a "∩" (no conditional), "∪" (Y or N conditional), or "⊂" (logical conditional). In this manner only the character following the op code need be examined in order to branch to the proper conditional handling routine.

If the parsing phase detects a blank object, it causes the symbol "⍀" to be placed in the first position in the object field of the parsed statement. In this way, the interpreter need only test the first character in the object field of a source statement since the assumption will be that the field was blank. This restriction should not cause problems for the PILOT user since this character must not occur in valid statements. It is the character that is used to indicate a remark or comment in the YORK APL language.

Since PILOT statements may contain an optional label, a table lookup routine will be necessary. A number of algorithms exist for performing this task. As the length of any PILOT program will be restricted to a maximum of 50

statements, and noting that in most cases relatively few of these statements will contain a label, it becomes evident that an efficient, yet simple program is desired. Little progress would be made if we employed a complex table lookup routine that consumed an excessive amount of our limited storage and workspace areas only to find that a simpler method would have been superior in terms of both storage and time requirements.

The first algorithm (6) which could be used for the task is very straightforward. It involves a simple sequential search in which each entry in the table is compared to the element for which we are searching, in this case, the label name. If the element is contained in the table, a note of its location is made. Otherwise, if the table is exhausted, it is obvious that the element was not contained in the table. This algorithm (6) may be written as:

Step 1.	[Initialize]	k←1
Step 2.	[Compare]	If A = $B_{row\ k}$ the algorithm terminates successfully.
Step 3.	[Advance]	k←k+1
Step 4.	[End of file?]	If k is less than or equal to the number of rows in table B go to 2. Otherwise the algorithm terminates unsuccessfully.

Figure 2 depicts the flowchart to perform this algorithm. It may be coded into the following APL program:

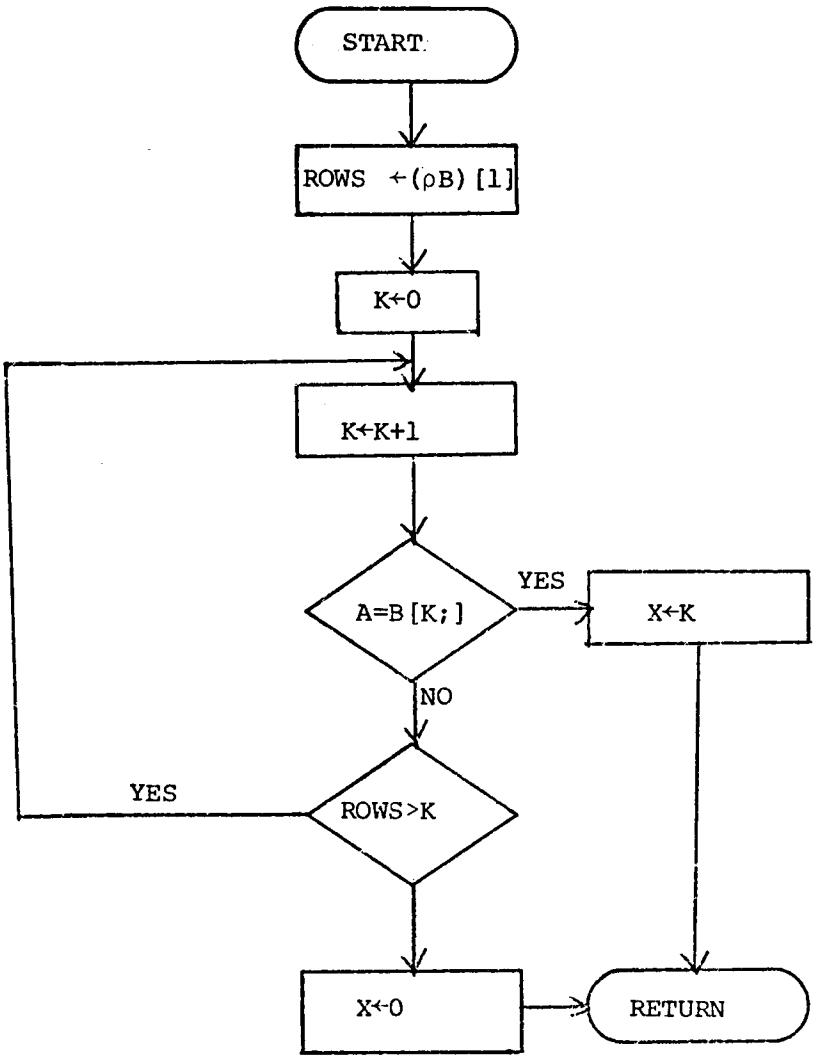


Figure 2

Flowchart for Simple Sequential Table Search

```
∇ X←A TABSER B; XXX; ROWS; QQ; K
 [1.1] XXX←TIMEON
 [2] ROWS←(ρB)[1]
 [2.5] QQ←ρB
 [2.6] A←A,(QQ[2]-ρA)ρ' '
 [3] K←0
 [4] LOOP: K←K+1
 [5] →FOUND×ι∧/A=B[K;]
 [6] →(ROWS>K)/LOOP
 [7] X←0
 [7.1] 'TIME '; TIMEON-XXX
 [7.3] →0
 [8] FOUND: X←K
 [8.2] 'TIME '; TIMEON-XXX
∇
```

Notice that we have utilized the TIMEON function to give a rough idea of its execution speed. This routine will be described in more detail in Chapter IV.

A few minor modifications could be made in the program TABSER in order to cause a substantial increase in speed. In this program, each time the label name A is found to be unequal to row k of table B, a second test must be performed to see if k exceeds the number of rows in B. By reducing the processing of two decisions to one, the program will operate more efficiently. This could be accomplished by concatenating the label name to the end of the label table. In this manner, the second test to determine if the number of rows in table B has been exceeded by the value of the pointer variable k should occur only when a table entry is the same as the label name A. This algorithm (6) may be stated in the following manner:

Step 1. [Initialize] $k \leftarrow 1$ and $B_{row\ N+1} \leftarrow A$

Step 2. [Compare] If $A = B_{row\ k}$ go to 4

Step 3. [Advance] $k \leftarrow k+1$ go to 2

Step 4. [End of file?] If k is less than or equal to the number of rows in table B the algorithm terminates successfully; otherwise it terminates unsuccessfully.

Figure 3 shows a flowchart of this algorithm. Careful observation yields that much of the processing time required will be spent on incrementing and testing k. By incrementing the variable k by two, then comparing A to row k of B, then row k+1 if necessary, an additional savings in time can be realized. This algorithm (6) could be expressed as:

Step 1. [Initialize] $k \leftarrow -1$ and $B_{row\ N+1} \leftarrow A$

Step 2. [Advance] $k \leftarrow k+2$

Step 3. [Compare] If $A = B_{row\ k}$ go to 5

Step 4. [Compare] If $A \neq B_{row\ k+1}$ go to 2

Step 5. [End of file?] If k is less than or equal to the number of rows in table B the algorithm terminates successfully; otherwise it terminates unsuccessfully.

The flowchart corresponding to this algorithm is contained in Appendix A. We may now write the following YORK APL program to perform these actions:

```
∇ R←A LOOKUP B; XXX; I; ROWS; QQ.
[1] ⍝ TABLE SEARCH BASED ON KNUTH, V. 3, P. 395
[1.1] XXX←TIMEON
[2] I←¯1
[3] ROWS←(⍴B)[1]
[4] QQ←⍴B
```

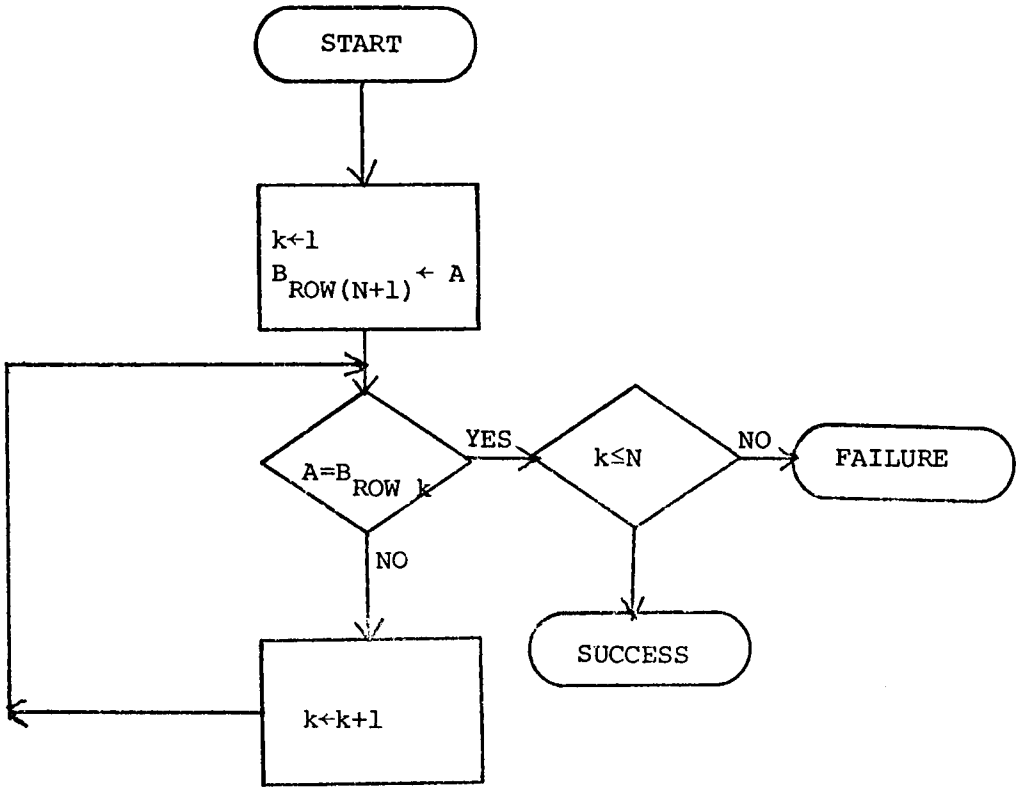


Figure 3

Flowchart of Modified Table Search Algorithm

```
[5] A←A,(QQ[2]-ρA)ρ' '
[6] B←B,[1] A
[7] Q3: I←I+2
[8] →Q4×ι∧/A=B[I;]
[9] →Q3×ι∧/A≠B[I+1;]
[10] I←I+1
[11] Q4: →(ROWS≥I)/OUT
[12] R←0
[12.2] 'TIME WAS '; TIMEON-XXX
[13] →0
[14] OUT: R←I
[14.4] 'TIME WAS '; TIMEON-XXX
∇
```

Since the TIMEON function was also incorporated into this program, we may run tests to determine how much more efficiently the latter algorithm performs in relation to the first. Note that the intermediate algorithm was not programmed since we immediately discovered improvements which led to the third algorithm. Table 1 contains the average times required to execute the two routines TABSER and LOOKUP. Sample A involved searching a table comprised of

AAAA
BBBB
CCCC
DDDD
EEEE
FFFF
GGGG
AAAA
BBBB
KKKK

for the string "AAAC." Sample B involved searching the same table for the string "KKKK." In Sample C the table

```
TREE
MIST
ALPH
BETA
FIVE
SIXE
OCTA
ATEE
NINE
TENN
```

was searched for any occurrence of "ALDA." Similar results were obtained by using longer tables.

Table 1. Average Execution Times of Table Search Routines Based on Samples Taken at Various Times During Operating Hours.

SAMPLE	Program (Execution Time) TABSER	LOOKUP	LUKUP	L4	Improvement by Using LUKUP instead of TABSER
A	11.2	9.6	7.6	7.0	35.5%
B	10.6	9.3	7.1	6.5	38.7
C	14.8	12.4	10.2	9.8	33.8

Improvement at each Program Change

SAMPLE	TABSER-LOOKUP	LOOKUP-LUKUP	LUKUP-L4
A	14%	21%	8%
B	12	24	8
C	16	17	4

Now let us compute the expected improvement by using LOOKUP instead of TABSER. If we ignore statements that deal with the TIMEON function, we may compute the overall number of steps that are executed in TABSER:

```
∇X←A TABSER B;ROWS;QQ;K
 [1]  ROWS←(ρB)[1]                    1
 [2]  QQ←ρB                           1
 [3]  A←A,(QQ[2]-ρA)ρ' '              1
 [4]  K←0                             1
 [5]  LOOP: K←K+1                     C
 [6]  →FOUND×ι∧/A=B[K;]               C
 [7]  →(ROWS>K)/LOOP                  C-S
 [8]  X←0                             1-S
 [9]  →0                              1
 [10]  FOUND: X←K                     S
∇
```

By adding the number of steps indicated to the right of each statement, we have 3C-S+6 steps required, where C is the number of comparisons and S is 1 for success and 0 for failure in the result of the search.

In a similar manner we may compute the overall number of steps required to execute LOOKUP as follows:

```
∇R←A LOOKUP B;I;ROWS;QQ
 [1]  I←-1                            1
 [2]  ROWS←(ρB)[1]                    1
 [3]  QQ←ρB                           1
 [4]  A←A,(QQ[2]-ρA)ρ' '              1
 [5]  B←B,[1]A                        1
 [6]  Q3: I←I+2                       (C-S+2)/2
 [7]  →Q4×ι∧/A=B[I;]                  (C-S+2)/2
 [8]  →Q3×ι∧/A≠B[I+1;]                (C-S+2)/2
 [9]  I←I+1                           (C-S)MOD 2
 [10]  Q4: →(ROWS≥I)/OUT              1
 [11]  R←0                            1-S
 [12]  →0                             1
 [13]  OUT: R←I                       S
∇
```

Once again we total the number of steps indicated on the right to get 1.5C-1.5S+11+(C-S) mod 2. This represents an expected reduction of approximately 50 percent of the number

of overall steps that must be executed by using LOOKUP instead of TABSER. It is important to note, however, that the number of difficult steps remains the same for both routines; it is only the overall reduction that is shown here.

A noticeable discrepency exists between our expected improvement of 50 percent and the experimental results of approximately 14 percent. This difference is primarily due to the different amounts of time required to execute the various APL primitives and operations. Note that in using LOOKUP we have reduced the number of times the incrementation statement is encountered by burying an incrementation in line 8. This causes a decrease in execution time required since APL must perform certain housekeeping routines, e.g., updating the I-beam functions, before executing the next statement. Performing these actions requires a finite amount of time for each statement, so decreasing the number of statements to be executed actually results in decreasing the execution time. With this in mind, let us now try incrementing by three in the following program:

```
∇ R←A LUKUP B;I;ROWS;QQ;XXXX
 [.5]  XXXX←⌶21
 [1]   I←-1
 [2]   ROWS←(⍴B)[1]
 [3]   QQ←⍴B
 [4]   A←A;(QQ[2]-⍴A)⍴' '
 [5]   B←B,[1]A
 [6]   Q3:  I←I+3
 [7]   →Q4×⍳∧/A=B[I;]
 [8]   →Q4×⍳∧/A=B[I+1;]
```

```
[9]   →Q3×ι∧/A≠B[I+2;]
[10]  I←I+2
[11]  Q4: →(ROWS≥I)/OUT
[11.1]  'TIME ';(⌶21)-XXXX
[12]  R←0
[13]  →0
[14]  OUT: R←I
[15]  'TIME ';(⌶21)-XXXX
∇
```

Resulting execution times are shown in Table 1. In our experiments, the average improvement by using LUKUP instead of LOOKUP is approximately 20.7 percent. Now, let us try incrementing by four:

```
∇ R←A L4 B;I;ROWS;QQ;XXXX
 [.5]  XXXX←⌶21
 [1]   I←-1
 [2]   ROWS←(ρB)[1]
 [3]   QQ←ρB
 [4]   A←A;(QQ[2]-ρA)ρ' '
 [5]   B←B,[1]A
 [6]   Q3: I←I+4
 [7]   →Q4×ι∧/A=B[I;]
 [8]   →Q4×ι∧/A=B[I+1;]
 [8.1]  →Q4×ι∧/A=B[I+2;]
 [9]   →Q3×ι∧/A≠B[I+3;]
 [10]  I←I+3
 [11]  Q4:  →(ROWS≥I)/OUT
 [11.1]  'TIME ';(⌶21)-XXXX
 [12]  R←0
 [13]  →0
 [14]  OUT: R←I
 [15]  'TIME ';(⌶21)-XXXX
∇
```

The information in Table 1 shows an average improvement of only 6.7 percent by using L4 instead of LUKUP.

Since, in our experiments, a substantial decrease in required execution time occurred by use of LUKUP, we will

utilize this routine to perform table lookup in PILOT. This same routine may also be used in the process of variable resolution. One table called VARTAB contains variable names, and two auxiliary tables named VALTAB and VACTAB contain the values for numeric and character variables, respectively. The pointer into the variable name table also points to the corresponding location of the value for that variable in either VALTAB or VACTAB. Variable names are of the form "∪variablename" for numeric variables or "∩variablename" for character variables, where "variablename" is one to eight characters in length.

During the parsing phase, when a variable name is encountered, the prefix ("∪" or "∩") is removed and the variable name table VARTAB is searched for that variable name. If the variable name has already been entered, processing continues. Otherwise, the variable name is entered in the next available location in the table. Initially all numeric variables are initialized at zero, and all character variables are initialized at blanks. A standard procedure in PILOT is that any undefined variables simple retain their initial values and processing continues in the usual manner. In this implementation, however, a warning message is ussued to the user if undefined variables are encountered.

If a variable value is to be accepted as input via an ANSWER statement, the parsed statement contains a character string consisting of the name of the table (VALTAB or VACTAB) in which the entry is to be made, the pointer into the table which serves as the subscript, and either a quad (⎕) for numeric entry or a quote quad (⍞) for literal entry completes the string. Upon execution the UNQUOTE function, which strips the quotes from a character string and executes it, is used to perform input. A question mark is typed so that if no variable is used, the user will be alerted to the fact that the system is awaiting input. If a variable is awaiting input a quad is also typed. Then the user is free to enter a value. In this way, variable resolution is accomplished.

Label resolution occurs in a similar manner. As each statement is accepted, the first character is examined. If it is a star (*), the scan for the label continues until the first blank is encountered. In the event that the label consists of more than eight characters, it is truncated to the leftmost eight characters. The label table LABTAB is dimensioned in such a way as to allow a ninth character to be entered for each label. This character is either a "0" or a "1" to indicate whether the label has been defined or not. In this way, forward label reference is permitted since a label that is undefined at the moment may be entered in the

table with a "1" in the rightmost position, and later in the parsing phase the "1" can be reset to "0" if the label is defined. Thus, when the parsing phase has been completed, undefined labels can easily be located by scanning the rightmost column for the location of any ones which might be present. For each that is located, the program will print the label name corresponding to that location accompanied by an appropriate error message.

When a labeled statement is accepted, the label name is entered in the table called LABTAB. The number corresponding to its position in the parsed statement table is then placed in the corresponding position in a label value table called LABVAL. If the object of a PILOT statement is a label, e.g., the JUMP statement, the parsing phase places the character string "LABVAL" followed by a subscript that is the pointer to the location of the label in LABTAB in the object field. This pointer value is determined through the use of the table search routine LUKUP. During execution, the UNQUOTE function is applied to the contents of the object to convert the literal to a numeric value. This value is then assigned to the location counter so that the parsed statement corresponding to this location will be interpreted next. Utilization of this scheme permits forward label reference to be accomplished rather easily.

Subroutines may be accessed through the USE (U) statement. This instruction causes a temporary transfer of control to a subroutine, where the subroutine name is the object. At the end of the subroutine, execution returns to the statement immediately following the U-statement.

APL programs may be called by using the U-statement. If the first nonblank character of the object is not a star, the APL program corresponding to the object is loaded and executed. After the last line of the called program is executed, standard linkage of APL causes the line which immediately follows the U-statement to be executed next.

PILOT subroutines are specified by the use of a label name, complete with the required star prefix, in the object field. This label refers to a statement in the program, so it is clear that forward reference is required. When a USE statement is encountered, label resolution is performed in the same manner as for an ordinary statement label. Then the variable LEVEL is incremented by one and tested; if the value of LEVEL exceeds two, this indicates that a subroutine has attempted to call another subroutine. In this case an error message is printed and execution is terminated. If the test is satisfied, however, the contents of the location counter LOC are stored in the variable called HOLDER, the UNQUOTE function is applied to the object to convert it to a numeric value, the location counter then receives this value,

and a JUMP to that location is made. Each time an END statement is encountered, the variable LEVEL is decremented by one and tested. If it is equal to zero, the processing is terminated and the message "INTERPRETATION COMPLETE" is printed. Otherwise the END statement is used to complete the USE routine by reloading the location counter LOC with the contents of the variable HOLDER. This value is then incremented by one and interpretation continues at the entry in the parsed statement table corresponding to that value.

An elementary example of a PILOT program which uses a subroutine accessed by the U-statement is the following:

```
T: THIS IS THE BEGINNING
U: *MIDDLE
*END T: THIS IS THE END
E:
*MIDDLE T: THIS IS THE MIDDLE
E:
```

The result of the execution of the program is as follows:

```
THIS IS THE BEGINNING
THIS IS THE MIDDLE
THIS IS THE END
INTERPRETATION COMPLETE
```

The parsed statement table is:

```
T⊃' THIS IS THE BEGINNING'
U⊃*LABVAL[1]
T⊃' THIS IS THE END.'
E⊃⍬
T⊃' THIS IS THE MIDDLE.'
E⊃⍬
\
```

Note that in the second statement, the star precedes the object, which is a pointer into the label value table. Although we use the same label resolution method, we need a star in the object to indicate to the interpreter that this is a PILOT subroutine. This is easily accomplished by coding *OBJECT*←'*';*OBJECT*, which results in prefixing a star to the object.

During the program definition phase, each time the user inputs a PILOT statement it is checked for errors at the same time that the statement is being broken down into its components, i.e., the op code, conditional, object, variables and labels. When an error is discovered, an error message is printed and the system waits for the user to input a substitute line. This action is repeated until the user enters a valid line to be used as a replacement or a reverse slash (\) is entered to indicate to the parsing phase that control is to be transferred to the interpretive phase.

As the user is entering his program, he may make a mistake in typing. If he has not hit the carriage return key on the line which contains the error, he may make the necessary correction by backspacing to the error, striking the attention key, then retyping the remainder of the line from that point. If, however, the error is in a previous statement, it may be corrected using the FIX command. The format of this command is

FIX n statement

where "n" is the number of the line to be corrected, which is determined by counting statements from the beginning of the program—ignoring any R-statements, and "statement" is a PILOT statement to replace the one which contains the error. The location counter is stored when a FIX command is encountered. The statement number that is specified in the FIX command is loaded into the location counter, and parsing continues in the normal manner. Immediately before returning for the next input line, the parser checks to see if a FIX is in effect by examining the FIX switch (FSWIC). If it is on, the location counter must be restored to its previous value, and processing continues as usual. If the FIX switch is equal to zero, no change is necessary.

If the current statement is a COMPUTE statement, the usual parsing procedure occurs until variable resolution time. Immediately after the pointer into the variable value table is substituted for the variable name in the object, a test is made to see if an APL replacement symbol (←) follows that variable name. If so, quotes are placed around the pointer to convert it to a literal string, so that at execution time the value at that location will be used, but not prior to that time. Then scanning the object for variable names continues until all have been found and resolved. Immediately

before the components are combined to form the new parsed statement, the subroutine CHEK is called to perform error analysis on the object. This error analysis is accomplished through the use of the Error Trap feature of YORK APL.

Error Trapping is based on the I28 function. It is evoked by specifying "E∆pgmname←stmtno" after the program has been defined, where pgmname is the name of the program which is to be monitored by the function and stmtno is the statement number or numbers which are to be checked. The first line of the program is reserved for the I28 function; this line may specify where control should be transferred in the event that the error occurs.

Error types are specified by a numeric value as follows:

1	Domain Error
2	Syntax Error
3	Value Error
4	Length Error
5	Rank Error
6	Number too big
7	Division by zero

For instance, if the statement being monitored contained a number divided by zero, the I28 function would contain the value 7. If no error occurs, the function remains the empty vector (⍳0). With this in mind, we can now write the following short program to perform Error Trapping for division by zero.

```
∇ D←UZERO OBCOPY
 [1] →(7=⌶28)/ER
 [2] ⍎'R→';OBCOPY
 [3] D←0
 [4] →0
 [5] ER: 'ATTEMPTED DIVISION BY ZERO'
 [6] D←1
∇

E∆UZERO←2

OBCOPY←'2÷0'

UZERO OBCOPY
ATTEMPTED DIVISION BY ZERO
  1
```

In order to evoke the Error Trap feature we must type "*E∆UZERO←2*" so that any attempt to divide by zero in statement 2 of UZERO will set the I28 function equal to 7. In the first statement we indicate that if an attempt to divide by zero is made, i.e., when I28=7, control should be transferred to the statement labeled ER. This action causes the error message "ATTEMPTED DIVISION BY ZERO" to be printed, then the variable called D is set equal to 1 to indicate that an error occurred. If no error occurs in statement 2, D is set equal to zero and I28 remains the empty vector. At this point we should note that the use of a default procedure of PILOT could cause this error to occur. Since PILOT allows undefined variables to maintain their initial values (which are blanks for character variables and zero for numeric variables), if a C-statement specifies that a value is to be divided by a variable that is undefined, standard PILOT

action results in an attempt to divide by zero. For this reason, a warning message is issued to the user when an undefined variable is encountered, then processing continues in the usual manner. If a division by zero error is encountered, the error message in our implementation should indicate that if such a warning has been issued, it could be at the root of the error.

In our analysis of the object of the COMPUTE statement, it is necessary that we test for all possible errors. This may be done by using the Error Trap by coding:

```
[1] →((⍳8∈1+⌶28)/GO,DOM,SYN,VAL,LEN,RAN,BIG,ZER
```

as the first statement in the program, this location is reserved for the I28 function if it is to be used. GO refers to the statement to set the variable to be returned equal to zero to indicate that no errors were present. The other labels refer to statements in the program which output appropriate error messages and then set the variable to be returned equal to a value of 1 to indicate that an error occurred.

For example, we could code the routine VAL, which handles value errors, as follows:

```
VAL: 'VALUE ERROR IN COMPUTE STATEMENT. POSSIBLY'
'AN UNDEFINED VARIABLE OR ATTEMPT MADE TO'
'USE PROGRAM NOT IN WORKING STORAGE.'
D←1
→0
```

By combining the routines in this fashion, we are now able to define a subroutine called CHEK which is passed the object from LEX, executes the object using the UNQUOTE function in conjunction with the Error Trap, and then returns a value of D. Then LEX can check the variable to see if an error was found; if so, the user should enter a substitute line. However, if no errors were encountered, the parsing should be completed, then the parsed statement should be entered in the parsed statement table.

A listing of the subroutine CHEK is contained in Appendix B.

The casual observer might note that a table driven technique might be more beneficial. Due to the nature of APL, however, the attempt to utilize this technique causes many problems concerning branching to arise, and their resolution requires more storage. For that reason the brute force approach was chosen.

If it becomes necessary in the future to add instructions to the language, the op codes should be added to statements 204 and 670 of LEX to cause a branch to the proper routine to occur. If special handling is required during the parsing phase, the routine to accomplish this should be placed at a location immediately following an absolute branch. A routine to perform the necessary interpretive actions should be placed after statement 717 in LEX.

CHAPTER IV

THE MATCH STATEMENT

Probably the most important instruction in the PILOT language is the MATCH (M) statement. As was mentioned in a previous chapter, this instruction facilitates conversational programs by searching user responses for specific character strings. The M-instruction causes a moving window string match to be performed in which the input to the most recent A-statement is compared to each of the items which comprise the object of the M-statement. If either of the items is contained in the input string, the match-key is set to one. Otherwise, it remains equal to zero to indicate that no match was found. The items are separated by commas which serve as a means to indicate the presence of leading or trailing blanks in each of the items to be compared.

As this instruction is used quite frequently in PILOT programs, care should be taken in selecting an efficient searching algorithm to be used. Knuth (6) pointed out that the most time-consuming portion of many programs is that portion which performs searching, and that a substantial increase in execution speed may be gained by using a more efficient method. Let us first consider the most obvious approach to our problem—a simple sequential search. In this method, the input string is searched linearly by comparing

each element of the input string sequentially to its corresponding element in the pattern string. If a match is found, the program sets the match-key to one in order to indicate that the pattern was contained in the input string, and execution is terminated. On the other hand, if no match is found, the testing continues through the end of the input string and the program terminates with the match-key set equal to zero. Figure 4 is a flowchart to perform a simple sequential search of an input string for a pattern string.

We shall now write a short YORK APL program, called SEEK, to perform these actions. It will first input two character strings called MSTRING and PATTERN, then it will test PATTERN against the first X characters of MSTRING, where X is the length of the string PATTERN. If a match is found, the word "YES" will be printed and the program will terminate. However, if no match is found, the string MSTRING is rotated left one position. Then the counter I is incremented by one and compared to the length of MSTRING. If I is greater than the length of MSTRING, the entire character string has been searched unsuccessfully. In this case, the word "NO" is printed and the program terminates. Otherwise, the entire process is repeated.

The program could be written as follows:

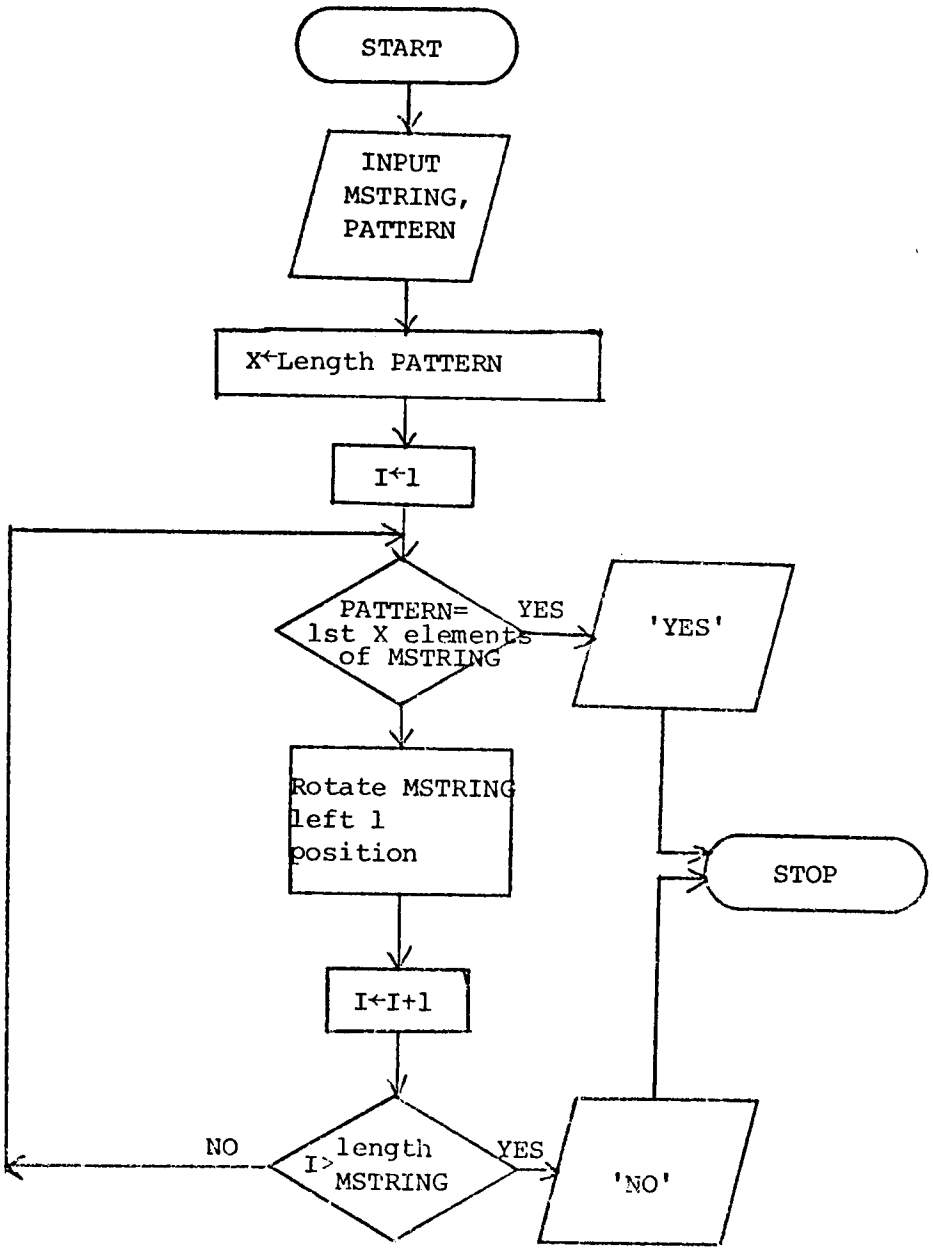


Figure 4

Flowchart of Simple Sequential Search

```
     ∇SEEK
[1]  ⍝ INPUT STRING TO BE SEARCHED
[2]  MSTRING←⍞
[3]  ⍝ INPUT PATTERN STRING
[4]  PATTERN←⍞
[5]  ⍝ DETERMINE LENGTH OF PATTERN
[6]  X←⍴PATTERN
[7]  ⍝ INITIALIZE COUNTER
[8]  I←1
[9]  ⍝LOOP FOR SEARCH
[10] AGIN: →YES×⍳∧/(PATTERN=X⍴MSTRING)
[11] ⍝ LEFT-ROTATE MSTRING ONE POSITION
[12] MSTRING←1⊖MSTRING
[13] ⍝ INCREMENT COUNTER
[14] I←I+1
[15] ⍝ TEST FOR I > LENGTH OF MSTRING
[16] →(I>⍴MSTRING)/NO
[17] ⍝ IF NOT, REPEAT
[18] →AGIN
[19] ⍝ A MATCH WAS FOUND
[20] YES: 'YES'
[21] →0
[22] ⍝ NO MATCH WAS FOUND
[23] NO: 'NO'
[24] ∇
```

As an example, let MSTRING be the character string "AAAAAAB" and let PATTERN be the pattern string "AAAB." Now call SEEK.

```
     SEEK
INPUT STRING TO BE SEARCHED
AAAAAAB
INPUT PATTERN STRING
AAAB
YES
```

In this example, the pattern is A^nB and the input string is $A^{2n}B$. The major drawback of this method is that in general it requires $(n+1)^2$ character comparisons to be made in order to perform the search. Another problem with using

the simple linear search is that it involves "backing up," i.e., if a mismatch occurs, the next comparison begins from the first character in the pattern string. If this need were eliminated, a substantial savings in time could be achieved.

Using a linear search, the average number of characters that must be tested in MSTRING is approximately M/2, where M is the length of MSTRING. One should also note that, in general, this method requires (1+(N-M))M = 0(MN) units of time, where M and N are the respective lengths of MSTRING and PATTERN. Clearly, for long character strings the user will be unwilling to tolerate the slow response time which this program would require. Thus, we must attempt to find a more efficient method. In fact, an algorithm for performing pattern-matching exists which only requires 0(M+N) units of time. Let us now consider that method.

Perhaps the best point at which to begin this discussion is with the concept of the moving-window string match. Returning for a moment to the previous example, let us examine how the comparisons were made. At the first iteration (I = 1), the character strings had the following appearance:

```
MSTRING = A A A A A A B
              ↓
PATTERN = A A A B
```

Since the first four characters of MSTRING are not equal to PATTERN, MSTRING is rotated 1 position and I = I+1 = 2. Now,

```
MSTRING = A A A A A B A
                  ↓
PATTERN = A A A B
```

does not result in a match, so MSTRING is rotated once again and I becomes equal to 3. Comparing once more we have

```
MSTRING = A A A A B A A
                  ↓
PATTERN = A A A B
```

which results in the same action being repeated. At last a match is found at I = 4

```
MSTRING = A A A B A A A
PATTERN = A A A B
```

This time, the word "YES" is printed and program execution is terminated.

This method is a variation of the moving-window string match; in our example the string itself is rotated. Now consider placing the pattern string over the input string in such a way that the pattern can be moved in a slide rule fashion over the input string. This is the general concept of a moving-window string match, which we will illustrate with the same example strings as were used in the last example.

The initial configuration is

```
A A A B
      ↓
A A A A A A B
```

As before, a mismatch occurs in the fourth position, so this time we slide the pattern string right one position:

```
  A A A B
        ↓
A A A A A A B
```

Since this gives a mismatch, we slide PATTERN again:

```
    A A A B
          ↓
A A A A A A B
```

and once more after another mismatch:

```
      A A A B

A A A A A A B
```

giving a successful match. Notice that the pattern is not moved unless a mismatch occurs. In addition, it is not necessary to remember the characters that were scanned previously since the position in the pattern provides sufficient information to recreate them. In this example it was necessary to move the pattern string only one position at a time since the same character, i.e., "A," was repeated.

Let us now consider a second example in which this is not true.

Let MSTRING be the character string "BABCBABCAABCABCABC" and let PATTERN be the string "ABCABCAB." Now the initial configuration is

```
A B C A B C A B
↓
B A B C B A B C A A B C A B C A B C
```

Since A ≠ B we shift the pattern right one position and try again.

```
  A B C A B C A B
      ↓
B A B C B A B C A A B C A B C A B C
```

Another mismatch occurs, this time in the fourth position in PATTERN. Thus we can deduce that the previous four characters of the input string were "A B C x," where A ≠ x. It is not important to know what the character value of x is; we only need to shift the pattern right four positions since shifting less than four positions would obviously be in vain. Now we have

```
        A B C A B C A B
            ↓
B A B C B A B C A A B C A B C A B C
```

This time a mismatch occurs in the fifth position of the pattern, so we shift it right five positions, yielding

A B C A B C A B

B A B C B A B C A A B C A B C A B C

This time a match was found. This description suggests that with a table indicating how many positions to slide the pattern whenever a mismatch occurs at its jth position, the pattern-matching program could be executed much more efficiently. Table 2 shows such a table for the pattern string PATTERN.

Table 2. Failure-link array for "ABCABCAB"

j	1	2	3	4	5	6	7	8
pattern(j)	A	B	C	A	B	C	A	B
flink(j)	0	1	1	1	2	3	4	5

The algorithm to be used for the calculation of the failure-link (flink) array will be presented shortly. A graphic means of representing the array is shown in the finite state diagram shown in Figure 5.

At each step in the procedure, either the pattern or the pointer into the input string is moved. Since the pattern may move a maximum of N times and the pointer may move a maximum of M times, where N and M are the respective lengths of the pattern and the input string, only M+N steps

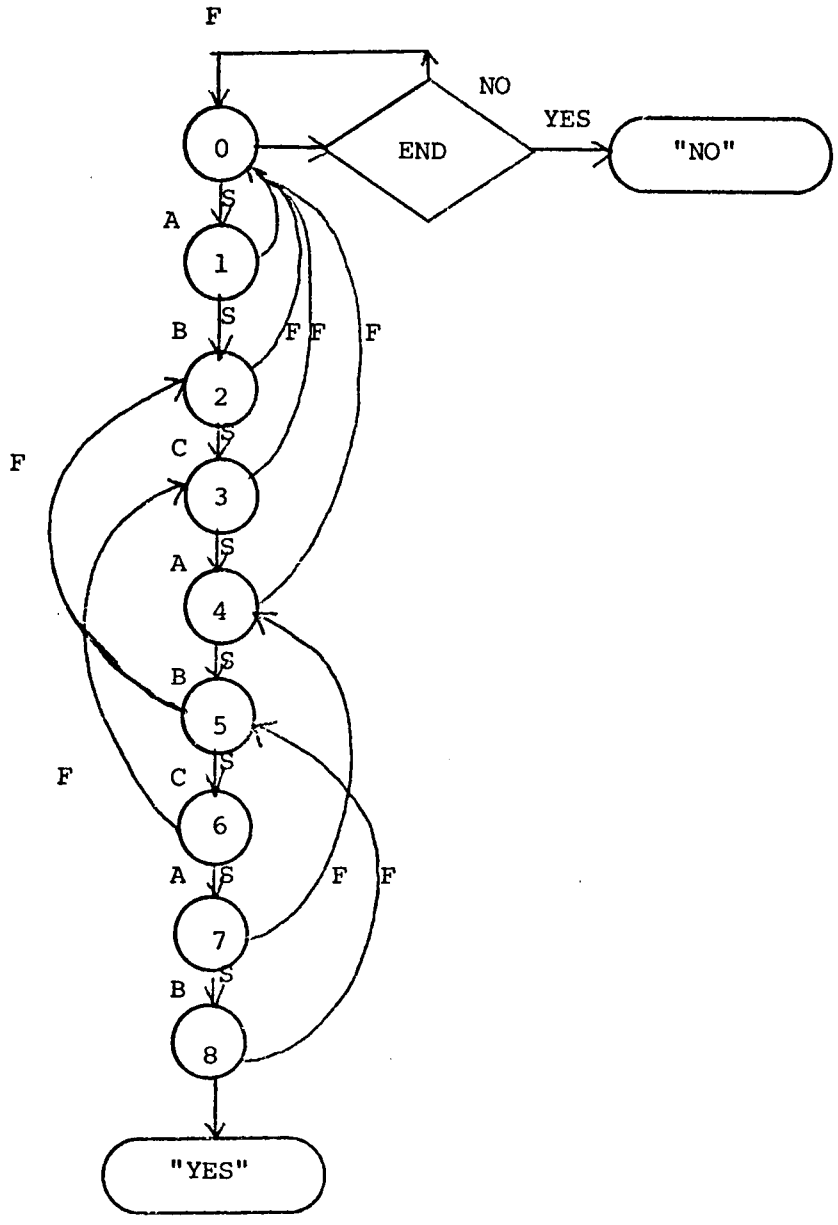


Figure 5

Finite State Diagram of Failure-Link
Array for Pattern = "ABCABCAB"

need be executed in the worst case once the flink table has been derived. Obviously it is much more convenient to maintain a pointer variable—say j—for the operations involved rather than physically moving the pattern. Our movement of the pattern here was for illustrative purposes only.

The general form of the pattern-matching algorithm of Knuth, Morris and Pratt (5) is as follows:

```
place pattern at left;
while pattern not fully matched
     and text not exhausted do
     begin
          while pattern character differs from
              current text character
              do shift pattern appropriately;
          advance to next character of text;
end;
```

Again, we shall assume for now that the input text is contained in an array named MSTRING and that the pattern is stored in an array called PATTERN. Let i be the variable pointer to indicate the current member of MSTRING and let j point to the corresponding member of PATTERN. M and N are again the lengths of MSTRING and PATTERN, respectively. Thus, the above algorithm may be written as (1):

```
j := k := 0;
while j ≤ m and k ≤ n do
     begin
          while j > 0 and MSTRING (k) ≠ PATTERN(j)
               do j := FLINK(j);
          k := k+1;     j := j+1;
     end;
```

When execution of the program has been completed, the value of j must be examined to determine the results of the match. If the value of j exceeds the length of MSTRING, positions k-m through k-1 contain the substring PATTERN. However, if $j \leq m$, PATTERN was not contained in MSTRING.

The validity of this program may be proved (5) as follows: Let p be the position in MSTRING immediately preceding the first element of PATTERN, written p = k-j. Thus, MSTRING(p+i) = PATTERN(i) for some $1 \leq i < j$, but for $1 \leq t < p$, MSTRING (t+i) ≠ PATTERN (i) for some i, where $1 \leq i \leq m$.

It is necessary that a means of computing the failure-link table be found so that it remains constant throughout the program execution. By inspection of the program, we see that whenever the step j := FLINK(j) is executed, the value of j must be greater than zero. We also are aware that the last j elements of MSTRING at that time were the characters

PATTERN(1) through PATTERN(j-1)x, where x cannot be equal to PATTERN(j) since that is precisely where the mismatch occurred. It is desired that we determine the minimal shift value which can be used so that a substring of elements of MSTRING match those of PATTERN. That is to say, the value of FLINK(j) should be the maximum i such that i is less than the value of j, and the previous i elements of MSTRING were PATTERN(1) through PATTERN(i-1)x, with PATTERN(i) $\neq$ PATTERN(j). For the case in which no i exists that meets these requirements, set FLINK(j) = 0. Thus, it follows that MSTRING(t+1) through MSTRING(k) $\neq$ PATTERN(1) through PATTERN(k-t), where $k-j \leq t < k-FLINK(j)$. Now we have established a constant relation to fit our needs.

Morris and Pratt (8) presented their method of performing pattern matching by the following three procedures written in an Algol-like language form:

```
begin integer array String ( 1:M), Pattern, Flink ( 1:N);
     comment  The procedure Subs will return true iff an
     N-character substring of String matcnes Pattern.
     Setup is used to fill the vector of failure-links,
     Flink, so that Subs need never back up in String.
     Match is a common subroutine;
     procedure Match (Ch,j); value Ch; integer Ch, j;
          comment  Match sets j to the index of the next
          character in Pattern to be compared to the
          character after Ch (wherever it comes from);
          begin Loop:  if j = 0 then go to INC;
               if Ch = Pattern (j) then go to INC;
               j := Flink (j); go to Loop;
          INC: j := j+1
     end;
```

```
procedure Setup;
     begin integer k, j; j:= 0;
     comment i will sweep down Pattern while j bounces
     around,
     always above (<) i;
     for i := 1 step 1 until N do
          begin Flink (i) := j;
               comment  For 1≤k≤i, let M = Flink (k) - 1.
               Then if M 0 the sequence Pattern (1) ...
               Pattern (M) matches Pattern (k-M) ...
               Pattern (k-1);
               Match (Pattern (i),j);
          end;
     end Setup;


Boolean procedure Subs;
     begin integer i, j; j := 1;
          comment i sweeps down string while j bounces
          around in pattern;
          if M ≠ 0 then
          for i := 1, i+1 while i≤M∧j≤N
               do Match (String (i),j);
          Subs :=j>N;  comment true iff pattern is
          exhausted;
     end Subs;
comment  The program using the procedures is placed here;
end;
```

In order to concentrate only on the method in which the failure-link array is computed, we trace execution of the steps, ignoring for now any statements that are not involved directly in this calculation. Thus, we see that the general form of the failure-link array computation is:

```
begin

     j:=0;

     for i:=1 until N  do

          begin
```

```
                flink (i):=j
                while j>0 and pattern (i) ≠ pattern (j)
                do
                     j:=flink j;
                j:=j+1;
        end;
```

It is now a rather simple task to code the following YORK APL program to perform these steps:

```
      ∇SETUP PATTERN
[1]   ⍝ INITIALIZATION
[2]   J←0
[3]   FLINK←100⍴0
[4]   ⍝ CALCULATE THE LENGTH OF PATTERN
[5]   N←⍴PATTERN
[6]   ⍝ SETUP FAILURE LINK ARRAY
[7]   I←1
[8]   L: →(I>N)/OUT
[9]   ⍝ DEFINE FLINK [I]
[10]  FLINK[I]←J
[11]  ⍝ IF J > 0 COMPARE PATTERN [I] TO PATTERN [J]
[12]  LOOP:  →(J=0)/INCR
[13]  ⍝ IF THEY ARE EQUAL, INCREMENT BOTH I AND J
[14]  →(PATTERN[I] = PATTERN[J])/INCR
[15]  ⍝ IF THEY ARE NOT EQUAL, USE FAILURE-LINK
[16]  J←FLINK[J]
[17]  →LOOP
[18]  ⍝ INCREMENT IF J = 0 OR IF PATTERN[I] = PATTERN[J]
[19]  INCR: J←J+1
[20]  I←I+1
[21]  →L
[22]  ⍝ WHEN I EXCEEDS THE LENGTH OF PATTERN
[23]  ⍝ DISPLAY FLINK AND TERMINATE EXECUTION
[24]  OUT: 'FLINK '; FLINK
[25]  ∇
```

As an example, assume that PATTERN = "ABCABCAB" as in a previous example. We now call the subroutine SETUP, with PATTERN as its only argument:

```
      SETUP PATTERN
FLINK 0 1 1 1 2 3 4 5
```

FLINK is computed to be exactly what we expected. Now that a means for computing the failure link array is at our disposal, let us combine it with the matching algorithm stated earlier in order to write the YORK APL subroutine listed below:

```
∇ R←PATTERN COMP STRING
 [1]  J←0
 [2]  FLINK←⍳0
 [3]  M←⍴STRING
 [4]  N←⍴PATTERN
 [5]  I←1
 [6]  L: →(I>N)/OUT
 [7]  FLINK[I]←J
 [8]  LOOP: →(J=0)/INCR
 [9]  →(PATTERN[I] = PATTERN[J])/INCR
[10]  J←FLINK[J]
[11]  →LOOP
[12]  INCR: J←J+1
[13]  I←I+1
[14]  →L
[15]  OUT: J←1
[16]  →(0=M)/SOUT
[17]  I←1
[18]  L2: →((I>M)∨(J>N))/SOUT
[19]  LEOP: →(J=0)/INCRE
[20]  →(STRING[I] = PATTERN[J])/INCRE
[21]  J←FLINK[J]
[22]  →LEOP
[23]  INCRE: J←J+1
[24]  I←I+1
[25]  →L2
[26]  SOUT: →(J>N)/YES
[27]  R←'NO'
```

```
     [28] →0
     [29] YES: R←'YES'
    ∇
```

The next step is to consider ways to make the subroutine more efficient (5). Although our original sequential search program required M times N units of time for the worst case, this worst case seldom occurs in realistic data. Thus, the most noticeable advantage of our new version is that there is no need for backing up after a partial match. Several refinements may be made in the current version of the program to allow it to operate more efficiently.

One inefficiency in the program is that the program spends unnecessary time in finding that MSTRING(i) ≠ PATTERN(1), where i = 1, 2, 3, . . ., M, since in a large alphabet the possibility of the occurrence of a partial match is somewhat remote. Another case is in the event that j = 1 and MSTRING(i) ≠ PATTERN(1), j is set equal to FLINK(1), which is universally zero. In the next step, it sees that j = 0, so it is incremented to become equal to 1 again. Then it checks to see if 1 is greater than N, and yet another test is made to see if 1 = 0. This obvious waste of time may easily be alleviated by handling the case where j = 1 as a special routine.

Yet another waste of time is evident in the repetitive testing to see if i is greater than the length of MSTRING, or

if j exceeds the length of PATTERN. Instead, a character which will never be used in a match—say "α"—may be catenated to the end of PATTERN. In a similar manner, catenate a second character that will never be used—say "⊥"—and the first element in PATTERN to MSTRING, in that order. This eliminates the need for frequent testing to see if j exceeds the value of N. In addition, the value -1 may be catenated to FLINK so that the test to see if j is negative may be placed in a section of code that is executed less often.

The modified algorithm (5) may now be written as follows:

```
     k := 0;  a := PATTERN (1);
     PATTERN (m+1) := "α";  FLINK (m+1) := -1;
     MSTRING (n+1) := "⊥";  MSTRING (m+2) := a;
advance:  repeat k := k+1  until  MSTRING (k) = a;
     if k>n  then go to input exhausted;
          j := 1;
char matched:  j := j+1;  k := k+1;
loop:  comment j>0;
     if MSTRING (k) = PATTERN (j) then go to char matched;
     j := FLINK (j);
     if j = 0 then go to advance;
     if j = 1 then begin
          if MSTRING (k) ≠ a then go to advance
          else go to char matched end;
     if j>0 then go to loop;
comment MSTRING(k-m) through MSTRING (k-1) matched;
```

Combining our routine SETUP, which we used to calculate the failure-link array, with this modified algorithm, we now are able to write this YORK APL subprogram for use in the MATCH function:

```
∇ R←PATTERN ELT STRING;XXX;J;FLINK;M;N;I;K;A
 [1]   ⍝ CALL TIMING ROUTINE
 [2]   XXX←TIMEON
 [3]   J←0
 [4]   ⍝ SETUP INITIAL FAILURE LINK ARRAY
 [5]   FLINK←50⍴0
 [6]   ⍝ CALCULATE LENGTHS OF STRING AND PATTERN
 [7]   M←⍴STRING
 [8]   N←⍴PATTERN
 [9]   ⍝ SET UP FAILURE LINK ARRAY
 [10]  I←1
 [11]  L: →(I>N)/OUT
 [12]  FLINK[I]←J
 [13]  LOOP: →(J=0)/INCR
 [14]  →(PATTERN[I] = PATTERN [J])/INCR
 [15]  J←FLINK[J]
 [16]  →LOOP
 [17]  INCR:J←J+1
 [18]  I←I+1
 [19]  →L
 [20]  ⍝ CATENATE SPECIAL CHARACTERS TO PATTERN AND STRING
 [21]  OUT: K←1 ⋄ A←PATTERN[1]
 [22]  PATTERN←PATTERN,'⍺'
 [23]  FLINK[I]←¯1
 [24]  STRING←STRING,'⊥'
 [25]  STRING←STRING,A
 [26]  ⍝ CHECK LOCATION OF A IN STRING
 [27]  ADVANCE: →(STRING[K]=A)/TST1
 [28]  K←K+1
 [29]  →ADVANCE
 [30]  TST1: →(K>M)/NO
 [31]  J←1
 [32]  MATCHED:  J←J+1
 [33]  K←K+1
 [34]  LOOPE: →(STRING[K]=PATTERN[J])/MATCHED
 [35]  J←FLINK[J]
 [36]  →(J=0)/ADVANCE
 [37]  →(J≠1)/TST2
 [38]  →(STRING[K]≠A)/ADVANCE
 [39]  →MATCHED
 [40]  TST2: →(J>0)/LOOPE
 [41]  ⍝ PATTERN WAS CONTAINED
 [42]  R←'YES'
 [43]  'TIME REQUIRED WAS '; TIMEON-XXX
 [44]  →0
 [45]  NO: R←'NO'
 [46]  'TIME REQUIRED WAS '; TIMEON-XXX
 [47]  ⍝ PATTERN NOT CONTAINED IN STRING...OUTPUT TIME USED
 [48]  ⍝ AND TERMINATE EXECUTION
∇
```

Let us now try executing the subroutine:

```
      PATTERN←'ABCABCAB'
      MSTRING←'BABCBABCAABCABCABC'
      PATTERN ELT MSTRING
TIME REQUIRED WAS 66
YES
```

Note that in this subroutine we defined the temporary variables that are created in the execution of the program to be local variables, i.e., they are to be deleted after execution of the program is completed to avoid cluttering the workspace needlessly. This is accomplished by including them in the function definition line, separated by semicolons.

A second feature of YORK APL is incorporated in this subroutine—the I21 (I-beam) function. This function, which is furnished by YORK APL, is accessed by calling the subroutine TIMEON with no arguments. TIMEON has been previously defined as follows:

```
∇ R←TIMEON
 [1] R←⌶21
∇
```

When called, TIMEON returns the result of the I21 function, which is the total amount of time—expressed in sixtieths of a second—that the computer has been utilized since the time that the user signed-on. This provides a convenient means of timing program execution, since we may call TIMEON at the beginning of the program, then just before program execution

is completed, we can call TIMEON again, subtract the old value from the new one, and output the result as the amount of time required.

It should be noted that the routine for calculating the failure-link array is not as efficient as possible. For instance, at the beginning of the program I is set equal to one, then compared to the value of N, which is the length of the pattern. Then the first element of the failure-link array is initialized at zero. A greater savings in time could be realized by making this a special case. By modifying this section of the program in a manner similar to that in which we rewrote a previous program (COMP), the following algorithm (1) is the result:

```
begin
     f(1) ← 0
     for j ← 2 until 1 do
          begin
               i ← f(j-1);
               while b_j ≠ b_i+1 and i > 0  do i ← f(i);
                    if b_j ≠ b_i+1 and i = 0  then f(j) ← 0
                    else  f(j) ← i+1
     end
end
```

This algorithm may now be coded into APL to replace the section used to compute the failure-link array in the previous program. The final version of the program becomes the following:

```
∇ R←PATTERN IN STRING;XXX;FLINK;M;N;J;I;A
 [1]  ⍝ CALL TIMING ROUTINE
 [2]  XXX←TIMEON
 [3]  ⍝ INITIALIZE FAILURE LINK ARRAY
 [4]  FLINK←50⍴0
 [5]  ⍝ CALCULATE LENGTHS OF STRING AND PATTERN
 [6]  M←⍴STRING
 [7]  N←⍴PATTERN
 [8]  ⍝ SETUP FAILURE LINK ARRAY
 [9]  FLINK[1] ← 0
 [10] J←2
 [11] LO1: →(J>N)/OUT
 [12] I←FLINK[J-1]
 [13] LO2: →((PATTERN[J] = PATTERN[I+1])∨(I≤0))/LO3
 [14] I←FLINK[I]
 [15] →LO2
 [16] LO3: →((PATTERN [J] ≠ PATTERN [I+1])∧(I=0))/ZERO
 [17] FLINK[J]←I+1
 [18] J←J+1
 [19] →LO1
 [20] ZERO: FLINK[J]←0
 [21] J←J+1
 [22] →LO1
 [23] ⍝ CATENATE SPECIAL CHARACTERS TO PATTERN AND STRING
 [24] OUT: K←1 ⋄ A←PATTERN[1]
 [25] I← N
 [26] PATTERN←PATTERN,'⍺'
 [27] FLINK[I]←¯1
 [28] STRING←STRING,'⊥'
 [29] STRING←STRING,A
 [30] ⍝ CHECK LOCATION OF A IN STRING
 [31] ADVANCE: →(STRING[K]=A)/TST1
 [32] K←K+1
 [33] →ADVANCE
 [34] TST1: →(K>M)/NO
 [35] J←1
 [36] MATCHED: J←J+1
 [37] K←K+1
 [38] LOOPE: →(STRING[K]=PATTERN[J])/MATCHED
 [39] J←FLINK[J]
 [40] →(J=0)/ADVANCE
 [41] →(J≠1)/TST2
 [42] →(STRING[K]≠A)/ADVANCE
 [43] →MATCHED
 [44] TST2: →(J>0)/LOOPE
 [45] ⍝ PATTERN WAS CONTAINED
 [46] R←1
 [47] 'TIME REQUIRED WAS '; TIMEON-XXX
```

```
 [48] →0
 [49] NO: R←0
 [50] 'TIME REQUIRED WAS '; TIMEON-XXX
 [51] ⍝ PATTERN NOT IN STRING...OUTPUT TIME USED
∇
```

In order to compare the relative execution times of these programs we now make use of the TIMEON function. The results of sample runs are shown in Table 3. Sample A represents the time required to search the input string "$J^{70}F$" for the pattern "F." Sample B indicates the time required to locate "TY" in the string "NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR PARTY."

Table 3. Average Times Required by Matching Routines Based on Samples Taken from Various Times During Operating Hours.

PROGRAM		SAMPLE A	SAMPLE B
	SEEK	121.0	104.7
	COMP	243.6	226.5
	ELT	91.58	103.3
	IN	87.66	101.6

Since experiments show the subroutine IN yields fastest results, we will incorporate it into our MATCH function. We now have a means of comparing two strings to see if one is

contained in the other, but the MATCH statement allows several strings to be specified, each of which is to be compared with the result of the last ANSWER statement. Therefore, we must write a routine to take the character strings, one at a time, and compare each with the result of the most recent ANSWER statement, using the subroutine IN. The following APL program accomplishes this action:

```
∇ R←ANS MATCH MSTRING;J;K;PATTERN
 [1] ⍝ CATENATE EXTRA COMMA TO END OF MATCH STRING
 [2] MSTRING←MSTRING,','
 [2.6] ANS←;ANS
 [2.78] MSTRING←;MSTRING
 [3] ⍝ REMOVE MULTIPLE BLANKS FROM ANSWER
 [4] J←1
 [6] REPEAT: J←J+1
 [7] →(J>⍴ANS)/TSTT
 [8] →(ANS[J]≠ANS[J-1])/REPEAT
 [9] →(ANS[J]≠' ')/REPEAT
 [10] ANS←(J-1)↑ANS),(J↓ANS)
 [11] J←J-1
 [12] →REPEAT
 [12.1] TSTT: J←0
 [13] ⍝ EXIT IF CUE WORDS EXHAUSTED
 [14] TST: →(2>⍴MSTRING)/AUS
 [15] K←1
 [16] AG:→(MSTRING[K]=',')/CFOUND
 [17] K←K+1
 [18] →AG
 [19] CFOUND: PATTERN←(K-1)↑MSTRING
 [20] ⍝ TRUNCATE MSTRING FOR NEXT MATCH
 [21] MSTRING←K↓MSTRING
 [22] ⍝ NOTE: LEADING/TRAILING BLANKS NOT REMOVED
 [23] ((⍴MSTRING)<(⍴PATTERN))/TST
 [24] J←((PATTERN ELT ANS)+J)>0
 [24.2] →TST
 [24.5] AUS: R←J
∇
```

Thus we now possess a program which can compare multiple strings as specified in a MATCH statement with the result of the most recent ANSWER statement in an efficient manner.

An extension of the MATCH function, which allows the user to search the contents of a variable for the presence of one or more string which comprise the object, has also been implemented. As is the case for all extended or modified PILOT instructions, the op code for this instruction consists of two characters - MV. In this manner instructions dealing with extensions or modifications to the language may readily be distinguished in the source program from core language instructions merely by examination of the op code.

The general form of the MV instruction is:

MV {conditional} : [unumeric variable name / ncharacter variable name], object

The variable name is required, followed by the character string(s) which compose the object. Use of this instruction provides greater programming flexibility by allowing the user to directly refer to previously defined variables to compare their contents to the elements of the object. As with the MATCH function, if any element of the object is contained in the variable contents, the match-key is set equal to 1. Otherwise it remains equal to zero.

CHAPTER V

SUMMARY AND CONCLUSIONS

The goal of this thesis has been not only to implement an interpretive PILOT system, but to select and analyze various techniques of accomplishing the desired results in order to determine which is best suited for its particular purpose in terms of time and space requirements. Strong emphasis has been placed on gaining efficiency where possible in an attempt to optimize the limited resources available. Initially we receive a workspace of approximately 32K bytes and a library of approximately 70K. The core requirements of the final versions of the programs are as follows:

ACCEPT	584
BLANK	120
CHEK	1112
IN	1400
LEX	17408
LUKUP	536
MATCH	968
TIMEON	56
UNQUOTE	64
USEIT	96

for a total of 22,344 bytes. Since this amount does not include the amount of memory allocated to variables and arrays that are generated, we must use a form of a planned-overlay structure in which we load only the programs required for the parsing phase, then, during initialization of the interpretive phase, we delete those programs that are no

longer necessary and load those which will be referenced by the second phase. This is accomplished by using the UNQUOTE function as described in Chapter III. Even with this structure, the amount of core available will limit the size of the parsed statement and source code tables to 50 by 50 arrays. It would be most helpful if PILOT could be allocated an independent workspace instead of loading both the interpretive program and the PILOT program in a single APL workspace of 32K. If this could be accomplished, both the length of PILOT statements and the maximum number of statements allowed could be increased.

Careful examination of several pattern-matching algorithms was made in order to determine which performs most efficiently. The selection process is described in Chapter IV. A similar approach to finding a desirable table search routine is discussed in Chapter III.

One extended instruction has been implemented—the MATCH-VARIABLE (MV) instruction. It provides greater flexibility since it allows the user to specify a previously defined variable as the first element of the object which is to be compared to the remainder of the object.

One command to the parsing phase has been provided. The FIX command allows the user to correct a previously defined line. In order to avoid exceeding the size of our workspace, the user must revert to APL commands in order to perform

other actions such as saving the PILOT code. If the workspace could be expanded, other PILOT commands could be added.

Another point of interest is the use of the Error Trap feature of YORK APL to perform error analysis on the object of a COMPUTE statement. The process of setting up the Error Trap procedure is described in Chapter III.

This implementation of the PILOT language is not a dead system. It could be used in a CAI course, for example, to allow the student to develop courses in PILOT as a part of the learning process. Another potential use is as an instructional tool by which an instructor may program a PILOT course, which may later be loaded and executed by the student.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

2. Richard C. Atkinson and H. A. Wilson, Computer-Assisted Instruction, Academic Press, Inc., New York, 1969.

3. Richard L. Gauthier and Stephen D. Ponto, Designing Systems Programs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1970.

4. David Gries, Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, 1971.

5. Donald E. Knuth, James H. Morris, Jr., and Vaughnan R. Pratt, "Fast Pattern Matching in Strings," Computer Science Department Publication STAN-CS-74-440, Stanford University, Stanford, California, 1974.

6. Donald E. Knuth, The Art of Computer Programming, Vol. 3, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

7. John A. Lee, The Anatomy of a Compiler, Reinhold Publishing Corporation, New York, 1967.

8. J. H. Morris, Jr. and Vaughnan R. Pratt, "A Linear Pattern-Matching Algorithm," Technical Report 40, University of California, Berkeley, California, 1970.

9. Sylvan Rubin, "A Simple Instructional Language," Computer Decisions 11 (1973), 17-18.

10. Sylvan Rubin, "PILOT-73," Information Sciences Laboratory Publication, Stanford University, Stanford, California, 1973.

11. J. Morgan Smyth, York APL, Ryerson Polytechnical Institute, Toronto, Canada, 1972.

12. John A. Starkweather, "PILOT-73 for the Datapoint 2200-12K," Computer Center Publication, University of California, Berkeley, California, 1973.

13. John A. Starkweather, "PILOT-73 Guide," Computer Center Publication, University of California, Berkeley, California, 1973.

APPENDICES

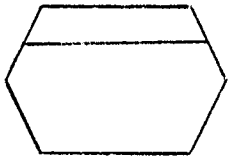
APPENDIX A

FLOWCHARTS

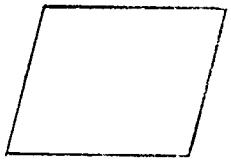
The flowcharts for the procedures are shown on the following pages. The following standard IBM flowcharting symbols are used.



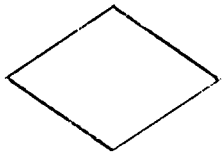
PROCESS



SUBROUTINE



INPUT/OUTPUT



DECISION



TERMINAL



OFFPAGE
CONNECTOR

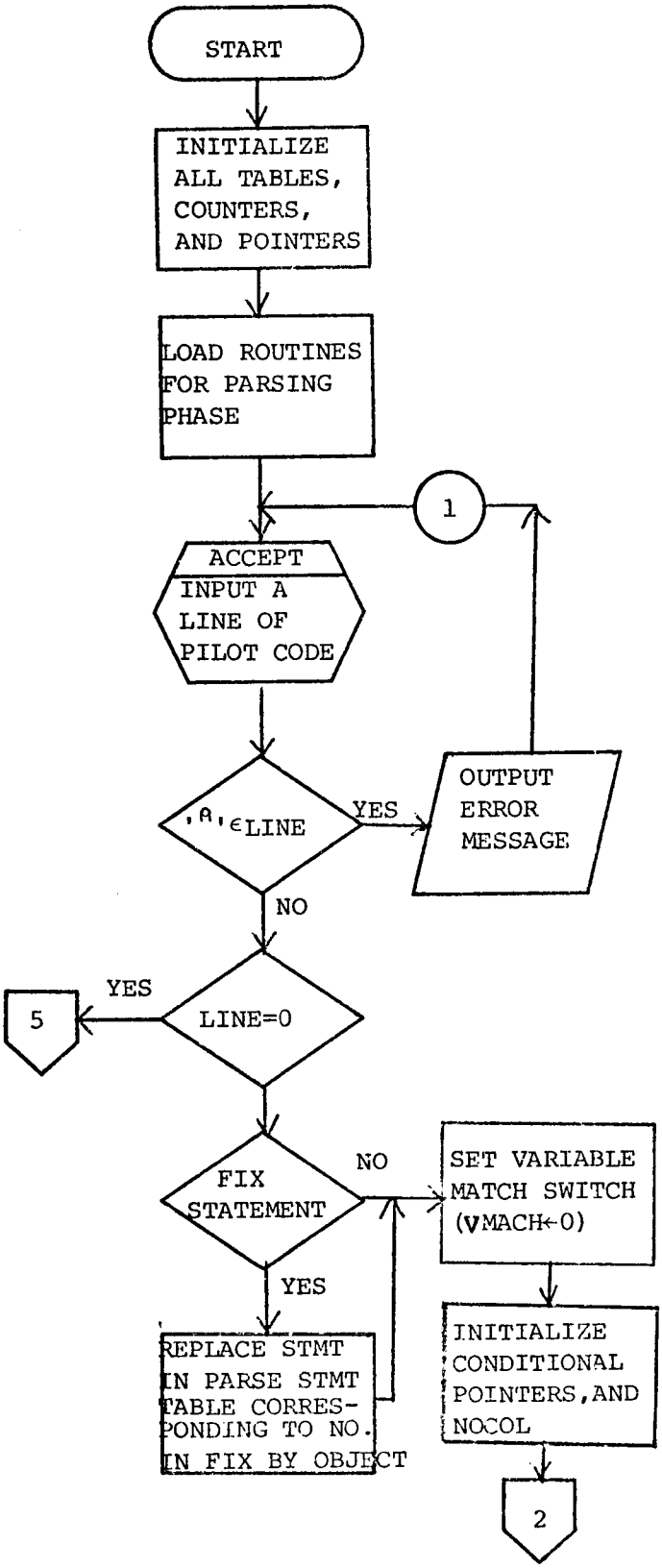


Figure 6

Flowchart for LEX

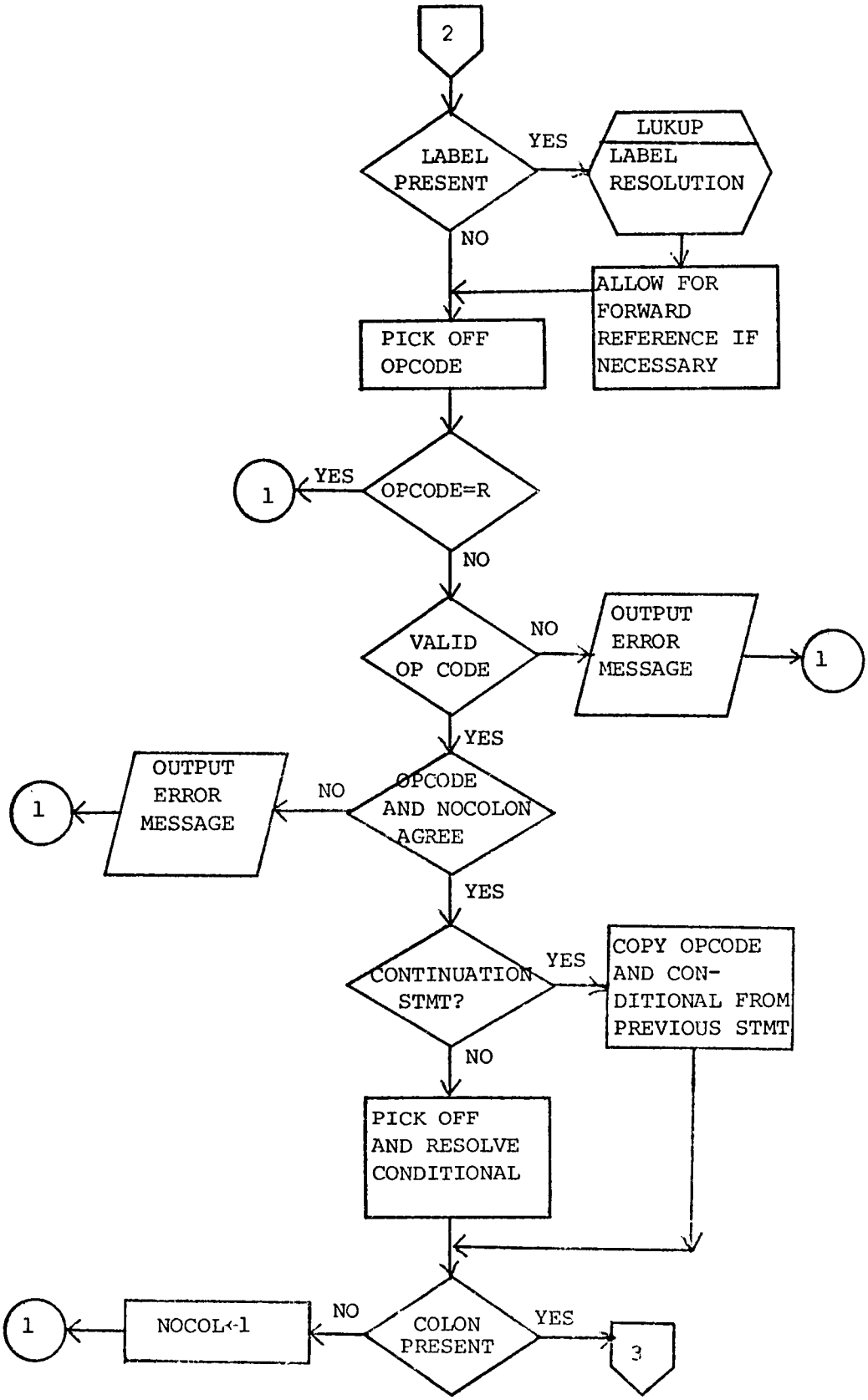


Figure 6 (Continued)

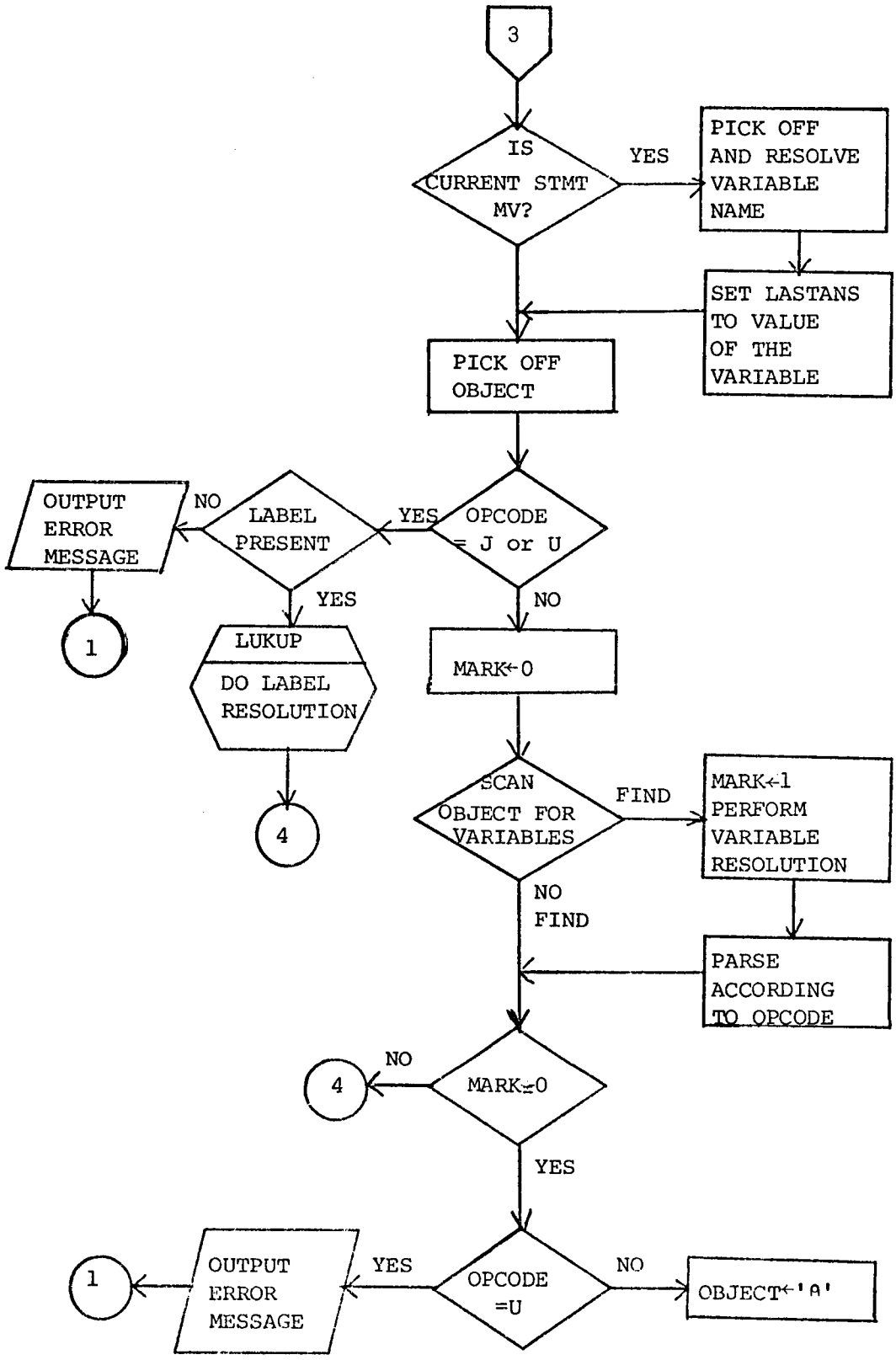


Figure 6 (Continued)

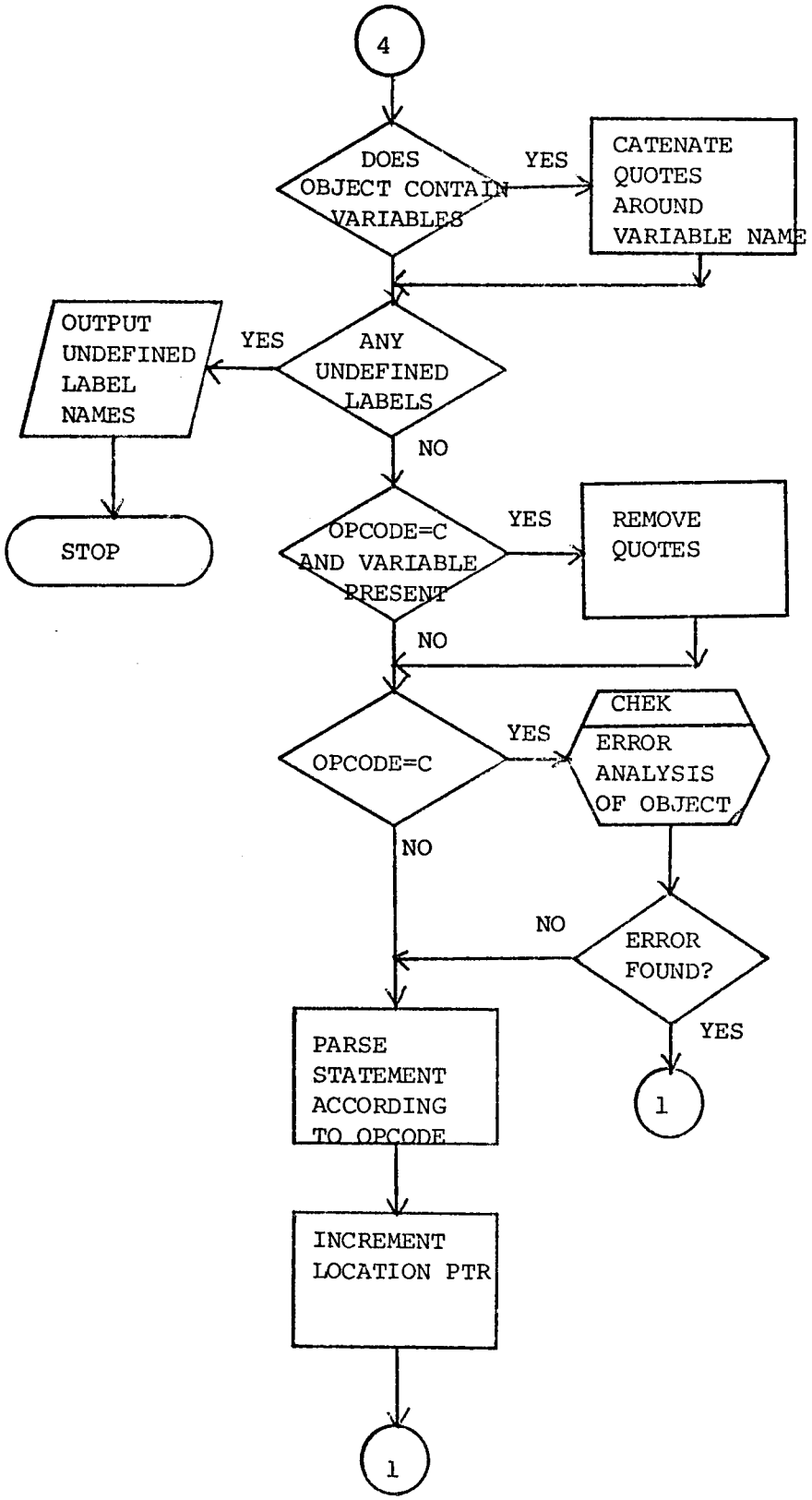


Figure 6 (Continued)

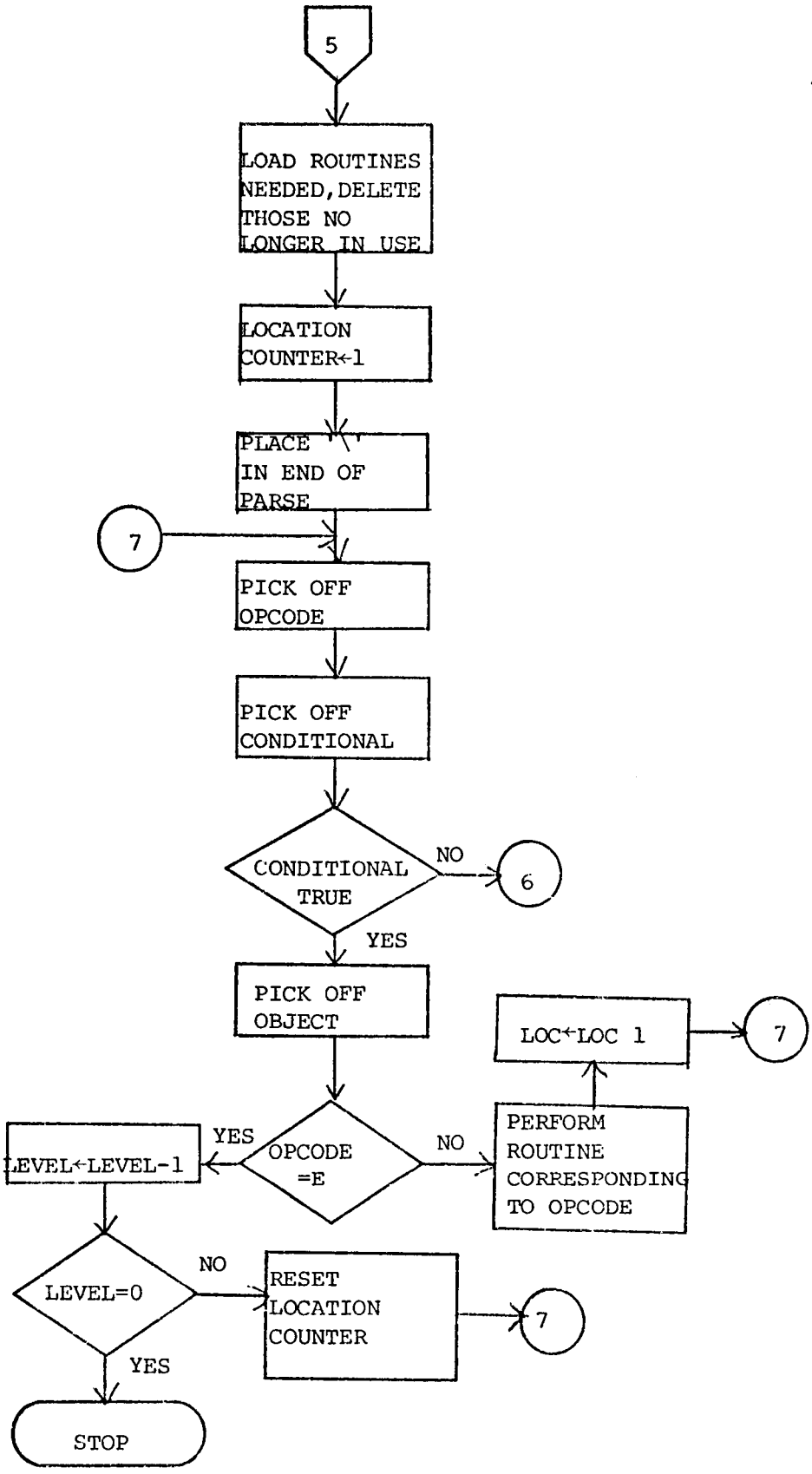


Figure 6 (Continued)

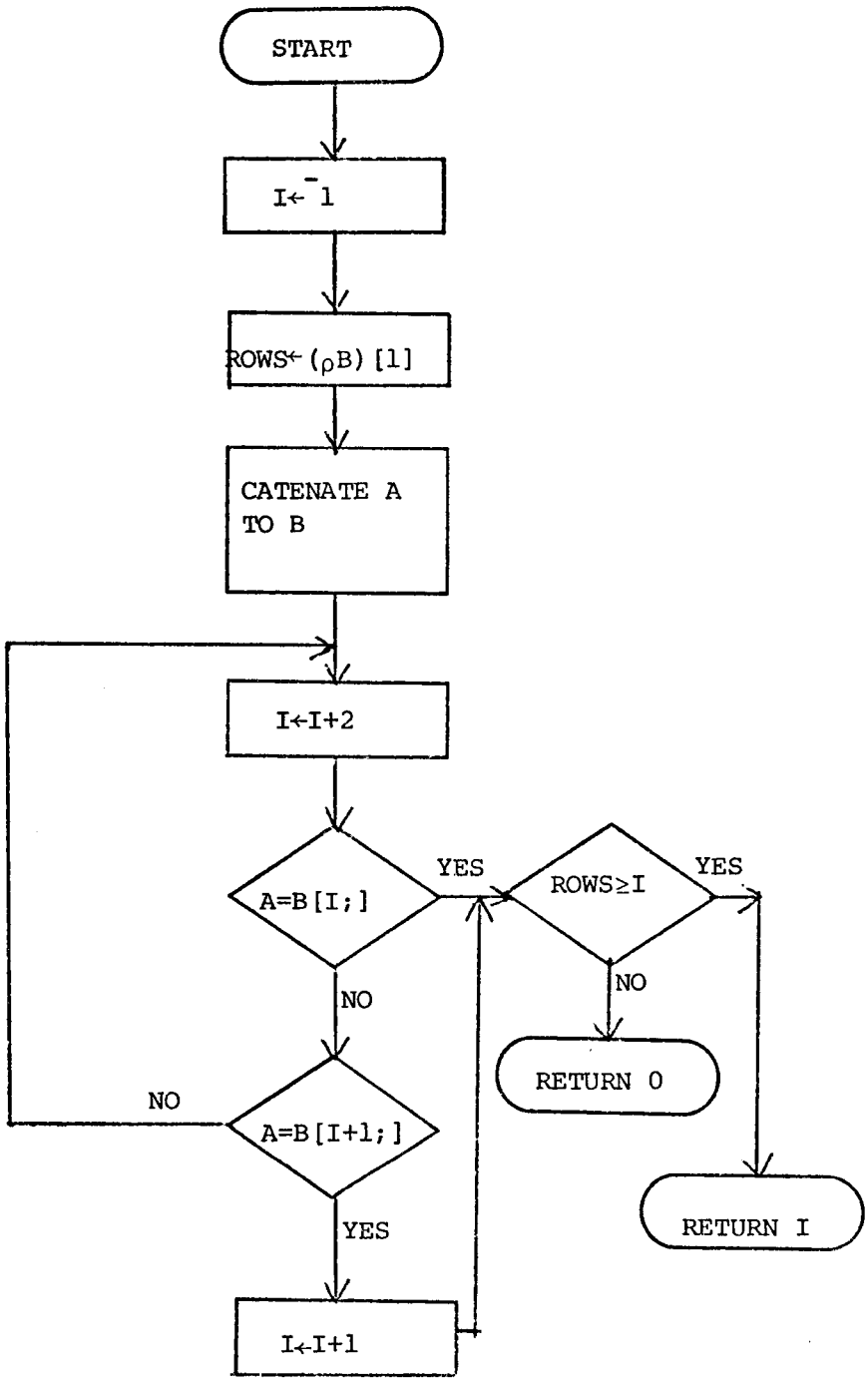


Figure 7

Flowchart for Subroutine LUKUP

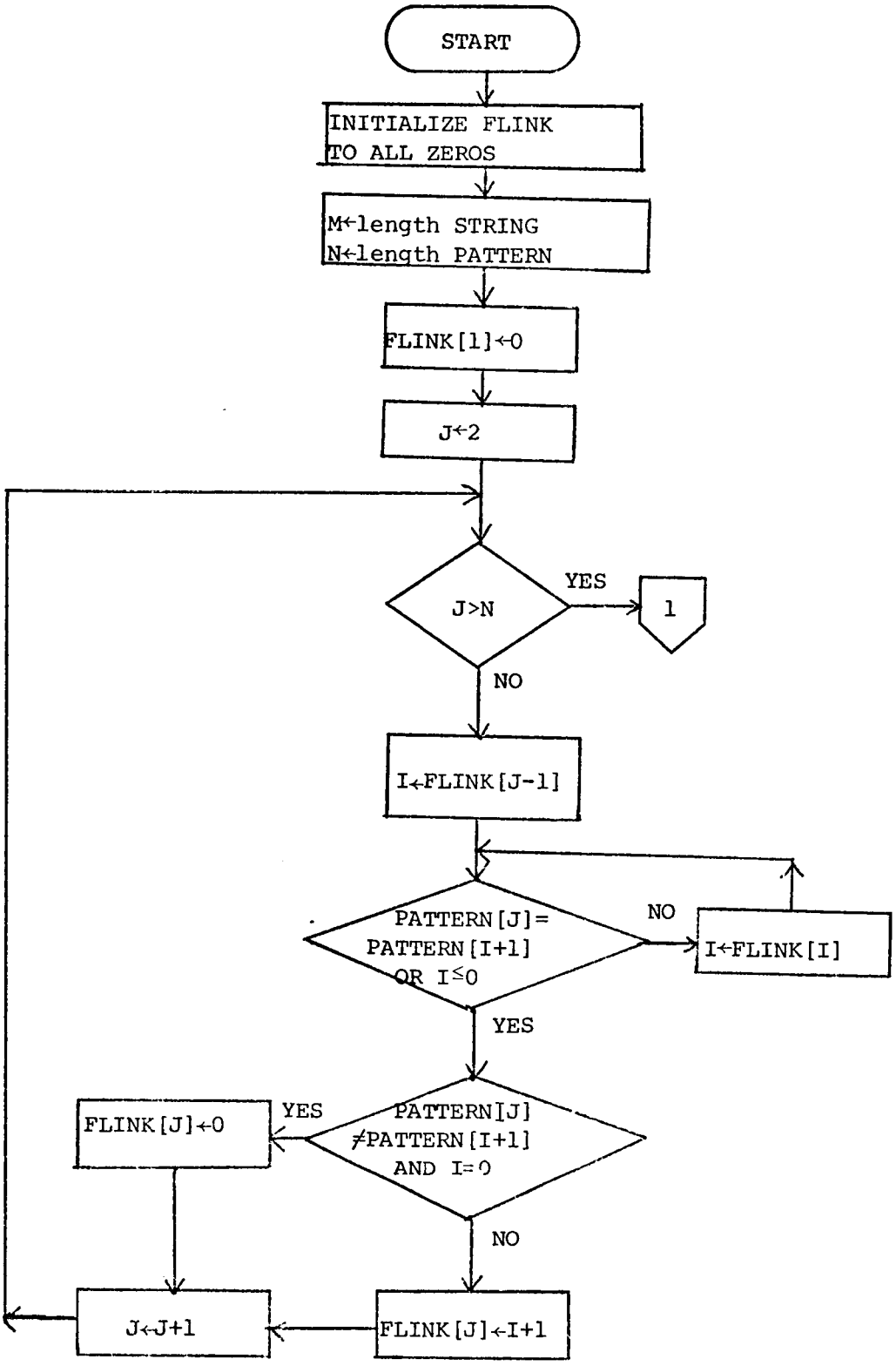


Figure 8

Flowchart for Subroutine IN

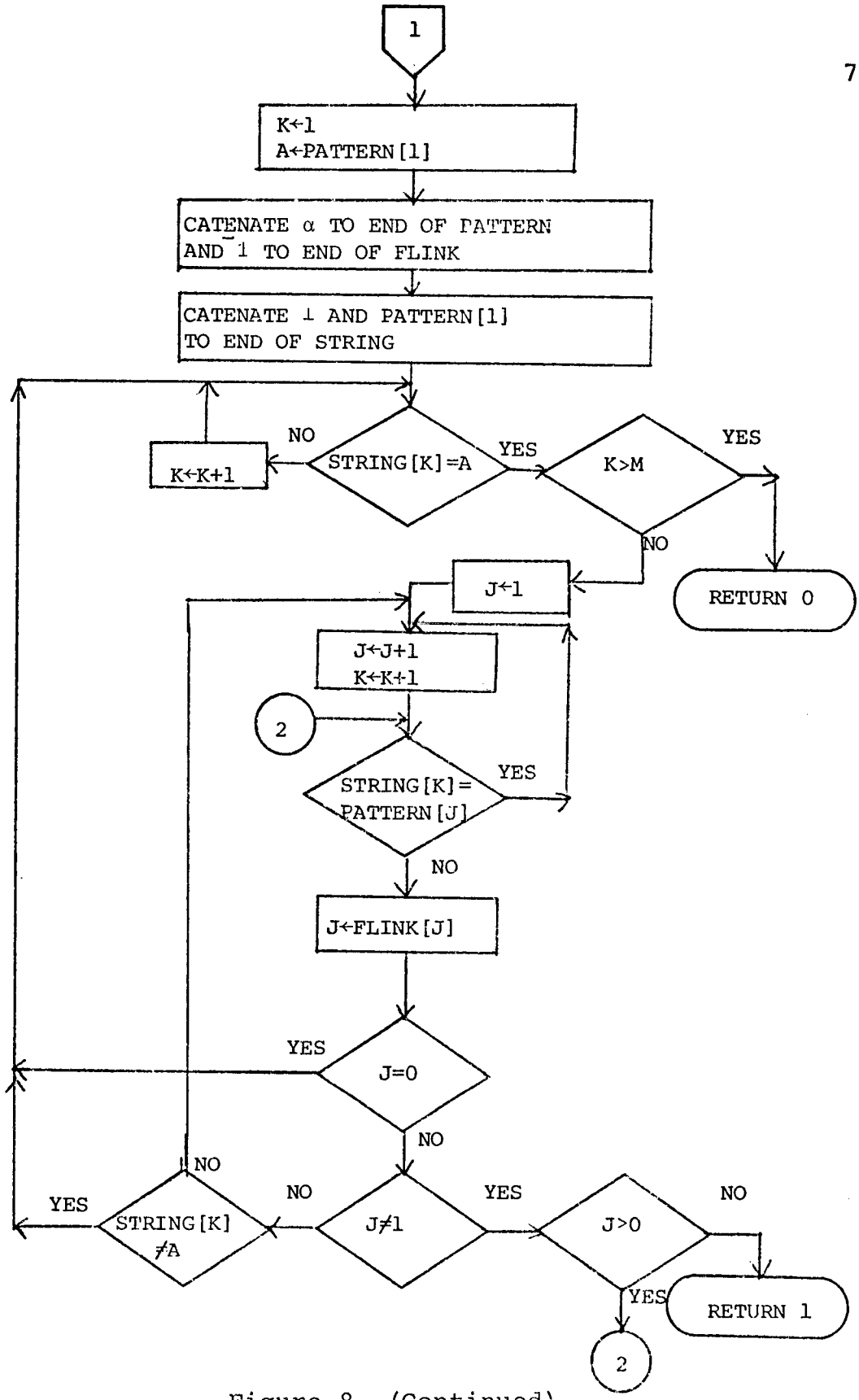


Figure 8 (Continued)

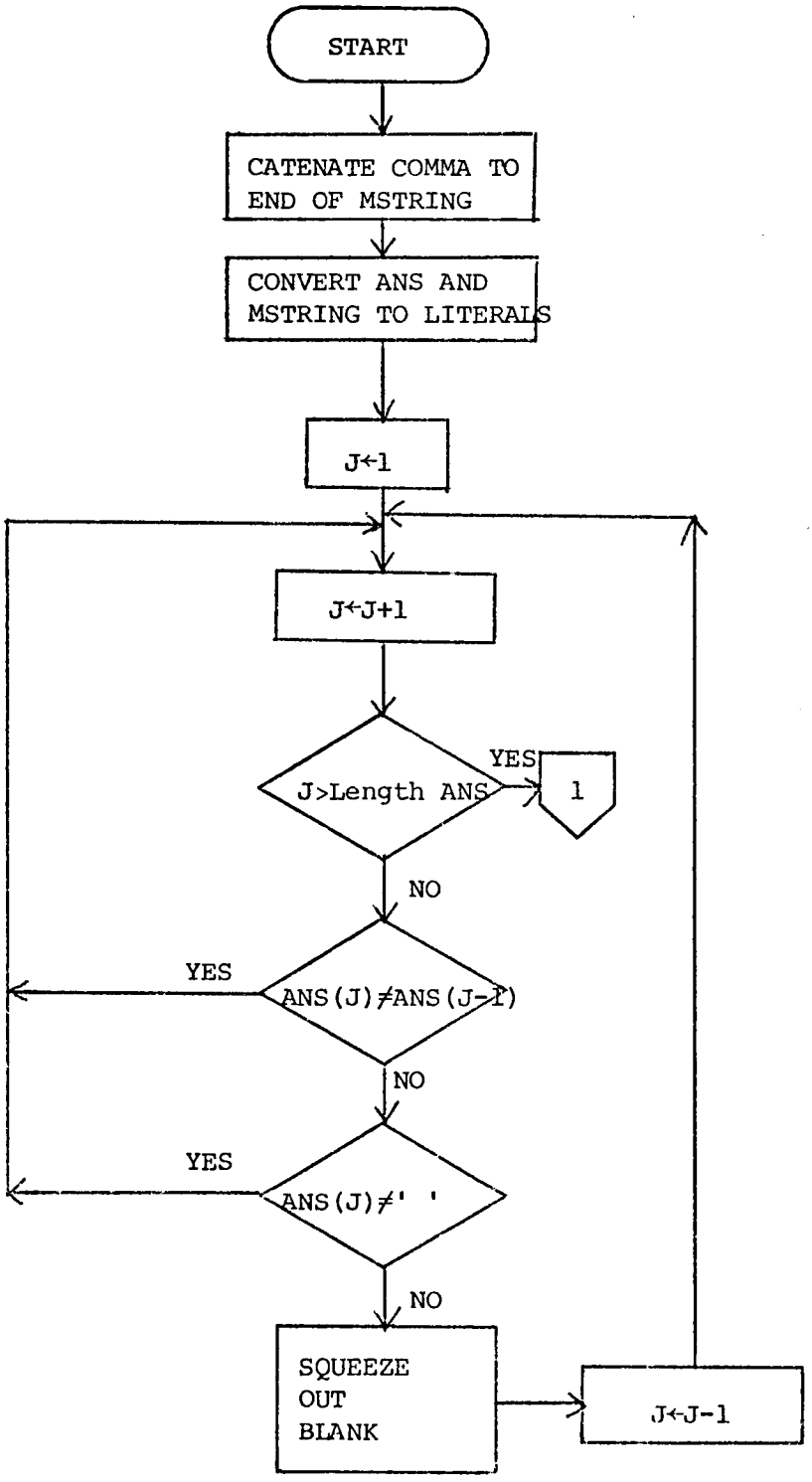


Figure 9

Flowchart of Subroutine MATCH

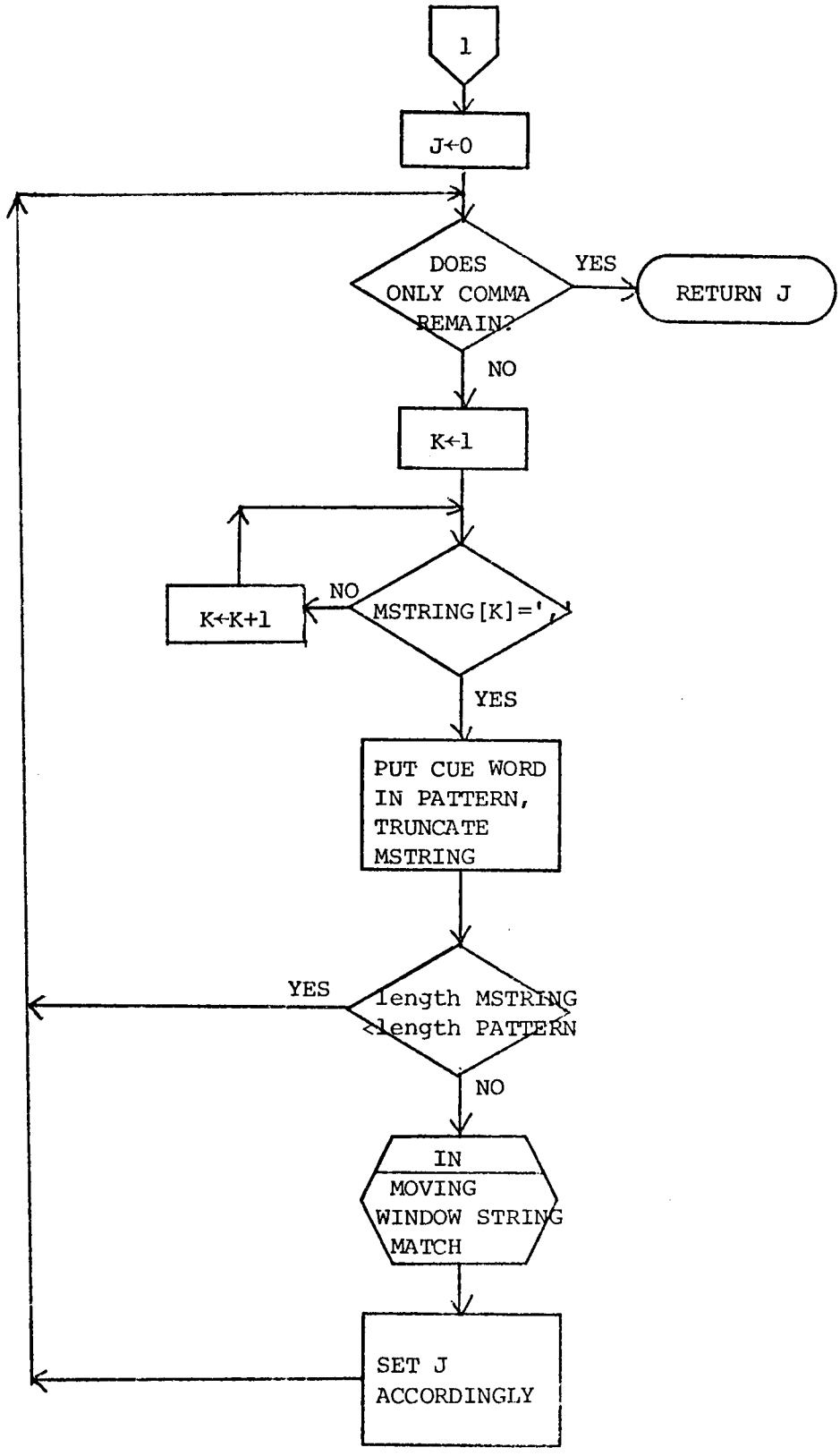


Figure 9 (Continued)

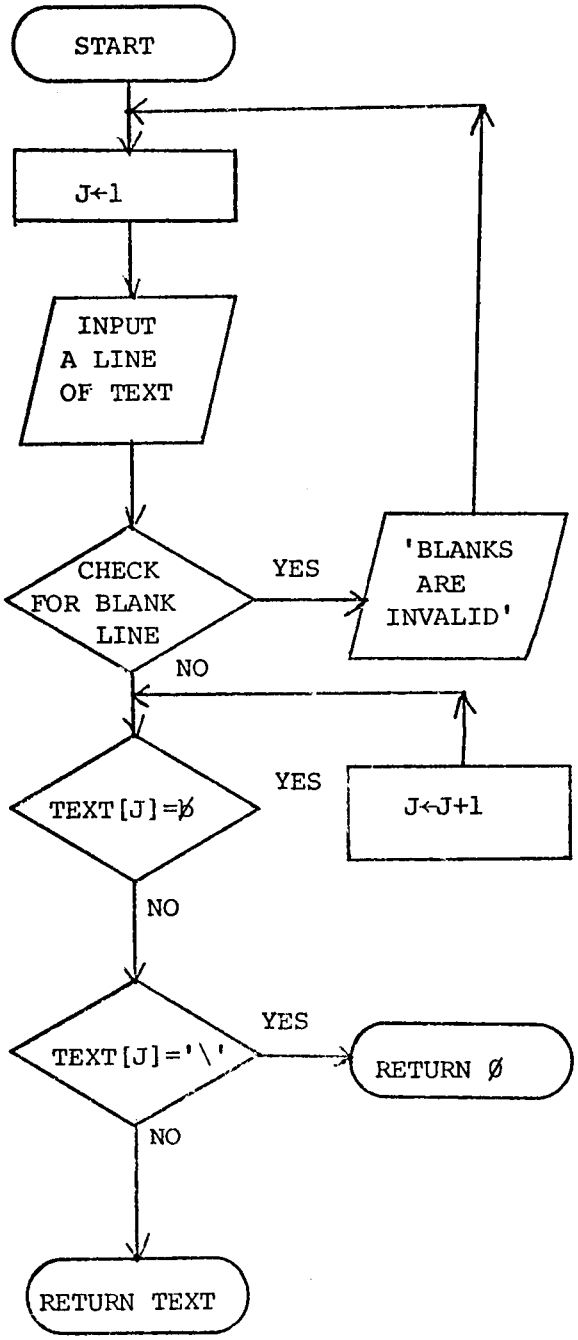


Figure 10

Flowchart for Subroutine ACCEPT

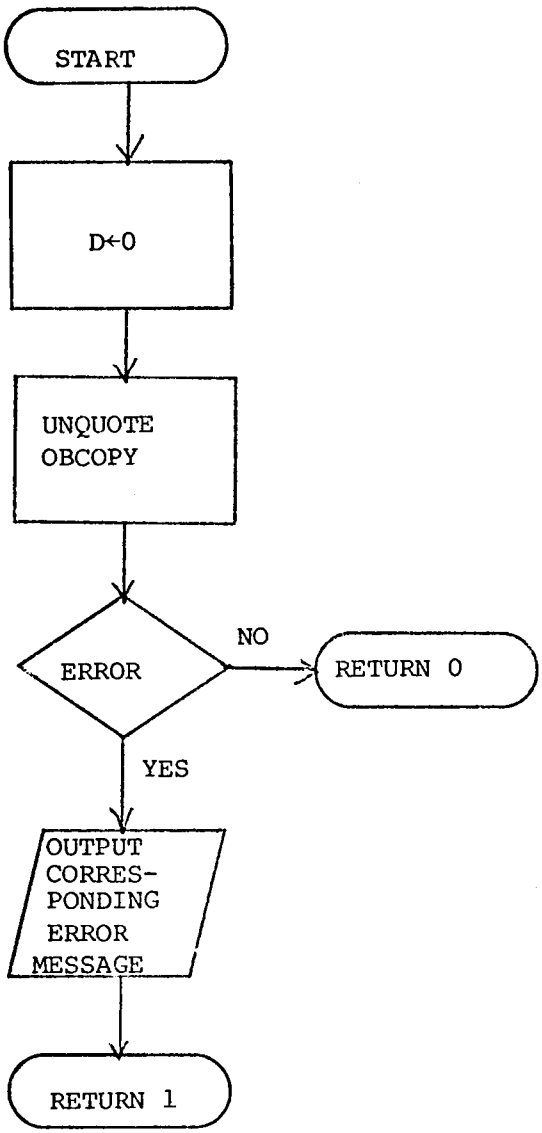


Figure 11

Flowchart for Subroutine CHEK

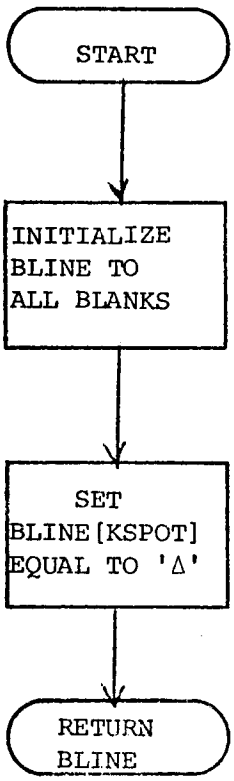


Figure 12

Flowchart for Subroutine BLANK

APPENDIX B

PROGRAM LISTING

This appendix contains a listing of the main program LEX and its subroutines ACCEPT, BLANK, CHEK, IN, LUKUP, MATCH, TIMEON, UNQUOTE, and USEIT.

```
∇ LEX
[1]   ⍝ THIS PROGRAM PERFORMS THE PARSING OF PILOT STATEMENTS
[2]   ⍝ INTO AN INTERMEDIATE FORM THEN, WHEN THE USER
[3]   ⍝ INPUTS A REVERSE SLASH (\), THE CONTROL IS
[4]   ⍝ TRANSFERRED TO THE SECTION OF CODE TO BEGIN THE
[5]   ⍝ INTERPRETATION PHASE.
[6]   ⍝ THE VARIABLE NAMED PARSE CONTAINS THE PARSED
[7]   ⍝ STATEMENT TABLE.  VARTAB IS A TABLE CONSISTING
[8]   ⍝ OF ALL VARIABLE NAMES DEFINED IN THE CURRENT
[9]   ⍝ PROGRAM.  VALTAB AND VACTAB ARE AUXILIARY TABLES
[10]  ⍝ WHICH CONTAIN THE VALUES FOR NUMERIC AND
[11]  ⍝ CHARACTER VARIABLES, RESPECTIVELY.  IF A VARIABLE
[12]  ⍝ IS UNDEFINED, IT RETAINS ITS
[13]  ⍝ INITIALIZATION VALUE, HOWEVER, A WARNING
[14]  ⍝ MESSAGE IS ISSUED TO THE USER.
[15]  ⍝ LABTAB IS A TABLE CONSISTING OF ALL LABEL NAMES
[16]  ⍝ REFERENCED IN THE CURRENT PROGRAM.
[17]  ⍝ ITS AUXILIARY TABLE IS LABVAL, WHICH CONTAINS
[18]  ⍝ THE STATEMENT NUMBERS ASSOCIATED WITH THE LABELS.
[19]  ⍝ INITIALIZE PARSED STATEMENT TABLE, VARIABLE TABLES
[20]  ⍝ LABEL, AND AUXILIARY TABLES.  CLEAR THE VARIABLE CALLED
[21]  ⍝ TEXT, WHICH WILL CONTAIN THE PILOT SOURCE CODE.
[22]  PARSE←50 50⍴' '
[23]  VALTAB←20⍴0
[24]  VACTAB←20 20⍴' '
[25]  LABVAL←20⍴0
[26]  LABTAB←20 9⍴' '
[27]  VARTAB←20 20⍴' '
[28]  TEXT←50 50⍴' '
[29]  ⍝ INITIALIZE POINTER TO THE NEXT AVAILABLE SLOT
[30]  ⍝ IN THE VARIABLE TABLE AND THE CORRESPONDING
[31]  ⍝ AUXILIARY TABLE.
[32]  CHPOINT←0
[33]  ⍝ LOAD THE ROUTINES REQUIRED BY THE PARSING PHASE.
[34]  ⍝ THIS ACTION FACILITATES UTILIZATION OF AN
[35]  ⍝ OVERLAY-STRUCTURE IN ORDER TO CONSERVE STORAGE.
[36]  ⍞')LOAD UNQUOTE'
[37]  ⍞ ')LOAD ACCEPT'
[38]  ⍞')LOAD BLANK'
[39]  ⍞ ')LOAD TIMEON'
[40]  ⍞')LOAD LUKUP'
[41]  ⍝ THIS VARIABLE WILL BE REFERENCED BY THE MATCH
        INSTRUCTION
[42]  LASTANS←''
[43]  ⍝ INITIALIZE FIX-SWITCH TO ZERO.  WHEN THIS VARIABLE
[44]  ⍝ EQUALS 1, THIS INDICATES THAT A FIX IS CURRENTLY
[45]  ⍝ IN EFFECT.
[46]  ⍝ INITIALIZE POINTER INTO NUMERIC
[47]  ⍝ VARIABLE TABLE TO 1.
```

```
[48] NUMPTR←0 ⋄ FSWIC←0
[49] ⍝ SET
[50] SWCH←0
[51] ⍝ INITIALIZE LOCATION COUNTER.
[52] LOC←1
[53] ⍝ INITIALIZE
[54] LPT ←1
[55] ⍝ INITIALIZE POINTER INTO PARSED STATEMENT TABLE.
[56] LOP←1
[57] ⍝ SET LEVEL INDICATOR TO 1.
[58] LEVEL←1
[59] ⍝ NOW BEGIN ACTUAL PARSING PHASE
[60] ⍝ INPUT A LINE OF PILOT CODE FROM USER.
[61] ⍝ THE END OF THE PROGRAM DEFINITION PHASE IS
[62] ⍝ INDICATED BY TYPING A REVERSE SLASH(\)
[63] ⍝ WHEN THE SUBROUTINE ACCEPT SEES THIS
[64] ⍝ SYMBOL, IT RETURNS A VALUE OF 0.
[65] INPUT: LINE←ACCEPT
[66] ⍝ IF '⍝' IS CONTAINED IN THE LINE, OUTPUT AN
[67] ⍝ ERROR MESSAGE.
[68] →('⍝'∈LINE)/ERCOMPU
[69] ⍝ IF THE SUBROUTINE ACCEPT RETURNS 0
[70] ⍝ BEGIN THE INTERPRETIVE PHASE.
[71] →(LINE=0)/INTERP
[72] ⍝ CHECK FOR A FIX STATEMENT.
[73] →((3↑LINE)≠'FIX')/FIXON
[74] ⍝ SCAN FOR THE FIRST NONBLANK CHARACTER
[75] ⍝ FOLLOWING THE WORD FIX
[76] K←4
[77] PLOOP: K←K+1
[78] →(K>⍴LINE)/EFIX
[79] →(LINE[K]=' ')/PLOOP
[80] ⍝ PICK OFF STATEMENT NUMBER OF
[81] ⍝ STATEMENT TO BE CORRECTED.
[82] KT1←K
[83] P2LOP: K←K+1
[84] →(LINE[K]≠' ')/P2LOP
[85] ⍝ SAVE CURRENT POINTER INTO PARSED STATEMENT TABLE AND
[86] ⍝ SET FIX SWITCH TO 1 TO INDICATE A FIX IS IN PROGRESS.
[87] HELD←LOP ⋄ FSWIC←1
[88] ⍝ CONVERT STATEMENT NUMBER TO NUMERIC AND
[89] ⍝ STORE IN LOC2.
[90] LOC2←UNQUOTE(K-⍴LINE)↓((KT1-1)↓LINE)
[91] LOP←LOC2
[91.1] CT←0 ⋄ KTN←0
[91.12] →(LOP>1)/CTGIN
[91.2] CT←1
[91.3] CTGIN: CT←CT+1 ⋄ →(KTN=LOP)/AUSGN
[91.4] →(TEXT[CT;1]='R')/CTGIN
```

```
[91.5] KTN←KTN+1
[91.6] →CTGIN
[91.7] AUSGN: →(TEXT[CT-1;1]='*')/FILABL
[92] ⍝ BE SURE THAT THE STATEMENT NUMBER DOES NOT EXCEED
[93] ⍝ THE CURRENT LOCATION COUNTER.  IF SO OUTPUT MESSAGE
       AND
[94] ⍝ RETURN TO INPUT.
[95] OOON: →(LOC2≤LOC)/ON1
[96] 'STATEMENT NO. IN FIX STATEMENT EXCEEDS RANGE'
[97] →INPUT
[98] ⍝ ALIGN LINE TO CONTAIN THE STMT TO BE CORRECTED.
[99] ⍝ IN THE USUAL MANNER OF FIRST CHARACTER NONBLANK
[100] ON1: K←K+1
[101] →(K>⍴LINE)/AUSOF
[102] →(LINE[K]=' ')/ON1
[103] LINE←(K-1)↓LINE
[104] →FIXON
[104.1] FILABL: KZ←1
[104.2] C22: KZ←KZ+1
[104.3] →(TEXT[CT-1;KZ]≠' ')/C22
[104.4] LLABL←1↓((KZ-1)↑TEXT[CT-1;])
[104.5] PLP←LLABL LUKUP LABTAB
[104.6] LABTAB[PLP;]←9⍴' '
[104.7] LABVAL[PLP]←0
[104.8] →OOON
[105] ⍝ IF THE FIX STATEMENT IS INCORRECT, OUTPUT
[106] ⍝ THE LINE AND AN ERROR MESSAGE.  THEN RETURN
[107] ⍝ TO INPUT.
[108] EFIX: LINE
[109] BLANK K
[110] 'INVALID FIX STATEMENT'
[111] →INPUT
[112] ⍝ IF ONLY BLANKS FOLLOW THE STATEMENT NUMBER IN
[113] ⍝ THE FIX STATEMENT, ENTER A BLANK LINE AT THAT
       LOCATION IN THE
[114] ⍝ PARSED STATEMENT TABLE AND SOURCE CODE.
[115] ⍝ THEN RESTORE THE POINTER INTO THE PARSED STATEMENT
       TABLE,
[116] ⍝ RESET FIX SWITCH TO ZERO AND RETURN TO INPUT.
[117] AUSOF: PARSE[LOP;]←50⍴' '
[118] TEXT[LOP;]←50⍴' '
[119] ⍝ RESTORE POINTER INTO THE PARSED STATEMENT
[120] ⍝ TABLE, RESET FIX SWITCH, AND INPUT
[121] ⍝ ANOTHER LINE.
[122] LOP←HELD
[123] FSWIC←0
[124] →INPUT
[125] ⍝ SET VARIABLE PRESENCE INDICATOR TO ZERO
[126] FIXON:VAA←0
```

```
[127] ⍝ INITIALIZE MATCH VARIABLE INDICATOR TO ZERO.
[128] VMACH←0
[129] K←1
[130] ⍝ INITIALIZE LOGICAL CONDITIONAL TO NULL STRING.
[131] LOGICAL←''
[132] ⍝ INITIALIZE CONDITIONAL TO BLANK.
[133] CONDIT←' '
[134] ⍝ INITIALIZE VARIABLE CALLED NOCOL TO 0.  IT INDICATES
[135] ⍝ THAT NO COLON WAS PRESENT
[136] ⍝ A COLON WAS
[137] ⍝ IN THE PREVIOUS STATEMENT IF IT EQUALS 1.
[138] ⍝ OTHERWISE IT INDICATES
[139] ⍝ A COLON WAS PRESENT.  THIS SWITCH IS
[140] ⍝ NECESSARY TO DETERMINE THE
[141] ⍝ THE CONTEXT OFF AN OP CODE OF ':', WHETHER IT IS A
        CONTINUATION
[142] ⍝ OF THE PREVIOUS STATEMENT OR IF IT IS A REPETITION
        OF THE
[143] ⍝ PREVIOUS OP CODE AND CONDITIONAL
[144] NOCOL←0
[145] ⍝ SCAN FOR LABEL
[146] ⍝ IF NONE IS PRESENT BRANCH AROUND THE LABEL
[147] ⍝ HANDLING ROUTINE.
[148] ⍝ →(LINE[1]≠'*')/NONE
[149] ⍝ PICK OFF THE LABEL.  IF LONGER THAN 8 CHARACTERS,
[150] ⍝ IT IS TRUNCATED AFTER ISSUANCE OF A WARNING MESSAGE.
[151] T1: K←K+1
[152] →(K=9)/LONG
[153] →(LINE[K] ≠ ' ')/T1
[154] ⍝ COPY THE LABEL TABLE AND BLANK OUT THE RIGHTMOST
[155] ⍝ (NINTH) COLUMN.  PERFORM TABLE LOOKUP USING THE
[156] ⍝ SUBROUTINE LUKUP.
[157] OUT1: LABEL←1↓((K-1)↑LINE)
[158] LAA←LABTAB
[159] LAA[;9]←20⍴' '
[160] LABPTR←LABEL LUKUP LAA
[161] 'LABEL IS '; LABEL; 'AT'; LABPTR
[162] ⍝ IF THE LABEL IS NOT CONTAINED IN THE TABLE, ENTER IT
[163] ⍝ AND THE CORRESPONDING STATEMENT NUMBER IN THE LABEL
[164] ⍝ VALUE TABLE.
[165] →(LABPTR=0)/LL
[166] ⍝ IF THE LABEL IS CONTAINED IN THE TABLE, CHECK TO
[167] ⍝ SEE IF COLUMN 9 OF THAT ROW IS BLANK.  IF SO, THE
[168] ⍝ VARIABLE HAS ALREADY BEEN DEFINED SO OUTPUT
[169] ⍝ AN ERROR MESSAGE.
[170] →(LABTAB[LABPTR;9]=' ')/DEFLAB
[171] ⍝ ENTER THE CORRESPONDING STATEMENT NUMBER
[172] ⍝ IN THE LABVAL TABLE AND A BLANK IN
[173] ⍝ COLUMN 9 OF THAT ROW OF LABTAB.
```

```
[174] LABVAL[LABPTR]←LOP
[175] LABTAB[LABPTR;9]←' '
[176] →NOLABEL
[177] ⍝ ERROR MESSAGE FOR PREVIOUSLY DEFINED LABEL
[178] DEFLAB:'ERROR -- PREVIOUSLY DEFINED LABEL'
[179] →INPUT
[180] ⍝ TRUNCATION WARNING FOR LABEL TOO LONG.
[181] LONG:'LABEL IS LONGER THAN 8 CHARACTERS'
[182] 'IT WILL BE TRUNCATED TO EIGHT'
[183] →OUT1
[184] ⍝ ENTER THE LABEL IN THE LABEL TABLE AND PLACE
[185] ⍝ THE CORRESPONDING STATEMENT NUMBER IN
[186] ⍝ LABVAL.
[187] LL: LARTAB[LPT;]←9↑((8↑LABEL],' ')
[188] LABVAL[LPT]←LOP
[189] ⍝ INCREMENT POINTER INTO LABEL TABLE
[190] LPT←LPT+1
[191] ⍝ SCAN TO NEXT NONBLANK CHARACTER
[192] ⍝ THIS WILL BE THE OP CODE.
[193] NOLABEL: K←K+1
[194] →(K>⍴LINE)/ERO
[195] →(LINE[K]=' ')/NOLABEL
[196] ⍝ PICK OFF THE OP CODE,
[197] NONE: OPCODE←LINE[K]
[198] ⍝ IF THE OP CODE IS R, ENTER THAT LINE IN
[199] ⍝ THE SOURCE LISTING, INCREMENT THE LOCATION
[200] ⍝ COUNTER, AND RETURN TO INPUT.
[201] →(OPCODE='R')/FIXUP
[202] ⍝ TEST FOR AN INVALID OP CODE.  IF SO, OUTPUT
[203] ⍝ AN ERROR MESSAGE AND RETURN TO INPUT.
[204] →(×/OPCODE≠'TAYRMCJUNE:')/ERZ
[205] ⍝ IF THE PREVIOUS STATEMENT DID NOT HAVE A COLON
[206] ⍝ AND THE CURRENT OP CODE IS NOT ':', OUTPUT
[207] ⍝ AN ERROR MESSAGE.
[208] ⍝ ON THE OTHER HAND, IF THE PREVIOUS STATEMENT
[209] ⍝ LACKED A COLON AND THE CURRENT OP CODE IS A
[210] ⍝ COLON, CATENATE THE CURRENT OBJECT TO THE
[211] ⍝ PREVIOUS ONE.
[212] →((NOCOL=1)∧(OPCODE≠':'))/ER7
[213] →((NOCOL=1)∧(OPCODE=':'))/CAT
[214] ⍝ IF EITHER THE OP CODE IS NOT A COLON OR
[215] ⍝ IF THE NOCOLON INDICATOR IS 1 GO TO T2.
[216] ⍝ OTHERWISE, COPY THE OP CODE AND CONDITIONAL
[217] ⍝ FROM THE PREVIOUS STATEMENT IN THE PARSED
[218] ⍝ STATEMENT TABLE.  THEN ENTER THE LINE
[219] ⍝ IN THE SOURCE CODE LIST AND
[220] ⍝ PREPARE TO EXAMINE THE OBJECT.
[221] →((OPCODE=':')∧(NOCOL=0))/T2
[222] →(LOC<1)/ERZ
```

```
[223] ⍝ COPY THE OP CODE AND CONDITIONAL
[224] ⍝ FROM THE PREVIOUS STATEMENT.
[225] OPCODE←PARSE[LOP-1;1]
[226] →(PARSE[LOP-1;2]='∩∪⊂')/NCD,SIMC,COMC
[227] SIMC: CONDIT←PARSE[LOP-1;3]
[228] →NCD
[229] COMC: Z←2
[230] BRA3: Z←Z+1
[231] →(PARSE[LOP-1;Z]≠'⊃')/BRA3
[232] CONDIT←2↓((K-1)↑PARSE[LOP-1;])
[233] ⍝ ENTER THE LINE INTO THE SOURCE CODE TABLE.
[234] NCD: TEXT[LOC;]←50↑(LINE,50⍴' ')
[235] →COLON
[236] ⍝ ENTER THE LINE IN THE SOURCE CODE LIST
[237] T2: TEXT[LOC;]←50↑(LINE,50⍴' ')
[238] ⍝ IF THE OP CODE IS AN M, EXAMINE THE NEXT
[239] ⍝ CHARACTER TO CHECK FOR A MATCH VARIABLE
[240] ⍝ INSTRUCTION.  IF SO,
[241] →(OPCODE≠'M')/NOMACH
[242] LINE[K+1]
[243] →(LINE[K+1]≠'V')/NOMACH1
[244] ⍝ SET THE VARIABLE MATCH SWITCH TO 1.
[245] VMACH←1
[246] K←K+1
[247] →NOMACH
[248] ⍝ SET VARIABLE MATCH SWITCH TO ZERO.
[249] NOMACH1:VMACH←0
[250] ⍝ ELIMINATE BLANKS WHILE SCANNING FOR
[251] ⍝ CONDITIONAL OR COLON.
[252] ⍝ IF THE LINE IS EXHAUSTED, OUTPUT ERROR
[253] ⍝ MESSAGE.
[254] NOMACH: K←K+1
[255] →(K>⍴LINE)/ERO
[256] →(LINE[K]='YN( :')/ACON,ACON,PACON,NOMACH,COLON
[257] ⍝ OUTPUT ERROR MESSAGE
[258] ERO: LINE
[259] BLANK K
[260] 'INVALID CONDITIONAL IN STATEMENT -- TRY AGAIN'
[261] →INPUT
[262] ⍝ CATENATE OBJECT TO PREVIOUS LINE IF
[263] ⍝ NO COLON WAS FOUND THERE.
[264] CAT: K←⍴DDT
[265] LINE←DDT,LINE
[266] →T2
[267] ⍝ PERFORM FIX -- REPLACE A LINE
[268] ⍝ AS INDICATED BY THE USER
[269] FIXUP: TEXT[LOC;]←50↑(LINE,50⍴' ')
[270] LOC←LOC+1
[271] →INPUT
```

```
[272] ⍝ ERROR MESSAGE FOR ILLEGAL OP CODE.
[273] ERZ: LINE
[274] BLANK K
[275] 'ERROR -- ILLEGAL OP CODE'
[276] →INPUT
[277] ⍝ OUTPUT PRECEDING STATEMENT AND A MESSAGE
[278] ⍝ INDICATING NO COLON WAS FOUND.
[279] ER7: 'NO COLON FOUND IN PRECEDING STATEMENT'
[280] TEXT[LOC-1;]
[281] →INPUT
[282] ⍝ SET CONDITIONAL FOR SIMPLE CONDITIONAL
[283] ACON: CONDIT←LINE[K]
[284] →T4
[285] PACON: KH←K-1
[286] CONDIT←LINE[K]
[287] ⍝ RESOLVE LOGICAL EXPRESSION
[288] ⍝ IF NO RIGHT PARENTHESIS APPEARS,
[289] ⍝ OUTPUT ERROR MESSAGE AND RETURN TO INPUT.
[290] T3: →(K≥⍴LINE)/ERLOG
[291] K←K+1
[292] →(LINE[K]≠')')/T3
[293] LOGICAL←(K-⍴LINE)↓(KH↓LINE)
[294] →T4
[295] ⍝ ERROR MESSAGE FOR INVALID LOGICAL EXPRESSION.
[296] ERLOG:LOGICAL
[297] 'INVALID LOGICAL EXPRESSION OR MISSING )'
[298] →INPUT
[299] ⍝ SCAN FOR A COLON.  IF NONE IS PRESENT, SET
[300] ⍝ NOCOLON TO 1.
[301] T4: K←K+1
[302] →(K>⍴LINE)/NOCOLON
[303] →(LINE[K]=' :')/T4,COLON
[304] ⍝ IF A CHARACTER OTHER THAN A BLANK OR COLON
[305] ⍝ IS ENCOUNTERED OUTPUT APPROPRIATE ERROR
[306] ⍝ MESSAGE AND RETURN TO INPUT.
[307] →ERCOL
[308] ⍝ THIS ROUTINE IS ACCESSED ONLY IF THE OPCODE IS
[309] ⍝ J (JUMP)
[310] JUMRTN: K←0
[311] ⍝ SCAN OBJECT FOR LABEL.  IF NONE IS PRESENT
[312] ⍝ OUTPUT APPROPRIATE ERROR MESSAGE.
[313] TST8: K←K+1
[314] →(K>⍴OBJECT)/ERJUMP
[315] →(OBJECT[K]=' *')TST8,LOOKUP
[316] →(OPCODE='U')/OUT3
[317] ⍝ ERROR MESSAGE FOR MISSING LABEL
[318] ERJUMP: 'ERROR: NO LABEL IN JUMP OR USE STATEMENT'
[319] LINE
[320] BLANK K
```

```
[321] →INPUT
[322] ⍝ ERROR MESSAGE FOR UNDEFINED VARIABLE
[323] ERLAB: LINE
[324] 'ERROR UNDEFINED LABEL'
[325] →INPUT
[326] ⍝ SET UP TO INDICATE NO COLON WAS PRESENT IN THE
[327] ⍝ CURRENT STATEMENT.
[328] NOCOLON: NOCOL←1
[329] DDT←LINE
[330] LOC←LOC+1
[331] →INPUT
[332] ⍝ THIS SECTION IS ACCESSED ONLY WHEN A LABEL IS
[333] ⍝ SPOTTED.  IT PICKS OFF THE LABEL AND PERFORMS
[334] ⍝ TABLE LOOKUP; IF THE LABEL IS NOT CONTAINED
[335] ⍝ IN THE TABLE, ALLOWANCE IS MADE FOR FORWARD
[336] ⍝ REFERENCE BY SETTING THE OBJECT EQUAL
[337] ⍝ TO THE ELEMENT OF THE LABEL VALUE TABLE
[338] ⍝ CORRESPONDING TO THE NEW ENTRY AND COLUMN 9
[339] ⍝ OF THAT ROW IS SET EQUAL TO 1 TO INDICATE
[340] ⍝ THAT IT IS UNDEFINED.  OTHERWISE, THE OBJECT
[341] ⍝ IS THE ENTRY IN THE LABEL VALUE TABLE
[342] ⍝ CORRESPONDING TO THE LOCATION OF THE LABEL
[343] ⍝ IN LABTAB.
[344] LOOKUP: KT←K
[345] LOOK2: K←K+1
[346] →(K>⍴OBJECT)/FOUNDIT
[347] →(OBJECT[K]≠' ')/LOOK2
[348] FOUNDIT: LABEL←KT↓((K-1)↑OBJECT)
[349] PTR←LABEL LUKUP LABTAB
[350] →(PTR=0)/ERLABA
[351] OBJECT←LABVAL[PTR]
[352] ⍝ FOR A USE STATEMENT, THE POINTER MUST BE
[353] ⍝ PREFIXED BY A STAR TO DIFFERENTIATE A USE
[354] ⍝ FROM A JUMP LABEL.
[355] →(OPCODE≠'U')/OUT3
[356] OBJECT←'*';OBJECT
[357] →OUT3
[358] ⍝ ARRANGE FOR FORWARD REFERENCE
[359] ERLABA: LABTAB[LPT;]←9↑(↑LABEL,8⍴' ');' '
[360] OBJECT←'LABVAL[';LPT;']'
[361] LABTAB[LPT;9]←'1'
[362] LPT←LPT+1
[363] →(OPCODE≠'U')/OUT3
[364] OBJECT←'*';OBJECT
[365] →OUT3
[366] ⍝ IF MATCH-SWITCH IS ON, GO TO SETVMACH.
[367] COLON: →(VMACH=1)/SETVMACH
[368] ⍝ OTHERWISE PICK OFF THE OBJECT
[369] COLO2: OBJECT←K↓LINE
```

```
[370] ⍝ INITIALIZE POINTER INTO OBJECT TO 1
[371] KEEPON: K←1
[372] ⍝ FOR OP CODE OF J OR U, BRANCH TO JUMRTN
[373] →(OPCODE='JU')/JUMRTN,JUMRTN
[374] ⍝ INITIALIZE SWITCH TO 0.  IF THIS VARIABLE
[375] ⍝ STILL EQUALS ZERO AFTER THE OBJECT HAS BEEN
[376] ⍝ SCANNED FOR BLANKS AND VARIABLE NAMES, THE
[377] ⍝ OBJECT IS SET TO THE SPECIAL CHARACTER '⍝'.
[378] MARK←0
[379] ⍝ SCAN FOR BLANKS AND VARIABLE NAMES.
[380] ⍝ IF POINTER EXCEEDS THE LENGTH OF THE OBJECT
[381] ⍝ BRANCH TO ROUTINE TO TEST FOR BLANK OBJECT.
[382] T5: →(K>⍴OBJECT)/TSTMARK
[383] ⍝ IF OBJECT (K) IS BLANK, INCREMENT POINTER
[384] ⍝ AND TEST THE NEXT CHARACTER.  OTHERWISE
[385] ⍝ INDICATE NONBLANK OBJECT BY SETTING
[386] ⍝ MARK EQUAL TO 1, THEN SCAN FOR
[387] ⍝ VARIABLE NAME.  IF A VARIABLE
[388] ⍝ NAME IS ENCOUNTERED, BRANCH TO
[389] ⍝ APPROPRIATE ROUTINE FOR HANDLING
[390] →(OBJECT[K]=' ')/T6
[391] MARK←1
[392] →(OBJECT[K]='∩∪')/VARST,VARN
[393] T6:K←K+1
[394] →T5
[395] ⍝ INDICATE THAT A '⍝' WAS CONTAINED
[396] ⍝ IN THE CURRENT STATEMENT.
[397] ERCOMPU:'⍝ IS AN INVALID CHARACTER'
[398] →INPUT
[399] ⍝ ROUTINE FOR RESOLUTION OF VARIABLE MATCH
[400] SETVMACH: K←K+1
[401] ⍝ SCAN FOR VARIABLE -- IF NOT FOUND
[402] ⍝ OUTPUT ERROR MESSAGE
[403] ⍝ SCAN TO COMMA, THEN PICK OFF THE VARIABLE
[404] ⍝ AND RESOLVE IT.
[405] →(K>⍴LINE)/ERVE
[406] →((LINE[K]≠'∪')∧(LINE[K]≠'∩'))/SETVMACH
[407] K2←K
[408] ALAN: K←K+1
[409] →(LINE[K]≠',')/ALAN
[410] VAR←(K-1+⍴LINE)↓((K2-1)↓LINE)
[411] VARIND←VAR[1]
[412] VAR←1↓VAR
[413] 'VAR',VAR
[414] PT3←VAR LUKUP VARTAB
[415] →(VARIND='∩∪')/CV,NV
[416] ⍝ ERROR MESSAGE FOR INVALID OR UNDEFINED VARIABLE NAME.
[417] ERVE: LINE ∘ BLANK K
[418] 'INVALID OR UNDEFINED VARIABLE NAME'
```

```
[419] →INPUT
[420] ⍝ ROUTINE TO SET LAST ANSWER (LASTANS) EQUAL TO THE
[421] ⍝ CHARACTER VARIABLE CONTAINED AT THE LOCATION
[422] ⍝ IN VACTAB POINTED TO BY PT3.
[423] CV: →(PT3=0)/ERVE
[424] LASTANS←VACTAB[PT3;]
[425] VMACH←0 ⋄ →COLO2
[426] NV: →(PT3=0)/ERVE
[427] LASTANS←,VALTAB[PT3]
[428] VMACH←0 ⋄ →COLO2
[429] ⍝ ERROR MESSAGE FOR MISSING COLON
[430] ERCOL: 'MISSING COLON IN STATEMENT'
[431] LINE
[432] BLANK K
[433] →INPUT
[434] ⍝ ERROR MESSAGE FOR INVALID LABEL IN JUMP OR
[435] ⍝ USE STATEMENT
[436] ERJU: 'JUMP OR USE STATEMENT HAS INVALID LABEL'
[437] OBJECT
[438] BLANK K
[439] →INPUT
[440] ⍝ ROUTINE TO PERFORM NUMERIC VARIABLE RESOLUTION.
[441] ⍝ SET VARIABLE PRESENCE SWITCH EQUAL TO 1
[442] ⍝ PICK OFF VARIABLE NAME PERFORM TABLE LOOKUP
[443] VARN: KST←K
[444] VAA←1
[445] VA1: K←K+1
[446] →(K>⍴OBJECT)/T9
[447] →(OBJECT[K]≠' ')/VA1
[448] T9: NUMVAR←(K-⍴OBJECT)↓((KST-1)↓OBJECT)
[449] 'NUMVAR IS', NUMVAR
[450] DAM: NUMPT←NUMVAR LUKUP VARTAB
[451] NUMPT
[452] →(OPCODE='A')/OPAN
[453] →(NUMPT>0)/FINE1
[454] →((1↑NUMVAR)≠'∪')/DAM1
[455] →(OPCODE='ACTJNUY')/OPAN,EVR,EVR,WARN,EVR,WARN,EVR
[456] 'INVALID USE OF VARIABLE IN STATEMENT ' ⋄ →INPUT
[457] EVR: OBJECT ⋄ BLANK K
[458] 'WARNING -- UNDEFINED VARIABLE ∪',NUMVAR
[459] CHPOINT←CHPOINT+1
[460] NUMPT←CHPOINT
[461] VARTAB[NUMPT,]←20↑(NUMVAR,20⍴' ')
[462] FINE1: OBJECT←((KST-1)↑OBJECT),'⍺VALTAB[',NUMPT,']⍺',
      (((K-1)-⍴OBJECT)↑OBJECT)
[463] ⍝ IF AN ← OCCURS AFTER THE VARIABLE
[464] ⍝ SET UP VARIABLE IN A DEEPER LEVEL OF QUOTES.
[465] TIR←OBJECT⍳'←'
[466] →((TIR<⍴OBJECT)∧(TIR>K))/APPLE
```

```
[467] →T6
[468] DAM1: NUMVAR←1↓NUMVAR
[469] →DAM
[470] APPLE: OBJECT←((OBJECT⍳'α')↑OBJECT);'''';((OBJECT⍳'α')
      -ρOBJECT)↑OBJECT
[471] OZ←(OBJECT⍳'α')↓OBJECT
[472] OZ1←(OBJECT⍳'α')↑OBJECT
[473] OZ←(((OZ⍳'α')-1↑OZ);'''';((OZ⍳'α')-(1+ρOZ))↑OZ
[474] OBJECT←OZ1;OZ
[475] →T6
[476] ⍝ ROUTINE TO PERFORM CHARACTER VARIABLE RESOLUTION
[477] ⍝ VERY SIMILAR TO DESCRIPTION OF NUMERIC VARIABLES
[478] ⍝ RESOLUTION DESCRIBED ABOVE
[479] ⍝ PICK OFF AND RESOLVE CHARACTER VARIABLE
[480] ⍝
[481] VARST:KST←K
[482] VAA←1
[483] VA2: K←K+1
[484] →(K>ρOBJECT)/T7
[485] →(OBJECT[K]≠' ')/VA2
[486] T7: CHVAR←(K-ρOBJECT)↓((KST-1)↓OBJECT)
[487] 'CHVAR '; CHVAR
[488] DAN:CHPTR←CHVAR LUKUP VARTAB
[489] →(OPCODE='A')/OPAC
[490] →(CHPTR>0)/OK
[491] →((1↑CHVAR)='∩')/DAN1
[492] →(OPCODE='ACTJNUY')/OPAC,OPCC,ERVAR,WARN,ERVAR,WARN,
      EVAR
[493] ⍝
[494] 'INVALID USE OF VARIABLE IN STATEMENT' ∘ →INPUT
[495] ⍝ WARNING MESSAGE FOR UNDEFINED CHARACTER VARIABLE
[496] ERVAR: OBJECT ∘ BLANK K
[497] 'WARNING -- UNDEFINED VAR ∩';CHVAR
[498] DEFIT: CHPOINT←CHPOINT+1
[499] CHPTR←CHPOINT
[500] VARTAB[CHPTR;]←20↑(CHVAR,20ρ' ')
[501] ⍝ PARSE POINTER TO VARIABLE IN OBJECT
[502] OK: CBJECT←((KST-1)↑OBJECT);'αVACTAB[';CHPTR;';]α';
      (((K-1)-ρOBJECT)↑OBJECT)
[503] →T6
[504] DAN1: CHVAR←1↓CHVAR
[505] →DAN
[506] WARN: 'INVALID USE OF VARIABLE IN JUMP OR USE STATEMENT'
[507] →INPUT
[508] ⍝ SET UP TO INPUT A NUMERIC VALUE
[509] OPAN: CHPOINT←CHPOINT+1
[510] NUMPT←CHPOINT
[511] VARTAB[NUMPT;]←20↑(NUMVAR,20ρ' ')
[512] FINE2: OBJECT←'VALTAB[';NUMPT;']←⍬'
```

```
[513] →OUT3
[514] ⍝ SET UP TO INPUT A CHARACTER VALUE
[515] OPAC: CHPOINT←CHPOINT+1
[516] CHPTR←CHPOINT
[517] VARTAB[CHPTR;]←20↑(CHVAR,20ρ' ')
[518] FINE4: OBJECT←'VACTAB[';CHPTR;';]←20ρ(⎕,20ρ'' '')'
[519] →T6
[520] OPCC: 'ERROR -- YOU ATTEMPTED TO USE A CHARACTER VALUE
      IN A COMPUTATION'
[521] →INPUT
[522] ⍝ BUILD A PARSED STATEMENT TABLE
[523] TSTMK2: OBJECT←OBJECT2
[524] ⍝ IF THE OBJECT IS ALL BLANKS
[525] ⍝ ADJUST OBJECT TO SPECIAL CHARACTER ⍝
[526] TSTMARK: →(MARK≠0)/OUT3
[527] ⍝ A U STATEMENT MUST NOT HAVE A BLANK OBJECT.  IF SO,
      OUTPUT
[528] ⍝ AN ERROR MESSAGE AND RETURN TO INPUT.
[529] →(OPCODE='U')/ERPGM
[530] OBJECT←'⍝'
[531] ⍝ IF NO α IS CONTAINED IN THE OBJECT, NO VARIABLE
[532] ⍝ WAS PRESENT.  OTHERWISE, PERFORM SPECIAL QUOTING
      AROUND
[533] ⍝ THE VARIABLE.
[534] OUT3: →((OBJECTι'α')>ρOBJECT)/LENNY
[535] FRO←0 ⋄ BAK←0
[536] ZK←0
[537] ⍝ SICH INDICATES WHETHER TO PARSE '; OR ;'
[538] SICH←1
[539] ZONKER: ZK←ZK+1
[540] →((OBJECTι'α')=1)/BAKK
[541] FRO←1
[542] BAKK:→((OBJECTι'α')=ρOBJECT)/NOBAK
[543] BAK←1
[544] NOBAK:→((OBJECTι'α')>ρOBJECT)/DUNNE
[545] PO←OBJECTι'α'
[546] →(SICH<0)/REV
[547] OBJECT←((PO-1)↑OBJECT);'''';(PO-ρOBJECT)↑OBJECT
[548] SICH←-SICH
[549] →NOBAK
[550] REV: OBJECT←((PO-1)↑OBJECT);';''';(PO-ρOBJECT)↑OBJECT
[551] SICH←-SICH
[552] →NOBAK
[553] DUNNE: →(FRO<1)/LES
[554] ⍝ PUT QUOTE BEFORE OBJECT
[555] OBJECT←'''';OBJECT
[556] LES: →(BAK<1)/LENNY
[557] ⍝ PLACE QUOTE AFTER OBJECT.
[558] OBJECT←OBJECT;''''
```

```
[559] ⍝ SCAN COLUMN 9 OF LABEL TABLE.  IF ALL BLANKS, ALL THE
[560] ⍝ LABELS HAVE BEEN DEFINED.
[561] ⍝ IF NOT, PICK OUT
[562] ⍝ THE PARTICULAR LABEL THAT WAS UNDEFINED AND
[563] ⍝ OUTPUT ERROR MESSAGE AND TERMINATE
[564] LENNY: →(∧/LABTAB[;9]≠' ')/CHEKLAB
[565] →((OPCODE='C')∧(VAA=1))/UNCOLA
[566] NOTQUITE: →(OPCODE≠'C')/ALMOST
[567] ⍞')LOAD CHEK'
[568] SIGNL←CHEK OBJECT
[569] →(SIGNL=0)/ALMOST
[570] TEXT[LOC;]←50⍴' '
[571] →INPUT
[572] ALMOST: →(CONDIT=' YN(')/NOC,NEX,NEX,LOG
[573] ⍝ BRANCH TO APPROPRIATE ROUTINE TO PARSE CONDITIONAL
[574] ⍝ ROUTINE FOR NO CONDITIONAL
[575] NOC:CD1←OPCODE,'∩'
[576] →ON
[577] ⍝ INDICATE WHICH LABELS ARE UNDEFINED
[578] CHEKLAB: BLK←LABTAB[;9]⍳' '
[579] 'LABEL ';LABTAB[BLK;];' IS UNDEFINED'
[580] 'PROCESSING TERMINATED DUE TO ERROR'
[581] →0
[582] ⍝ IF OPCODE = C AND VARIABLE IS PRESENT
[583] ⍝ REMOVE OUTERMOST QUOTES.  TEST
[584] ⍝ FOR ERRORS WITH SUBROUTINE CHEK.
[585] ⍝ IF SIGNL IS 0, NO ERRORS WERE PRESENT.
[586]
[587] UNCOLA:OBJECT←UNQUOTE OBJECT
[588] →NOTQUITE
[589] ERPGM: 'MISSING PROGRAMNAME IN A USE STATEMENT -- TRY
      AGAIN'
[590] ⍝ AT THIS POINT WE ARE READY TO RESOLVE THE CONDITIONAL
[591] ⍝ ROUTINE FOR SIMPLE (Y OR N) CONDITIONAL
[592] NEX: CD1←OPCODE,'∪', CONDIT
[593] →ON
[594] ⍝ ROUTINE FOR LOGICAL CONDITIONAL
[595] LOG: CD1←OPCODE,'⊂''', LOGICAL,''''
[596] ⍝ IF OP CODE IS M, CATENATE A COMMA TO THE END OF THE
[597] ⍝ OBJECT AND THEN GO TO SETPAR.  OTHERWISE ENCLOSE
[598] ⍝ THE OBJECT IN ANOTHER LEVEL OF QUOTES AND ENTER THE
[599] ⍝ PARSED STATEMENT IN THE PARSED STATEMENT TABLE
[600] ⍝ IF OBJECT WAS ALL BLANK OR IF OP CODE IS C, J, OR IF
[601] ⍝ OP CODE IS A, T, Y, OR N AND A VARIABLE WAS PRESENT
[602] ⍝ THEN GO TO SETPAR.
[603] ON:→((OBJECT='∩')∨(OPCODE='C')∨(OPCODE='J'))/SETPAR
[604] →(OPCODE='U')/SETPAR
[605] →((OPCODE='A')∧(VAA=1))/SETPAR
[606] →(OPCODE='M')/SETMAR
```

```
[607] →((∨/(OPCODE='TYN'))∧(VAA=1))/SETPAR
[608] PARSE⌊LOP;⌋←50↑(CD1,'⊃''';OBJECT;'''',(50ρ' '))
[609] ⍝ INCREMENT LOCATION COUNTER AND INPUT A NEW LINE
[610] →INCREM
[611] SETMAR: OBJECT←OBJECT,','
[612] SETPAR: PARSE[LOP;]←50↑(CD1,'⊃';OBJECT;(50ρ' '))
[613] ⍝ IF FIX SWITCH IS ON RESET IT TO 0 AND
[614] ⍝ RESTORE THE POINTER INTO THE PARSED
[615] ⍝ STATEMENT TABLE.
[616] INCREM: →(FSWIC≠1)/ATTAY
[617] LOP←HELD ∘ FSWIC←0
[618] ⍝ IF    IS ON RESET IT TO ZERO
[619] ⍝ AND RESTORE THE LOCATION COUNTER.  THEN
[620] ⍝ INCREMENT BOTH THE LOCATION COUNTER AND
[621] ⍝ THE POINTER INTO THE PARSED STATEMENT TABLE
[622] ⍝ BY 1 AND RETURN TO INPUT.
[623] ATTAY: →(SWCH≠1)/OKALL
[624] LOC←(HOLD-1)
[625] SWCH←0
[626] OKALL:LOC←LOC+1
[627] LOP←LOP+1
[628] →INPUT
[629] ⍝ THE INTERPRETIVE PHASE FOLLOWS.  IT IS ACCESSED WHEN
[630] ⍝ THE ACCEPT ROUTINE RETURNS A VALUE OF 0 AS THE INPUT
      LINE.
[631] ⍝ THIS ACTION IS CAUSED WHEN THE USER TYPES A REVERSE
[632] ⍝ SLASH (\) AS INPUT TEXT.  THIS ROUTINE IS
[633] ⍝ TERMINATED WHEN EITHER A \ IS ENCOUNTERED OR IF AN
      END
[634] ⍝ STATEMENT IS FOUND WHEN THE PROGRAM IS AT LEVEL 0.
[635] ⍝ INITIALIZATION FOR INTERPRETIVE PHASE
[636] INTERP: LOC←1
[637] TEXT
[638] ⍝ ADJUST FOR OVERLAY
[639] ⍎')ERASE ACCEPT'
[640] ⍎')ERASE BLANK'
[641] ⍎ ')LOAD USEIT'
[642] ⍎ ')LOAD UNQUOTE'
[643] ⍎ ')ERASE LUKUP'
[644] ⍎ ')ERASE CHEK'
[645] PARSE⌊LOP;1⌋←'\'
[646] MKEY←0
[647] CONDT←''
[648] ⍝ PICK OFF OPCODE
[649] OPCDE: OPCODE←PARSE[LOC;1]
[650] →(OPCODE='\ ')/ENDINT,NEXTLINE
[651] K←2
[652] ⍝ DETERMINE TYPE OF CONDITIONAL, IF ANY
[653] C←PARSE⌊LOC;2⌋
```

```
[654] ⍝ PICK OFF CONDITIONAL
[655] BR1: K←K+1
[656] →(K>50)/ERW
[657] →(PARSE[LOC;K]≠'⊃')/BR1
[658] COND←2↓((K-1)↑PARSE[LOC;])
[659] ⍝ BRANCH TO APPROPRIATE CONDITIONAL-HANDLING ROUTINE
[660] →(C='∩∪⊂')/OBJPICK,CONDI,LOGI
[661] CONDI: →(((CONDT='Y')∧(MKEY=1))∨((CONDT='N')∧(MKEY=0)))/
      OBJPICK
[662] →NEXTLINE
[663] ERW: 'K EXCEEDS LENGTH OF PARSED STMT TABLE'
[664] →NEXTLINE
[665] ⍝ LOGICAL CONDITIONAL ROUTINE
[666] LOGI: →((UNQUOTE CONDT)>0)/OBJPICK
[667] →NEXTLINE
[668] ⍝ PICK OFF OBJECT
[669] OBJPICK: OBJECT←K↓PARSE[LOC;]
[670] →(OPCODE='TAMJCUEYN')/TYPE,ANSW,MATCHIT,JUMP,COMPUTE,
      USE,END,YS,NO
[671] ⍝ MATCH FUNCTION ROUTINE
[672] MATCHIT: ⍞ ')LOAD IN'
[673] ⍞')LOAD TIMEON'
[674] ⍞')LOAD MATCH'
[675] MKEY←LASTANS MATCH OBJECT
[676] ⍞')ERASE TIMEON'
[677] ⍞')ERASE IN'
[678] ⍞ ')ERASE MATCH'
[679] →NEXTLINE
[680] TYPE: UNQUOTE OBJECT
[681] →NEXTLINE
[682] ⍝ ANSWER STATEMENT ROUTINE
[683] ANSW:'?'
[684] →(OBJECT[1]='⍝')/TRY2
[685] LASTANS←UNQUOTE OBJECT
[686] →NEXTLINE
[687] TRY2: LASTANS←⍬
[688] →NEXTLINE
[689] ⍝ JUMP STATEMENT ROUTINE
[690] JUMP:LOC←(UNQUOTE OBJECT)
[691] →OPCDE
[692] ⍝ COMPUTE STATEMENT ROUTINE
[693] COMPUTE:UNQUOTE OBJECT
[694] →NEXTLINE
[695] ⍝ USE STATEMENT ROUTINE
[696] USE: →(OBJECT[1]='*')/USELAB
[697] ⍝ USING AN APL SUBROUTINE
[698] USEIT OBJECT
[699] →NEXTLINE
[700] ⍝ USING A PILOT SUBROUTINE
```

```
[701] USELAB: HOLDER←LOC
[702] LEVEL←LEVEL+1
[703] OBJECT←1↓OBJECT
[704] →(LEVEL≤2)/JUMP
[705] ⍝ ATTEMPTED TO NEST SUBROUTINE CALLS.
[706] 'ERROR -- A SUBROUTINE MAY NOT CALL'
[707] 'ANOTHER SUBROUTINE'
[708] →ENDINT
[709] ⍝ END STATEMENT ROUTINE
[710] END: LEVEL←LEVEL-1
[711] →(LEVEL=0)/ENDINT
[712] LOC←HOLDER+1
[713] →OPCDE
[714] →NEXTLINE
[715] ⍝ YES STATEMENT ROUTINE
[716] YS: →(MKEY=1)/TYPE
[717] →NEXTLINE
[718] ⍝ NO STATEMENT ROUTINE
[719] NO: →(MKEY=0)/TYPE
[720] NEXTLINE: LOC←LOC+1
[721] →OPCDE
[722] ⍝ OUTPUT TERMINATION MESSAGE
[723] ENDINT: 'INTERPRETATION COMPLETE'
```

```
∇ R←ACCEPT;J;TEXT1
 [1]   ⍝ THIS SUBROUTINE IS CALLED BY LEX TO ACCEPT
 [2]   ⍝ THE NEXT LINE OF INPUT TO PILOT.  LEADING
 [3]   ⍝ BLANKS ARE ELIMINATED AND IF A BLANK
 [4]   ⍝ LINE IS INPUT, A WARNING MESSAGE IS
 [5]   ⍝ PRINTED AND THE SYSTEM WAITS FOR USER INPUT.
 [6]   ⍝ INITIALIZE POINTER INTO INPUT LINE
 [7]   J←1
 [8]   ⍝ INPUT A LINE OF TEXT
 [9]   TEXT1←⍞
 [10]  ⍝ CHECK FOR A BLANK LINE
 [11]  ⍝ IF SO, RETURN ERROR WARNING
 [12]  →(∧/(100↑(TEXT1))=' ')/BLKRTN
 [13]  ⍝ ELIMINATE LEADING BLANKS
 [14]  L3: →(TEXT1[J]≠' ')/NONBL
 [15]  J←J+1
 [16]  →L3
 [17]  ⍝ IF A \ IS INPUT, THIS INDICATES THE USER
 [18]  ⍝ HAS FINISHED HIS PROGRAM DEFINITION AND IS READY
 [19]  ⍝ FOR THE INTERPRETIVE PHASE TO BEGIN.
 [20]  ⍝ IN THIS EVENT, THE VALUE 0 IS RETURNED.
 [21]  ⍝ OTHERWISE THE INPUT LINE SANS LEADING
 [22]  ⍝ BLANKS IS RETURNED.
 [23]  NONBL: →(TEXT1[J]='\')/OUT
 [24]  ⍝ RETURN THE LINE OF TEXT
 [25]  R←(J-1)↓TEXT1
 [26]  →0
 [27]  ⍝ ERROR WARNING PRINTED IF BLANK LINE INPUT
 [28]  ⍝ THEN USER IS ALLOWED TO TRY AGAIN.
 [29]  BLKRTN:  'BLANK LINES ARE INVALID -- TRY AGAIN'
 [30]  →1
 [31]  ⍝ RETURN 0 IF PROGRAM INPUT IS COMPLETE
 [32]  OUT: R←0
∇
```

```
∇ R←PATTERN IN STRING;M;N;J;FLINK;K;A
 [1]   ⍝ THIS SUBROUTINE IS CALLED BY THE SUBROUTINE MATCH.
 [2]   ⍝ IT PERFORMS A MOVING WINDOW STRING MATCH TO
 [3]   ⍝ DETERMINE IF THE STRING PATTERN IS CONTAINED IN
 [4]   ⍝ STRING.
 [5]   ⍝ INITIALIZE FAILURE LINK ARRAY
 [6]   FLINK←50⍴0
 [7]   ⍝ CALCULATE LENGTHS OF STRING AND PATTERN
 [8]   M←⍴STRING
 [9]   N←⍴PATTERN
 [10]  ⍝ INITIALIZE FAILURE LINK ARRAY TO ZERO
 [11]  FLINK[1] ← 0
 [12]  ⍝ SET UP FAILURE LINK ARRAY
 [13]  ⍝ THIS ARRAY PROVIDES INFORMATION CONCERNING
 [14]  ⍝ WHERE TO LOOK NEXT IN THE STRING, ESP.
 [15]  ⍝ AFTER A PARTIAL MATCH.
 [16]  ⍝ BY DEFINITION, FLINK[1]=0.  SUPPOSE WE HAVE
 [17]  ⍝ COMPUTED FLINK[1],FLINK[2],...,FLINK[J].
 [18]  ⍝ LET FLINK[J] = I
 [19]  ⍝ EXAMINE PATTERN[J+1] AND PATTERN[I+1]
 [20]  ⍝ TO COMPUTE FLINK[J+1].  IF PATTERN[J+1]
 [21]  ⍝ EQUALS PATTERN[I+1], THEN FLINK[J+1] EQUALS
 [22]  ⍝ FLINK[J]+1 SINCE PATTERN[1]...PATTERN[I+1]
 [23]  ⍝ EQUALS PATTERN[J-I+1]...PATTERN[J+1]
 [24]  ⍝ BUT IF PATTERN[J+1]≠PATTERN[I+1] WE FIND THE
 [25]  ⍝ SMALLEST VALUE OF M WHICH SATISFIES EITHER
 [26]  ⍝ FLINKM[J]=U AND PATTERN[J+1]=PATTERN[U+1] OR
 [27]  ⍝ FLINKM[J]=0 AND PATTERN[J+1]≠PATTERN[1]
 [28]  ⍝ FOR THE FIRST CASE, FLINK[J+1]=U+1
 [29]  ⍝ FOR THE SECOND CASE, SET FLINK[J+1]=0.
 [30]  J←2
 [31]  ⍝ WHEN J EXCEEDS THE LENGTH OF PATTERN, THE
 [32]  ⍝ ARRAY HAS BEEN SET UP.
 [33]  LO1: →(J>N)/OUT
 [34]  I←FLINK[J-1]
 [35]  LO2: →((PATTERN[J]= PATTERN[I+1])∨(I≤0))/LO3
 [36]  I←FLINK[I]
 [37]  →LO2
 [38]  LO3: →((PATTERN [J] ≠ PATTERN [I+1] ∧(I=0))/ZERO
 [39]  FLINK[J]←I+1
 [40]  J←J+1
 [41]  →LO1
 [42]  ZERO: FLINK[J]←0
 [43]  J←J+1
 [44]  →LO1
 [45]  ⍝ CATENATE SPECIAL CHARACTERS TO PATTERN AND STRING
 [46]  OUT: K←1 ∘ A←PATTERN[1]
 [47]  PATTERN←PATTERN,'α'
```

```
[48] ⍝ CATENATE -1 TO FAILURE LINK ARRAY TO FLAG END.
[49] FLINK[N+1]←¯1
[50] ⍝ CATENATE ⊥ AND PATTERN[1] TO STRING
[51] STRING←STRING,'⊥'
[52] STRING←STRING,A
[53] ⍝ CHECK LOCATION OF A IN STRING
[54] ADVANCE: →(STRING[K]=A)/TST1
[55] K←K+1
[56] →ADVANCE
[57] ⍝ IF THE LOCATION POINTER TO A EXCEEDS THE
[58] ⍝ LENGTH OF STRING, PATTERN WAS NOT IN STRING.
[59] ⍝ SO RETURN 0.
[60] ⍝ OTHERWISE ADVANCE AND COMPARE.
[61] TST1: →(K>M)/NO
[62] J←1
[63] ⍝ INCREMENT POINTERS J AND K AND COMPARE
[64] ⍝ THE STRINGS.
[65] MATCHED: J←J+1
[66] K←K+1
[67] LOOPE: →(STRING[K]=PATTERN[J])/MATCHED
[68] J←FLINK[J]
[69] ⍝ IF J=0 CHECK TO SEE IF A=STRING[K]
[70] →(J=0)/ADVANCE
[71] ⍝ SEE IF END OF FAILURE LINK ARRAY.
[72] ⍝ IF YES, RETURN 1
[73] ⍝ OTHERWISE CONTINUE LOOP
[74] →(J≠1)/TST2
[75] →(STRING[K]≠A)/ADVANCE
[76] →MATCHED
[77] TST2: →(J>0)/LOOPE
[78] ⍝ PATTERN WAS CONTAINED
[79] R←1
[80] →0
[81] NO: R←0
∇
```

```
∇ R←UNQUOTE X
[1]  ⍝ THIS SUBROUTINE IS CALLED BY LEX.
[2]  ⍝ X MUST BE A LITERAL STRING.
[3]  ⍝ UNQUOTE FIRST REMOVES THE QUOTES
[4]  ⍝ FROM X AND THEN IT EXECUTES THAT
[5]  ⍝ ARGUMENT.  THE RESULT IS RETURNED VIA R.
[6]  ⍎'R←',X
∇
```

```
∇ R←ANS MATCH MSTRING;J;K;PATTERN
[1]   ⍝ THIS SUBROUTINE IS CALLED BY THE PROGRAM
[2]   ⍝ LEX TO PICK OFF A CHARACTER STRING FROM
[3]   ⍝ THE OBJECT OF THE MATCH STATEMENT.
[4]   ⍝ PLACE IT IN THE VARIABLE PATTERN, AND
[5]   ⍝ CALL THE SUBROUTINE IN TO DETERMINE IF
[6]   ⍝ THAT STRING WAS CONTAINED IN THE MOST
[7]   ⍝ RECENT USER RESPONSE.  THIS SUBROUTINE
[8]   ⍝ RETURNS 1 IF ANY ELEMENT OF THE OBJECT
[9]   ⍝ WAS CONTAINED IN THE MOST RECENT USER
[10]  ⍝ RESPONSE, 0 OTHERWISE.
[11]  ⍝ CATENATE EXTRA COMMA TO END OF MATCH STRING
[12]  ⍝ TO INDICATE WHEN TO TERMINATE LOOP
[13]  MSTRING←MSTRING,','
[14]  ⍝ CONVERT ANS AND MSTRING TO
[15]  ⍝ LITERALS IF THEY ARE NOT ALREADY
[16]  ⍝ ANS IS THE LAST USER ANSWER
[17]  ⍝ MSTRING IS THE OBJECT OF THE MATCH STATEMENT.
[18]  ANS←;ANS
[19]  MSTRING←;MSTRING
[20]  ⍝ REMOVE MULTIPLE BLANKS FROM ANSWER
[21]  J←1
[22]  REPEAT: J←J+1
[23]  →(J>⍴ANS)/TSTT
[24]  →(ANS[J]≠ANS[J-1])/REPEAT
[25]  →(ANS[J]≠' ')/REPEAT
[26]  ANS←(J-1)↑ANS),(J↓ANS)
[27]  J←J-1
[28]  →REPEAT
[29]  ⍝ INITIALIZE MATCH SUCCESS INDICATOR J
[30]  TSTT: J←0
[31]  ⍝ THE FOLLOWING LOOP PICKS OFF THE NEXT
[32]  ⍝ CUE WORD TO BE MATCHED
[33]  ⍝ AGAINST THE INPUT STRING ANS
[34]  ⍝ BY SCANNING TO THE NEXT COMMA.
[35]  ⍝ EXIT IF CUE WORDS EXHAUSTED
[36]  ⍝ I.E., IF THE LENGTH OF MSTRING IS LESS
[37]  ⍝ THAN 2, ONLY A COMMA CAN REMAIN.
[38]  TST: →(2>⍴MSTRING)/AUS
[39]  K←1
[40]  AG:→(MSTRING[K]=',')/CFOUND
[41]  K←K+1
[42]  →AG
[43]  ⍝ WHEN A COMMA IS SPOTTED,
[44]  ⍝ PLACE THE STRING IN PATTERN.
[45]  CFOUND: PATTERN←(K-1)↑MSTRING
[46]  ⍝ TRUNCATE MSTRING FOR NEXT MATCH
[47]  MSTRING←K↓MSTRING
```

```
 [48] ⍝ NOTE: LEADING/TRAILING BLANKS NOT REMOVED
 [49] ⍝ IF THE LENGTH OF MSTRING EXCEEDS THAT OF PATTERN
 [50] ⍝ CHECK TO SEE IF PATTERN IS CONTAINED IN MSTRING.
 [51] ⍝ IF THE LENGTH OF MSTRING IS LESS THAN
 [52] ⍝ THAT OF PATTERN, CHECK TO SEE IF
 [53] ⍝ THE CUE WORDS ARE EXHAUSTED.
 [54] ((⍴MSTRING)<(⍴PATTERN))/TST
 [55] ⍝ J IS 1 IF ANY OF THE CUE WORDS ARE
 [56] ⍝ CONTAINED IN MSTRING.
 [57] J←((PATTERN IN ANS)+J)>0
 [58] →TST
 [59] ⍝ WHEN ALL CUE WORDS HAVE BEEN
 [60] ⍝ EXHAUSTED, RETURN THE VALUE OF J.
 [61] AUS: R←J
∇
```

```
∇ USEIT PGMNAME
 [1]  ⍝ THIS SUBROUTINE IS CALLED BY LEX.
 [2]  ⍝ IT USES THE UNQUOTE FEATURE TO
 [3]  ⍝ CAUSE THE PROGRAM CALLED 'PGMNAME'
 [4]  ⍝ TO BE LOADED INTO WORKING STORAGE,
 [5]  ⍝ THEN TO BE EXECUTED.
 [6]  ⍝ THIS ACTION OCCURS WHEN THE MAIN
 [7]  ⍝ PROGRAM, LEX, ENCOUNTERS A USE STATEMENT.
 [8]  ⍝ LOAD THE PROGRAM
 [9]  ⍎ ')LOAD ';PGMNAME
 [10] ⍝ EXECUTE THE PROGRAM
 [11] ⍝ AFTER THE PROGRAM HAS BEEN EXECUTED
 [12] ⍝ CONTROL RETURNS TO THE NEXT SEQUENTIAL
 [13] ⍝ STATEMENT IN LEX.
 [14] ⍎ PGMNAME
∇
```

```
∇ R←TIMEON
 [1]  ⍝ THIS SUBROUTINE WAS USED IN TIMING THE
 [2]  ⍝ VARIOUS SEARCH SUBROUTINES DESCRIBED IN
 [3]  ⍝ CHAPTERS III AND IV.
 [4]  ⍝ THE ⌶21 (I-BEAM) FUNCTION RETURNS THE TOTAL
 [5]  ⍝ AMOUNT OF TIME, IN SIXTIETHS OF A SECOND,
 [6]  ⍝ THAT THE COMPUTER HAS BEEN UTILIZED SINCE
 [7]  ⍝ SIGN-ON.  IT CAN BE USED TO TIME A FUNCTION'S
 [8]  ⍝ PERFORMANCE BY STORING THE CURRENT ⌶21
 [9]  ⍝ EXECUTING THE FUNCTION, THEN
 [10] ⍝ SUBTRACTING THE OLD VALUE FROM THE NEW ONE.
 [11] R←⌶21
∇
```

```
∇ D←CHEK OBCOPY
[1]  →((⍳8)∈1+⌶28)/GO,DOM,SYN,VAL,LEN,RAN,BIG,ZER
[2]  GO:  D←0
[3]  ⍎'R←';OBCOPY
[4]  →0
[5]  ⍝ THIS SUBROUTINE IS CALLED BY LEX TO PERFORM ERROR
[6]  ⍝ ANALYSIS ON THE OBJECT OF A COMPUTE STATEMENT.
[7]  ⍝ IT UTILIZES THE ERROR TRAP FEATURE OF YORK APL.
[8]  ⍝ THE ERROR TRAP FOR THIS SUBROUTINE IS EVOKED
[9]  ⍝ USING
[10] ⍝           EΔCHEK←3
[11] ⍝ IN LINE 3 OF CHEK WILL BE HANDLED BY THE
[12] ⍝ ERROR TRAP.  IN THIS MANNER THE FUNCTION DOES
[13] ⍝ NOT BECOME SUSPENDED.
[14] ⍝ THE FIRST LINE OF THE FUNCTION IS RESERVED
[15] ⍝ FOR THE ⌶28 (I-BEAM) FUNCTION.  IF AN ERROR
[16] ⍝ IS ENCOUNTERED WHILE THE FUNCTION IS EXECUTING,
[17] ⍝ THIS I-BEAM FUNCTION IS SET TO THE VALUE
[18] ⍝ CORRESPONDING TO THAT ERROR.
[19] ⍝ THE TABLE OF VALUES FOR THE ERRORS IS:
[20] ⍝      1      DOMAIN ERROR
[21] ⍝      2      SYNTAX ERROR
[22] ⍝      3      VALUE ERROR
[23] ⍝      4      LENGTH ERROR
[24] ⍝      5      RANK ERROR
[25] ⍝      6      NUMBER TOO BIG
[26] ⍝      7      DIVISION BY ZERO
[27] ⍝ FOR INSTANCE, IF A VALUE ERROR SHOULD OCCUR,
[28] ⍝ ⌶28 WILL CONTAIN THE VALUE 3.  EACH VALUE THAT
[29] ⍝ ⌶28 MAY BE SET TO CORRESPONDS TO THE NUMBERING
[30] ⍝ SEQUENCE ABOVE.  IF NO ERROR OCCURS WHEN
[31] ⍝ THE UNQUOTE IS APPLIED TO THE LITERAL STRING CALLED
[32] ⍝ OBJECT, ⌶28 REMAINS THE EMPTY VECTOR (⍳0)
[33] ⍝ AND THE SUBROUTINE RETURNS 0.
[34] ⍝ OTHERWISE AN APPROPRIATE ERROR MESSAGE
[35] ⍝ IS PRINTED, AND THE SUBROUTINE RETURNS A
[36] ⍝ VALUE OF 1.
[37] ⍝ DOMAIN ERROR ENCOUNTERED
[38] DOM: 'DOMAIN ERROR IN COMPUTE STATEMENT'
[39] D←1
[40] →0
[41] ⍝ SYNTAX ERROR ENCOUNTERED
[42] SYN: 'SYNTAX ERROR IN COMPUTE STATEMENT'
[43] D←1
[44] →0
[45] ⍝ VALUE ERROR ENCOUNTERED
[46] VAL: 'VALUE ERROR IN COMPUTER STATEMENT.  POSSIBLY'
[47] 'AN UNDEFINED VARIABLE OR ATTEMPT MADE TO '
[48] 'USE PROGRAM NOT IN WORKING STORAGE'
```

```
[49] D←1
[50] →0
[51] ⍝ LENGTH ERROR ENCOUNTERED
[52] LEN: 'LENGTH ERROR IN COMPUTE STATEMENT'
[53] D←1
[54] →0
[55] ⍝ RANK ERROR ENCOUNTERED
[56] RAN: 'RANK ERROR IN COMPUTE STATEMENT'
[57] D←1
[58] →0
[59] ⍝ NUMBER TOO LARGE
[60] BIG: 'NUMBER IN COMPUTE STATEMENT EXCEEDS'
[61] 'RANGE OF PILOT SYSTEM'
[62] D←1
[63] →0
[64] ⍝ DIVISION BY ZERO ENCOUNTERED
[65] ZER: 'ATTEMPTED DIVISION BY ZERO'
[66] 'IF AN UNDEFINED VARIABLE MESSAGE WAS OUTPUT'
[67] 'THAT COULD BE THE PROBLEM, SINCE PILOT'
[68] 'USES THE INITIALIZATION VALUES FOR ANY'
[69] 'UNDEFINED VARIABLES'
[70] D←1
[71] →0
∇
```

```
∇ R←BLANK KSPOT
 [1]   ⍝ THIS SUBROUTINE IS CALLED BY LEX.
 [2]   ⍝ IT IS USED IN CONJUNCTION WITH THE PRINTING
 [3]   ⍝ OF A LINE OF INPUT TEXT TO INDICATE WHERE
 [4]   ⍝ IN THE LINE AN ERROR OCCURRED.  THE ARRAY
 [5]   ⍝ BLINE IS INITIALIZED TO ALL BLANKS, THEN
 [6]   ⍝ A '∆' IS POSITIONED AT THE LOCATION IN
 [7]   ⍝ BLINE CORRESPONDING TO THE LOCATION
 [8]   ⍝ OF THE ERROR IN THE PILOT STATEMENT.
 [9]   ⍝ DURING A PROGRAMMING SESSION,
 [10]  ⍝ WHEN AN ERROR IS DETECTED IN A LINE
 [11]  ⍝ OF INPUT TEXT, LEX CAUSES THE LINE TO BE
 [12]  ⍝ PRINTED, THEN IT CALLS THIS SUBROUTINE
 [13]  ⍝ TO PRINT THE NEXT LINE, WHICH IS IN EFFECT
 [14]  ⍝ A POINTER TO THE ERROR.
 [15] BLINE←100⍴' '
 [16] BLINE[KSPOT]←'∆'
 [17] R←BLINE
∇
```

```
∇ R←A LUKUP B;XXX;I;ROWS;QQ
 [1]  ⍝ THIS SUBROUTINE IS CALLED BY LEX.  IT PERFORMS A
 [2]  ⍝ TABLE SEARCH FOR LABEL OR VARIABLE RESOLUTION.
 [3]  ⍝ THE VARIABLE A IS A LABEL OR VARIABLE NAME AND
 [4]  ⍝ THE VARIABLE B IS A TABLE, EITHER THE
 [5]  ⍝ LABEL NAME TABLE (LABTAB) OR
 [6]  ⍝ THE VARIABLE NAME TABLE (VARTAB).
 [7]  ⍝ IF A IS CONTAINED IN B,
 [8]  ⍝ R RETURNS A VALUE OF I, THE ROW IN TABLE B
 [9]  ⍝ WHICH CONTAINS A.
 [10] ⍝ IF HOWEVER, A IS NOT CONTAINED IN B,
 [11] ⍝ R RETURNS A VALUE OF 0 TO LEX.
 [12] ⍝ INITIALIZE ROW POINTER
 [13] I←¯1
 [14] ⍝ SET THE VARIABLE ROWS EQUAL TO
 [15] ⍝ THE NUMBER OF ROWS IN B.
 [16] ROWS←(⍴B)[1]
 [17] QQ←⍴B
 [18] ⍝ ADJUST THE LENGTH OF A AND
 [19] ⍝ CATENATE A TO THE END OF B.
 [20] A←A,(QQ[2]-⍴A)⍴' '
 [21] B←B,[1] A
 [22] ⍝ INCREMENT POINTER BY 3.
 [23] Q3: I←I+3
 [24] ⍝ IS ROW I OF B EQUAL TO A?
 [25] ⍝ IF SO, CHECK TO SEE IF THE POINTER I
 [26] ⍝ EXCEEDS THE NUMBER OF ROWS IN B.
 [27] ⍝ IF NOT, ROW I OF B CONTAINS A.
 [28] ⍝ SO RETURN THE POINTER.
 [29] ⍝ OTHERWISE, RETURN ZERO.
 [30] →Q4×⍳∧/A=B[I;]
 [31] ⍝ IF A EQUALS ROW I+1 OF B, INCREMENT I BY 2
 [32] ⍝ AND CHECK TO SEE IF ROWS EXCEEDS I.
 [33] ⍝ IF SO, RETURN THE POINTER I.
 [34] ⍝ OTHERWISE, A WAS NOT CONTAINED IN B,
 [35] ⍝ SO RETURN 0.
 [36] →Q4×⍳∧/A=B[I+1;]
 [37] →Q5×⍳∧/A=B[I+2;]
 [38] ⍝ CONTINUE TO REPEAT THE LOOP UNTIL
 [39] ⍝ I POINTS TO THE ROW WHICH CONTAINS
 [40] ⍝ A IN TABLE B.  BY THE TESTS DESCRIBED
 [41] ⍝ ABOVE, IT CAN BE DETERMINED WHETHER
 [42] ⍝ I POINTS TO THE VALUE OF A THAT WAS
 [43] ⍝ CATENATED ONTO THE END OF B, OR IF
 [44] ⍝ A IS CONTAINED WITHIN THE ORIGINAL
 [45] ⍝ TABLE CALLED B.
 [46] →Q3
 [47] ⍝ INCREMENT POINTER BY 2
 [48] Q5: I←I+2
```

```
[49] ⍝ SEE IF I POINTS TO THE CATENATED VALUE OF A
[50] ⍝ OR IF A IS ACTUALLY CONTAINED IN TABLE B.
[51] Q4: →(ROWS≥I)/OUT
[52] ⍝ RETURN VALUE OF ZERO
[53] R←0
[54] →0
[55] ⍝ RETURN VALUE OF POINTER.
[56] OUT: R←I
∇
```

APPENDIX C

SAMPLE PROGRAMS

This appendix contains two sample PILOT programs. The first is set up to calculate Zeller's Congruence, i.e., to compute the day of the week given the month, day, year, and century. The APL subroutine ZELLER is used in the PILOT program by utilization of the USE statement. The second example is a conversational routine designed to quiz the user on the parts of a slide rule. Although both are trivial programs, their main purpose is to demonstrate the use of the PILOT language.

Each time the M-statement is executed, the number 14 is output. Tracing the function as well as making a number of modifications failed to solve this problem. The function itself, however, performs properly.

EXAMPLE 1

```
      T: WOULD YOU LIKE TO KNOW WHAT DAY
      T: OF THE WEEK YOU WERE BORN ON?
      A:
      M:YES,OK,SURE
      UY:ZELLER
      Y:SO NOW YOU KNOW
      JY:*LAST
      N: COME ON.  TRY IT.
      U:ZELLER
      T: WAS THAT SO BAD?
      *LAST E:
```

```
∇ ZELLER
 [1] ⍝ SUBROUTINE TO COMPUTE ZELLER'S CONGRUENCE
 [2] T←12 4⍴'JAN FEB MAR APR MAY JUNEJULYAUG SEPTOCT NOV DEC '
 [3] WK←7 4⍴'SUN MON TUE WED THURFRI SAT '
 [4] 'INPUT MONTH, DAY, CENTURY AND YEAR, SEPARATED BY BLANKS'
 [5] START: INFO←⍞
 [6] IN←INFO[1]
 [7] T[IN;]; ' ';INFO[2];' ';INFO[3];INFO[4]
 [8] →(INFO[1]>2)/TWO
 [9] INFO[1]←INFO[1]+10
 [10] INFO[4]←INFO[4]-1
 [11] →THRE
 [12] TWO: INFO[1]←INFO[1]-2
 [13] THRE: F←7|(INFO[2]+INFO[4]+⌊((2.6×INFO[1])-.2)+⌊
      (INFO[3]÷4)+⌊(INFO[4]÷4))-2×INFO[3]
 [14] 'OCCURRED ON A ';WK[F+1;]
∇
```

```
      WOULD YOU LIKE TO KNOW WHAT DAY
      OF THE WEEK YOU WERE BORN ON?
      ?
      SURE, WHY NOT?
```

```
14
INPUT MONTH, DAY, CENTURY AND YEAR, SEPARATED BY BLANKS
 ⍞      8      23      19          53
AUG  23 1953
OCCURRED ON A SUN
SO NOW YOU KNOW
```

```
INTERPRETATION COMPLETE
```

EXAMPLE 2

```
T: NAME ONE OF THE SCALES ON THE
T: SLIDE RULE USED FOR MULTIPLICATION
T: AND DIVISION?
*INP A:
M:D,C
Y: CORRECT
N: TRY AGAIN.
JN: *INP
T: HOW MANY MAJOR DIVISIONS ARE
T: THE C AND D SCALES DIVIDED INTO?
A:
M:9,NINE
TY: VERY GOOD.
N: NO, THE ANSWER IS 9.
T: WHAT IS THE ONLY LINEAR
T: SCALE ON THE SLIDE RULE?
A:
M:L
TY: CORRECT.
N: NO, THE ANSWER IS L.
T: THIS COMPLETES THE SAMPLE.
E:
\
```

NAME ONE OF THE SCALES ON THE
SLIDE RULE USED FOR MULTIPLICATION
AND DIVISION?
?
L

14
TRY AGAIN.
?
D

14
CORRECT
HOW MANY MAJOR DIVISIONS ARE
THE C AND D SCALES DIVIDED INTO?
?
EIGHT

14
NO, THE ANSWER IS 9.
WHAT IS THE ONLY LINEAR
SCALE ON THE SLIDE RULE?
?
THE ANSWER IS L, OF COURSE.

14
CORRECT.
THIS COMPLETES THE SAMPLE.
INTERPRETATION COMPLETE

VITA

Karen Sue Gregg was born in Newport, Tennessee, on August 23, 1953. She attended elementary school in Parrottsville, Tennessee and graduated from Parrottsville High School in 1971. In August 1973, she received a Bachelor of Arts degree in Mathematics from East Tennessee State University. She did graduate work in Mathematics prior to her transfer to the University of Tennessee in January 1974. She received the Master of Science degree in Computer Science from the University of Tennessee in June 1975.

As an undergraduate she was active in a number of campus organizations, including Gamma Sigma Sigma, Alpha Lambda Delta, and the University Center Program Committee Program Board. She was also President of the Math Club and Senior Manager of the campus theater. In 1973, she received the German Consulate Award and was elected to "Who's Who in American Colleges and Universities."

She is a member of the Association of Computing Machinery.