University of Pennsylvania

A PROPOSAL FOR EXAMPLE INDUCED PROGRAMMING

A thesis submitted to the Faculty of the Moore School of Electrical Engineering in partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer and Information Science)

Rochelle Fleischmann

1975

MS-CIS-79-7

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

2

A PROPOSAL FOR EXAMPLE INDUCED PROGRAMMING

Rochelle Fleischmann

A thesis submitted to the Faculty of the MOORE SCHOOL OF ELECTRICAL ENGINEERING in partial fulfillment of the requirements of the degree of Master of Science in Engineering for graduate work in Computer and Information Sciences

Philadelphia, Pa.

April, 1975

University of Pennsylvania

THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

Title of thesis: A PROPOSAL FOR EXAMPLE INDUCED PROGRAMMING

Abstract

The user, programmer-analyst, programmable device chain of communication is analyzed as a type of authoritarian hierarchy. The elimination of the function of the programmer-analyst is presented as a desirable goal. The modus operandi of the programmer-analyst is discussed and analogs suitable for implementation on a programmable device are presented. These analogs are embedded in a computer simulation and tested. Problem areas are discussed and further areas of investigation are suggested.

A thesis submitted to the Faculty of the MOORE SCHOOL OF ELECTRICAL ENGINEERING in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Sciences.

April, 1975

R Fleischmann

AUTHOR
ROCHELLE FLEISCHMANN

John W. Carr III

FACULTY SUPERVISOR
PROFESSOR JOHN W. CARR III

Acknowledgements

My sincere thanks to Dr. John W. Carr III for initially suggesting the problem investigated in this thesis and for his constant support and encouragement.

Special appreciation is extended to Phyllis Wharton and to Edward Treister for their patience and helpful suggestions.

My thanks go also to I. P. Sharp Associates Limited for providing me most generously with the use of their facilities.

TABLE OF CONTENTS

INTRODUCTION

Since the introduction of programmable devices, their availability has been severely limited to:

those who can program or are willing
and able to learn how;
those who can pay someone to do
the programming for them;
those who can take advantage of
preprogrammed packages.

Although historically access to programmable devices has been simplified considerably by the introduction of higher level languages, operating systems, etc., there still exists an elite of users determined by aptitude, extensive training and money. Declining costs of the devices themselves have made them available to an ever widening segment of society; however, true democratization rests on the elimination of the programmer.

The programmer-analyst acts as an intermediary between user and machine. In many respects his function resembles that of the high priest in a cultic religion. As long as the uninitiated must turn to him to intercede on their behalf, programmable devices will remain a mysterious and vaguely threatening force in society. This work therefore

has as its ultimate aim the democratization and demystification of the device.

ANALYSIS OF THE PROBLEM

Let us continue the cultic analogy. It is clear that in many such religions, it is not the priest himself who is central to the man god relationship but rather the knowledge that the priest possesses. Assumedly, man does not communicate directly with his god because he doesn't know how. His god demands a specific form of address which only the priest has mastered. The priest provides him with the formalized mode of behaviour appropriate to the message which he wishes to convey. This behaviour, of whatever form, reflects his needs, be they penitential, supplicational, or of some other nature. In a similar vein, the user approaches the programmer-analyst with a "message" which he wishes to have conveyed. The user would communicate directly with the computer if he could, but he doesn't know how. He therefore asks the programmer-analyst to provide him with a formalization of his problem--one which is acceptable to the computer. One must note, however, that the programmer-analyst's knowledge is not simply a catalogue of syntactically correct statements; he also often undertakes the crucial task of problem analysis. The importance of this point will become clear as this paper progresses.

We can now see that the program is a formal embodiment

of a user's message, or let us say, concept. We must now look more closely at two things:

1) how the concept is communicated from the user to the programmer-analyst.
2) how the programmer-analyst uses this concept to construct a program.

Communication Between the User and the Programmer-Analyst

The process which is to take place is the teaching of a concept to a student by a non-professional teacher. In this study the teacher is the user and the student is the programmer-analyst. By assuming a non-professional teacher (i.e., a non-programmer) we infer that the user knows what he wants done but has never formalized the process in the sense of a systematic analysis of the problem. He is capable of demonstrating what he wants to have done given an appropriate facility but he may not be skilled in analysis or communication. As a result he is likely to change his procedure from example to example without specifically explaining why. The effectiveness of this process depends, therefore, on the ability of the student to take an active role. He must be able to identify the point at which he fails to continue understanding--i.e., the point at which the teacher has made some logical leap from the student's point of view. He must also be able to ask questions which

will extract missing information merely implied in the statement of the problem.

Since it is our intention to eliminate the programmer-analyst as intermediary, his function must be taken over by the programmable device. The device must be able to accept information directly from the user in a form natural to him and must be able to elicit missing information.

We have said that the user is at least capable of demonstrating his concept. The programmer-analyst often takes advantage of this ability by asking the user to give some examples of what he wishes to have accomplished. This procedure obviates the necessity of an analytic approach on the part of the user and generally reduces communication problems. The programmer-analyst takes note of the examples and compares one to another. If he notices at any time that one example deviates from another one, he asks the user why the difference exists. Thus although the user has failed initially to point out the reason for a crucial change in procedure, the programmer-analyst has been able to account for it. Basically, the active participation of the programmer-analyst reduces to making three statements:

1) Show me what you want to do.

2) That's not what you did last time.

3) Why did you change procedure?

The model described herein uses analogs of these three statements. It contains an input section which accepts examples from the user in a form natural to him; it contains a comparison section which searches for changes between examples; and it contains an interrogatory section which actively requests information to account for unexplained inconsistencies. Note that underlying this approach is the assumption that the user will be able to explain any apparent inconsistencies which are discovered and that ultimately no such anomalies will remain.

There seems to exist a similarity between this approach and the TOTE unit (test, operate, test, exit) which Miller, Galanter, and Pribram[1] use as a model of most human problem solving. The first test corresponds to statement two above. The operate phase corresponds to statement three. The comparison of examples is now carried out again to determine whether the contradiction still exists; if it does not, the intake portion of the method is concluded corresponding to the exit element of the TOTE unit.

1 Miller, Galanter, and Pribram, Plans and the Structure of Behaviour, Holt, 1960, New York

At the conclusion of the intake phase the proposed method uses the information it has gleaned to construct a program embodying the user's concept.

Construction of the Program

In analyzing the nature of the given problem, a graphic representation of programs has proven useful. Of the various equivalent forms available, I have found the tree structure to be the most appropriate. In this structure, the nodes of the tree correspond to control statements and the branches correspond to computational statements. When only one branch extends downward from a node, the control statement is unconditional; otherwise, the flow passes through one of the possible branches based on a decision made at the node. We will ignore unconditional nodes in this study and instead focus in on the investigation of conditional ones. We will also limit ourselves to the investigation of binary trees in which exactly two branches extend downward from each node.

Since a program is the embodiment of a concept, a concept may be graphically represented as a tree. Each endpoint of that tree represents a subclass of the concept. The endpoints define a partition of the concept into mutually exclusive subconcepts. Each example presented by

the user falls into one of these subconcepts. Therefore in order to reconstruct the complete concept, the user must provide at least one example for each endpoint of the tree. It is then a matter of taking the union of these branches to arrive at the fully grown tree. A description of the algorithm to achieve this union follows in the next chapter.

DESCRIPTION OF TREE GROWTH ALGORITHM

There are two individuals involved in the communication process, the teacher and the student (the user and the programmable device). The first example given by the user becomes the first approximation of the tree. Its structure is trivial; it has no nodes and only one endpoint. Its only branch represents a list of computational statements which are always to be performed on the same constant data. The student remains passive because there can be no contradictions at this point.

As the second example is entered, the student compares each statement to its counterpart in the first example. Comparison is made on two dimensions, operand and operator. A difference in either dimension is sufficient to initiate the operate phase of the TOTE unit. Let us assume a new operand has been entered in the second example. Since, as we noted above, all operands in the first example are assumed to be constants, the use of a new constant must be accounted for. There are two possibilities which the student will examine:

1) The conflicting operands are not really constants but rather they are variables.

2) The conflicting operands are indeed constants and the user has neglected to explain the basis upon which the two different constants were chosen.

To determine which hypothesis is true, the student need only ask the question, "constant or variable?" The method of posing this question depends on the facilities available. A programmer-analyst will ask the question verbally; the computer simulation used in this thesis asks the question in print; a programmable calculator could ask it by locking its keyboard except for the two keys labelled constant and variable.

If the answer to the question is "variable", the student takes note that operands in this position in this example and all future ones are variables and need no longer agree. The second example can then be continued. If the answer to the question is "constant", the student can assume that a node in the tree has been reached and that he must grow a new branch from this node. The old branch emanating from the node will represent subconcepts of the same type as the first example, and the new branch will represent those similar to the second example. As we noted earlier, nodes correspond to conditional statements in a

program. Thus "growing the tree" means inserting a conditional control statement immediately preceding the noticed change in procedure.

We will assume that the condition which is tested to determine flow of control is of the form

A op B

where A is a significant variable whose value is being tested, op is a relational operator such as =, and B is a pivot value which splits the universe of possible values of A into two mutually exclusive partitions. For example,

SPEED = 30

is a condition in which SPEED is the significant variable, = is the relational operator, and 30 is the pivot value. We will also assume that the significant variable is one which has already made its appearance earlier in the tree.

The student must now identify the three constituents of the conditional part of the control statements. To do this he will begin to question the user about each variable which appeared earlier on the tree, including the intermediate results obtained from the computational statements. Of each one he will ask, "Is this the one that made you change procedure?" The programmer-analyst will pose the question verbally; the present simulation displays a variable in

print with a question mark following it and waits for a yes or no answer before proceeding to the next one; a programmable calculator could again lock its keyboard and display the variables on its screen in the same manner as the simulation.

When the student has identified the significant variable he can proceed to ask for the pivot value. If the pivot value is a constant, it can be entered directly by the user; if, however, the pivot value is another variable, the student can continue presenting variables from the point at which he left off. Again, the programmer-analyst will pose the question directly; the simulation prints the statement "ENTER PIVOT VALUE" and waits for a response; the programmable calculator could lock all operator keys except the pivot value entry key and require the user to enter his pivot value.

The last piece of information required is the relational operator. The programmer-analyst asks for it directly; the simulation prints the statement "ENTER RELATION" and waits for a response; and the programmable calculator could again use the strategy of locking appropriate keys. The student can now compose the condition which should result in having the constant from the first example and the con-

stant from the second example on different branches.

The student would now enter the third phase of the TOTE unit--the retesting phase. Does the condition described by the user actually place his two examples into two different subcategories? If not, the student can only say something equivalent to "Sorry, but you're not doing what you told me you're doing. Try another explanation." If, however, all goes well, the fourth phase of the TOTE unit is entered, the exit phase. The student now realizes he is on a new branch of the tree and has no comparing to do. He accepts whatever is described to him by the user.

We have discussed what takes place if the student notices a difference in operands which are both determined to be constants. If instead the difference is in operators, the procedure is identical to what takes place following the "constant or variable?" question: the student assumes that he has reached a node and asks the necessary questions to produce the conditional control statement at the node.

We now see that although we spoke earlier of constructing the tree by the union of its separate branches, we are doing something slightly different. Instead of merging the identical beginnings of the examples, we are actually

growing the tree as the examples are presented. The tree grows cotemporaneously with example presentation. In other words, the student expands his knowledge of the user's concept as it is presented to him.

All succeeding examples are handled in the same way as the second. When the current example reaches an already established node, the branch to be taken is determined from the appropriate, already entered variables in the current example.. Then, when the correct branch is determined, comparisons are made with the data on it.

The resulting tree can correctly handle all examples entered by the user if the user has entered examples of each subconcept. The tree can be said to be a faithful reproduction of the user's concept.

CONCLUSIONS

As the accompanying examples show, the present model can successfully learn a user's concept on the basis of examples given by him. However, it has at least two distinct problem areas.

A major area for continued investigation must be the problem of learning that a loop is present in the user's concept. The problem has at least two facets to it.

In the first place, the user may not know that his concept contains a loop. In fact, he may not know what a loop is. We might think of this possibility as the student looking for an entity or pattern for which no word exists in the user's language. The student and user might be speaking the equivalent of two different languages called analytic and Gestalt. Or we could account for this missing knowledge in another way. E. B. Hunt[2] talks about Piaget's theories of mental capabilities being a function of maturation. He says that the availability of concepts (e.g., looping) to individuals probably profoundly effects their learning strategies. I would think that it effects their

2 Hunt, E. B., Concept Learning an Information Processing Problem, p. 168, John Wiley & Sons, Inc., 1962, New York

concept teaching strategies as well. A frustrating breakdown in communication is often observed between teacher and student when one of them is explaining or asking about something in terms which the other simply cannot relate to. It is not a matter of simplicity or complexity but rather a missing link, and in this case it is not necessarily a matter of a logical connection which has been skipped. Asking about a loop would work with professionals in the elite. But what about everyone else?

Perhaps a very simple example would make this clearer. Suppose that the user wishes to convey the concept "take an average". He might begin by entering three numbers, totalling them and dividing the sum by three. As a second example he might wish to enter four numbers, total them and divide by four. He will trigger the operate phase of the TOTE unit as he enters his fourth number, and he won't be able to answer the questions posed because they are irrelevant to what he is doing. What is his significant variable? Even if he had entered as his first variable the number of numbers to be averaged, and thus had a significant variable, he would still have no pivot value. In order to have one, he would have to have a counter for the number of numbers he has entered so far, a most unnatural thing to have--unless of course you happen to be a programmer. The user might

recognize that he is looping, but then again, he would be more likely to say that he is entering four numbers. One cannot assume that he will understand what the programmer means by entering a variable number four times. The uninitiated user will see four as an adjective, the professional will see four as an adverb. This syntactic distinction might turn out to be the key to the problem.

The second facet of this problem is in the area of pattern recognition. Given that we have a tree which contains an unwound loop in it, how do we go about recognizing it? Let us take the simplest case, the trivial tree with no nodes in it. The one branch represents a series of computational statements. Let us represent them by the string

abcabcabcabcabcabcab

We now have several options for the pattern. It could be

abc

bca

cab

abcabc

bcabca

cabcab

abcabcabc

bcabcabca

cabcabcab

How do we determine which is the correct one? In addition, how do we determine the basis on which the loop continues to cycle? Given the first facet of the problem, can we ask the user in a way which is intelligible and yet does not presuppose knowledge of a professional nature? At what point would we ask the user, assuming that we know what to ask? Cotemporaneously with the presentation of the examples as we did for the conditional statements? At the end of each example? Or perhaps at the end of all example presentation?

Although I have given this first problem considerable thought, I have not come up with a viable solution. My guess is that the best approach will be from a learning theoretic and human engineering angle. Determine what questions would elicit the information needed without mentioning loops and then find a way of aking those questions.

The second problem area is the simple tedium of entering a large number of examples necessary for complex concepts. While this drawback does no damage to the theoretic basis of the model, it is a severe limitation to practical implementation. The model attempts to alleviate this problem by prompting the user and thus relieving him of some of the tedium; however, improved prompting algorithms should be considered.

The range of applications which the model proposed herein can handle is wide but still severely limited by the dual problems of looping and tedium; however, it can demonstrably produce a valid working program based on examples provided by a user.

APPENDICES

APPENDIX I

PROGRAM LISTING

CR

NUM

1234567890.¯ω

OPRS

+-×÷*<≤=≥>≠↑⍞↓○→

ROPRS

<≤=≥>≠

UNARY

+-×÷*<≤=≥>≠

```
      ∇ABSCREATE[□]∇
    ∇ FNM ABSCREATE N
[1]   ⍝IS TIE NUMBER N IN USE?
[2]    →L1×⍳N∈□NUMS
[3]   ⍝IS FNM IN LIB BUT NOT TIED?
[4]    →L2×⍳0=+/(□LIB⌷29)[;11+⍳10]∧.=10↑FNM
[5]   ⍝TIE FNM
[6]    FNM □TIE N
[7]   ⍝ERASE FNM
[8]   L1:FNM □ERASE N
[9]   ⍝CREATE FNM
[10]  L2:FNM □CREATE N
[11]   100000 □RESIZE N
    ∇

      ∇DELETE[□]∇
    ∇ DELETE I;K;L;S;M
[1]   ⍝DELETES ITH EXAMPLE
[2]    STACK←⍳0
[3]    K←,1
[4]   ⍝LEAF?∇
[5]   L0:→(¯1=1↑K←,□READ 1,1↑K)↑0
[6]   ⍝BRANCH?
[7]    →(2=⍴K)↑L1
[8]   ⍝DETERMINE WHICH BRANCH WAS TAKEN.
[9]   ⍝CANNOT USE POINTER SINCE PIVOT VALUE MAY HAVE BEEN ER
      ⍝ASED.
[10]   K←□READ 1,1+1↑K
[11]   →(K[3]=0)↑L4
[12]   →(I∈□READ 2,K[3])↑L2
[13]   →L1,K←K[2 4]
[14]  L2:K←K[1 3]
[15]  ⍝S IS LIST OF EXAMPLE NUMBERS IN COMPONENT K[2] FILE 2
      .
[16]  ⍝L IS LOGICAL VECTOR OF NON-ITH EXAMPLE.
[17]  L1:→(∧/L←I≠S←□READ 2,K[2])↑0
[18]  ⍝DELETE ITH EXAMPLE FROM INDEX COMPONENT.
[19]   (S←L/S) □REPLACE 2,K[2]
[20]  ⍝DELETE ITH OPERAND FROM PROGRAM COMPONENT.
[21]  ⍝DELETE ITH RESULT FROM PROGRAM COMPONENT.
[22]   →(1=⍴S←□READ 1,K[1])↑L3
[23]   L←(~L)/⍳⍴L←1↓L
[24]   M←(S=',')/⍳⍴S
[25]   M←((⍴M)⍴ 0 1)/M
[26]   (S←(M[L-1]↑S),M[L]↓S) □REPLACE 1,1↑K
[27]  ⍝INDEX THROUGH EXAMPLE.
[28]  L3:→L0,K[1]←K[1]+1
[29]  L4:→(K[4]=0)↑L5
[30]   →(I∈□READ 2,K[4])↑L1,K←K[2,4]
[31]   →L0,K←K[1]+1
[32]   →(I∈□READ 2,K[3])↑L2
[33]   →L0,K←K[2]
    ∇
```

```
      ∇DISPLAY[□]∇
    ∇ R←DISPLAY;V
[1]   →(0=⍴STACK)↑L1,R←0
[2]   →0,□←1↑STACK
[3]  L1:'⍬'
    ∇

      ∇DUMP[□]∇
    ∇ DUMP I;J;K;L;R;S
[1]  ⍝DISPLAYS ITH EXAMPLE.
[2]   →((I<1)∨I>M)↑ERR1
[3]   'EXAMPLE ';,'I2' □FMT I
[4]   K←0,0
[5]  L2:K←POINTER K[1]
[6]  ⍝LEAF REACHED?
[7]   →(K[1]=¯1)↑0
[8]  ⍝THIS EXAMPLE ABORTED HERE?
[9]   →((⍴S)<L←(S←□READ 2,K[2])⍳I)↑0
[10] ⍝CHECK FOR NULLARY OPERATOR.
[11]  →(1=⍴S←□READ 1,1↑K)↑L3
[12]  J←2,(S=',')/⍳⍴S
[13]  L←2×L-1
[14]  '      ';J[L-1]↓¯1↓J[L]↑S;1↑S
[15]  J[L]↓¯1↓J[L+1]↑S
[16] L4:→L2
[17] L3:S
[18]  →(S='○')↑L4
[19]  →L4,□←0
[20] ERR1:'INVALID EXAMPLE NUMBER.'
    ∇
```

```
      ∇ENTER[□]∇
    ∇ R←ENTER EG;I;N
[1]   ⍝RETURNS A ZERO IF INPUT STRING CONTAINS AN ERROR. OTH
       ERWISE IT RETURNS I: THE OPISTION OF THE OPERATOR
[2]   ⍝OPPS IS VECTOR OF RECOGNIZED OPERATORS.
[3]   ⍝I IS INDEX OF OPERATOR IN INPUT STRING.
[4]   ⍝OSW IS SET TO ONE IF OPERAND COMES FROM STACK.
[5]    →(1≠⍴I←(EG∈OPRS)/⍳⍴EG)↑ERR1,P←OSW←0
[6]   ⍝THE OPERATOR SHOULD FOLLOW AN OPERAND.
[7]   ⍝UNARY IS THE VECTOR OF OPERATORS WHICH NEED AN OPERAN
       D.
[8]    →((1=I)∧(EG[I]∈UNARY))↑ERR2
[9]   ⍝NUM IS VECTOR OF VALID NUMERIC CHARACTERS.
[10]  ⍝N IS THE OPERAND IN LITERAL FORM.
[11]   →(∨/~(N←(I-1)↑EG)∈NUM)↑ERR3
[12]   →(I≠⍴EG)↑ERR4
[13]  ⍝RETURNS POSITION OF OPERATOR IN INPUT STRING.
[14]   →(~OSW←'⍵'∈N)↑L1,R←I
[15]   →0
[16]  L1:→(1≥⍴□FI N)↑0
[17]   'TOO MANY OPERANDS. RE-ENTER.'
[18]   →R←0
[19]  ERR1:→0,⍴□←'WRONG NUMBER OF OPERATORS. RE-ENTER.'
[20]  ERR2:→0,⍴□←'NO OPERAND. RE-ENTER.'
[21]  ERR3:→0,⍴□←'INVALID NUMERIC CHARACTER. RE-ENTER.'
[22]  ERR4:→0,⍴□←'OPERATOR NOT LAST SYMBOL. RE-ENTER.'
    ∇

      ∇FIRSTEG[□]∇
    ∇ FIRSTEG;ACC;EG;R
[1]    'BEGIN ENTERING FIPST EXAMPLE.'
[2]   ⍝SET ACCUMULATOR TO ZERO.
[3]   L0:STACK←⍳0
[4]    ACC←'0'
[5]   ⍝CREATE NEW PROGRAM AND INDEX FILES.
[6]    'PROG' ABSCREATE 1
[7]    'INDEX' ABSCREATE 2
[8]   ⍝INITIALIZE FIRST COMPONENT OF EACH.
[9]    (2,1) □APPEND 1
[10]   (0,1) □APPEND 2
[11]  L1:→('END'∧.=3↑EG←,GETL '      ')↑L2
[12]   →('CANCEL'∧.=6↑EG)↑L0
[13]  ⍝RETURNS A ZERO SCALAR WHEN ERROR DETECTED.
[14]   →(0=R←ENTER EG)↑L1
[15]  ⍝R IS THE POSITION OF THE OPERATOR IN THE LITERAL INPU
       T STRING.
[16]  ⍝APPEND THIS EXPRESSION TO THE PROGRAM TREE.
[17]   NEXT R
[18]   →L1
[19]  ⍝APPEND THE END OF BRANCH SYMBOL.
[20]  L2:(¯1,(□SIZE 2)[2]-1) □REPLACE 1,(□SIZE 1)[2]-1
    ∇
```

```
      ∇GETL[□]∇
    ∇ R←GETL X
[1]   R←(-ρX)↓(ρX)↑⍞,⍞←X←,X
    ∇

      ∇LAST[□]∇
    ∇ R←LAST Y;Z;W
[1]   Z←□READ 1,|Y
[2]   W←2,(',',=Z)/ιρZ
[3]   →(Y>0)↑L1
[4]   R←(1↑¯2↑W)↓¯1↓Z
[5]   →0
[6]  L1:R←(1↑¯3↑W)↓¯1↓(1↑¯2↑W)↑Z
    ∇

      ∇MAIN[□]∇
    ∇ MAIN;M;S;OSW;R;PROG;STACK;PIVOTS
[1]   PIVOTS←ι0
[2]   M←1
[3]   FIRSTEG
[4]  L0:→('N'=1↑GETL 'ANY MORE EXAMPLES (Y/N)?')↑L2
[5]  L4:NEXTEG
[6]   →L0
[7]  L2:→('N'=1↑GETL 'DO YOU WANT TO TRY IT OUT?(Y/N)')↑L3
[8]   PRINTPROG
[9]   F
[10]  R←3 □FD F
[11]  PROG
[12]  →L2
[13] L3:→('Y'=1↑GETL 'ANY MORE EXAMPLES (Y/N)?')↑L4
    ∇
```

```
      ∇NEXT[⎕]∇
    ∇ NEXT R;TEMP;Z
[1]   ⍝R IS THE POSITION OF THE OPERATOR IN THE LITERAL INPU
      T STRING.
[2]   ⍝OSW=1 IF OPERAND COMES FROM STACK.
[3]    →(L2,L3,L4,L5,L6,L1)['+⎕+o→'⍳EG[R]]
[4]   ⍝PUSH ACC ONTO STACK
[5]   L2:→L7,STACK←(⎕FI ACC),STACK
[6]   ⍝DISPLAY TOP POSITION OF STACK
[7]   L3:→0,⍴DISPLAY
[8]   ⍝DROP TOP POSITION OF STACK
[9]   L4:→L7,STACK←1↓STACK
[10]  ⍝CLEAR ACC
[11]  L5:→L7,⍴ACC←'0'
[12]  ⍝ROTATE STACK ONE POSITION
[13]  L6:→L7,STACK←1⌽STACK
[14]  ⍝TEMP IS LITERAL STRING ARITHMETIC EXPRESSION.
[15]  L1:→OSW↑L10
[16]   →L11,⍴TEMP←ACC,EG[R],(R-1)↑EG
[17]  L10:TEMP←ACC,EG[R],Z←,'E15.10' ⎕FMT 1↑STACK
[18]  ⍝EXECUTE TEMP AND DISPLAY.
[19]  L11:⎕←ACC←XEQ TEMP
[20]   →OSW↑L12
[21]  ⍝APPEND OPERATOR, OPERAND TYPE, OPERAND, ',', RESULT,
      ',',
[22]   (EG[R],'0',((R-1)↑EG),',',(ACC←,'E15.10' ⎕FMT ACC),',
      ') ⎕APPEND 1
[23]  ⍝SET UP NEW POINTERS.
[24]  L8:(((⎕SIZE 1)[2]+1),(⎕SIZE 2)[2]) ⎕APPEND 1
[25]   (0,M) ⎕APPEND 2
[26]   →0
[27]  ⍝NULLARY: APPEND OPERATOR ONLY
[28]  L7:(EG[R]) ⎕APPEND 1
[29]   →L8
[30]  L12:(EG[R],'2',Z,',',(ACC←,'E15.10' ⎕FMT ACC),',')
       ⎕APPEND 1
[31]   →L8
    ∇
```

```
      ∇NEXTEG[□]∇
    ∇ NEXTEG;J;ACC;R;X;K;TEMP;EG;B
[1]   ⍝M IS EXAMPLE NUMBER (GLOBAL)
[2]    'ENTERING EXAMPLE ';M←M+1
[3]    STACK←⍳0
[4]    ACC←'0'
[5]    J←2⍴0
[6]   L0:J←POINTER 1↑J
[7]   L1:→(L0,L1+1)[PROMPT]
[8]    →('END'∧.=3↑EG)↑L14
[9]    →('CANCEL'∧.=6↑EG)↑L8
[10]  ⍝ENTER RETURNS A ZERO SCALAR IF AN ERROR IS DETECTED I
       N THE INPUT STRING.
[11]   →(0=R←ENTER EG)↑L1
[12]  ⍝IF THIS IS A DISPLAY REQEST, DO SO AND THEN GET NEXT
       INPUT LINE.
[13]   →(EG[P]='□')↑L21
[14]   →L1,⍥DISPLAY
[15]  ⍝J[1] IS COMPONENT NUMBER IN FILE 1 OF NEXT OPERAND IN
        THIS EXAMPLE.
[16]  ⍝J[2] IS RELATED COMPONENT NUMBER IN FILE 2.
[17]  ⍝IF J[1]=¯1, PREVIOUS EXAMPLES ENDED HERE. CREATE A NE
       W BRANCH.
[18]  L21:→(¯1=1↑J)↑L4
[19]  ⍝IF NULLARY OPERATOR EXPECTED,
[20]  ⍝SKIP OPERAND CHECKS
[21]   →(1=⍥X←□READ 1,1↑J)↑L9
[22]  ⍝IF OPERAND IS A VARIABLE CHECK NO FURTHER.
[23]   →((~OSW)∧'1'=X[2])↑L2
[24]  ⍝CHECK WHETHER THIS IS A STACK VARIABLE
[25]   →('2'≠X[2])↑L20
[26]   →(L4,L2)[OSW+1]
[27]  ⍝IF NEXT OPERAND IS A CONSTANT AND IS THE SAME AS ALL
       OF THE PRECEDING CONSTANTS, CHECK NO FURTHER.
[28]  L20:→((□FI LAST 1↑J)=□FI(R-1)↑EG)↑L2
[29]   →('C'=1↑GETL 'CONSTANT OR VARIABLE (C/V)?')↑L4
[30]  ⍝CHANGE SWITCH TO VARIABLE INDICATOR.
[31]   X[2]←'1'
[32]  L2:X←X,((R-1)↑EG),','
[33]   ((□READ 2,J[2]),M) □REPLACE 2,J[2]
[34]  ⍝CHECK OPERATOR NEXT.
[35]  L9:→(EG[P]≠1↑X)↑L10
[36]   →(L22,L23,L24,L25,L26)['↑↓⍥→'⍳EG[R]]
[37]  L22:→L0,STACK←(□FI ACC),STACK
[38]  L23:→L0,STACK←1↓STACK
[39]  L24:→L0,⍴ACC←'0'
[40]  L25:→L0,STACK←1⌽STACK
```

```
[41] ⍝ CHECK FOR STACK OPERAND
[42] L26:→(~OSW)↑L28
[43]  TEMP←ACC,EG[R],,'E15.10' ⎕FMT 1↑STACK
[44]  →L27
[45] ⍝TEMP IS ARITHMETIC EXPRESSION IN LITERAL STRING FORM.
[46] L28:TEMP←ACC,EG[R],(R-1)↑EG
[47] ⍝EXECUTE AND DISPLAY TEMP.
[48] L27:⎕←ACC←XEQ TEMP
[49] ⍝SAVE RESULT
[50] L30:(X←X,(ACC←,'E15.10' ⎕FMT ACC),',') ⎕REPLACE 1,1↑J
[51]  →L0
[52] ⍝REMOVE OPERAND INFO PREVIOUSLY STORED BECAUSE A BRANC
      H POINT HAS BEEN REACHED.
[53] L10:X←⎕READ 1,1↑J
[54] ⍝CREATE A NEW BRANCH.
[55] L4:→(0=SIZE J)↑L7
[56]  'CONTINUING'
[57]  →(~'END'∧.=3↑EG)↑L5
[58]  R←⎕READ 1,X←(⎕SIZE 1)[2]-1
[59]  TEMP←R↓X+1
[60]  P[TEMP]←¯1
[61]  P ⎕REPLACE 1,X
[62]  →0
[63] L5:NEXT R
[64] L6:→('END'∧.=3↑EG←GETL '      ')↑L11
[65]  →('CANCEL'∧.=6↑EG)↑L8
[66]  →(L6,L5)[1+0≠R←ENTER EG]
[67] ⍝ERROR IN SETTING UP OF CONDITION.
[68] L7:'DOES NOT TALLY WITH PAST EXAMPLES.'
[69] ⍝DISPLAY MTH EXAMPLE.
[70]  DUMP M
[71] ⍝DELETE MTH EXAMPLE.
[72] L8:DELETE M
[73]  →0
[74] L11:(¯1,(⎕SIZE 2)[2]-1) ⎕REPLACE 1,(⎕SIZE 1)[
      2]-1
[75]  →0
[76] L14:→(~¯1=1↑J)↑L4
     ∇
```

```
      ∇POINTER[□]∇
    ∇ R←POINTER I;J;X;X1
[1]   ⍝RETURNS A VECTOR OF LENGTH TWO.
[2]   ⍝R[1] POINTS TO COMPONENT IN FILE 1 CONTAINING THE NEX
      T OPERAND.
[3]   ⍝R[2] POINTS TO COMPONENT IN FILE 2 RELATED TO THE NEX
      T OPERAND.
[4]    →((2=⍴R)∧¯1=(R←,□READ 1,B←I+1)[1])↑0
[5]    →(2=⍴P)↑B←,0
[6]   ⍝A BRANCH POINT IN THE PROGRAM HAS BEEN REACHED.
[7]   ⍝THE CONDITIONAL IS EXECUTED TO DETERMINE WHICH BRANCH
        TO TAKE.
[8]   ⍝RETURNS NEXT OPERAND ALONG THE APPROPRIATE BRANCH.
[9]   L1:→(0=Y←¯1↑X←□READ 1,P+1)↑L2
[10]   Z←LAST Y
[11]   →L3
[12]  L2:Z←,'F15.10' □FMT Y[7]
[13]  L3:X1←Z,(□READ 1,P),LAST X[6]
[14]   J←2-XEQ X1
[15]   B←R+1
[16]   R←X[J,J+2]
[17]  ⍝TEST FOR ANOTHER BRANCH
[18]   →(0=R[2])↑0
[19]   →L1,R←R[1]
    ∇
```

```
      ∇PRINTPROG[□]∇
    ∇ PRINTPROG;Z;STACK;K;LIM;P;T;Y;P;X;A1;A2;A3;A4
[1]   ⍝CREATE STACK OF BRANCH POINTS.
[2]    STACK←⍳P←0
[3]    Z←(~Z=' ')/Z←,'□;O□,I3' □FMT PIVOTS
[4]    F← 2 50 ⍴(50↑'R←PROG;T;S',Z),50↑'S←⍳T←0'
[5]    K←1
[6]    Z←''
[7]    LIM←(□SIZE 1)[2]
[8]   L0:→(LIM≤K)↑0
[9]   ⍝BRANCH POINT OR LEAF?
[10]   →(¯1=(R←,□READ 1,K)[1])↑LEAF
[11]  ⍝REGULAR CONTROL
[12]   →(2=⍴R)↑L1
[13]  ⍝SAVE LEFT BRANCH ON STACK.
[14]  L16:STACK←STACK,(T←□READ 1,P+1)[1 3],P
[15]  ⍝SET UP CONDITIONAL STATEMENT.
[16]   →(0=T[3])↑L7
[17]   Z←Z,'→(O',(,'I3' □FMT(|T[8])+T[8]<0),(□READ 1,R),'O',
       (,'I3' □FMT(|T[6])+T[6]<0),(')↑L'),,'I3' □FMT T[1]
[18]   →L8
[19]  L7:Z←Z,'→(',(,'E15.10' □FMT T[7]),(□READ 1,R),'O',(,'I
       3' □FMT(|T[6])+T[6]<0),(')↑L'),,'I3' □FMT T[1]
[20]  L8:→(T[1]=¯1)↑L14
[21]  ⍝DELETE BLANKS.
[22]   P←P,[1] 50↑(~Z=' ')/Z
[23]   Z←''
[24]  ⍝TAKE RIGHT BRANCH.
[25]   R←T[2 4],P
[26]  L1:→(R[1]=¯1)↑LEAF
[27]   →(R[2]=0)↑L15
[28]   R←R,P
[29]   A1←''
[30]   A2←A3←'T'
[31]   →(~R[1]∈PIVOTS)↑L17
[32]   A1←'O',(,'I3' □FMT R[1]),'←'
[33]  L17:→(~(R[1]+1)∈PIVOTS)↑L18
[34]   A2←'O',,'I3' □FMT R[1]+1
[35]  L18:→(~R[3]∈PIVOTS)↑L19
[36]   A3←'O',,'I3' □FMT R[3]
[37]  L19:→(L9,L10,L11,L12,L13)['↑↓O→'⍳1↑Y←□READ 1,P[1]]
[38]  L9:→L4,⍴Z←Z,'S←',A3,',S'
[39]  L10:Z←Z,'S←1↓S'
[40]   →L4
```

```
[41] L11:Z←Z,A2,'←0'
[42]  →L3
[43] L12:Z←Z,'S←1φS'
[44]  →L4
[45] ⍝IS OPERAND A CONSTANT, VARIABLE, OP FROM THE STACK?
[46] L13:→(L2,L5,L6)['012'⍳Y[2]]
[47] ⍝VARIABLE ASSIGNMENT STATEMENT
[48] L5:Z←Z,A2,'←',A3,Y[1],A1,'⍞'
[49] L3:P←R[1]+1
[50] ⍝DELETE BLANKS AND PRINT
[51] L4:F←F,[1] 50↑(~Z=' ')/Z
[52]  Z←''
[53]  →L0,K←R[1]+1
[54] LEAF:A4←'T'
[55]  →(~P∈PIVOTS)↑L20
[56]  A4←'0',,'I3' ⍞FMT P
[57] L20:Z←'→0,R←',A4
[58]  F←F,[1] 50↑(~Z=' ')/Z
[59] ⍝POP STACK.
[60]  →(0=⍴STACK)↑0
[61]  P←¯1↑R←¯3↑STACK
[62]  STACK←¯3↓STACK
[63]  Z←'L',(X←,'I3' ⍞FMT R[1]),':'
[64]  →L1
[65] ⍝CONSTANT ASSIGNMENT STATEMENT.
[66] L2:Z←Z,A2,'←',A3,Y[1],2↓((Y⍳',')-1)↑Y
[67]  →L3
[68] ⍝STACK ASSIGNMENT STATEMENT
[69] L6:Z←Z,A2,'←',A3,Y[1],'1↑S'
[70]  →L3
[71] L14:A4←'T'
[72]  →(~P∈PIVOTS)↑L21
[73]  A4←'0',,'I3' ⍞FMT P
[74] L21:Z←(¯4↓Z),'0,R←',A4
[75]  STACK←¯3↓STACK
[76]  →1↑L8
[77] L15:R←P[1]
[78]  →L16
     ∇
```

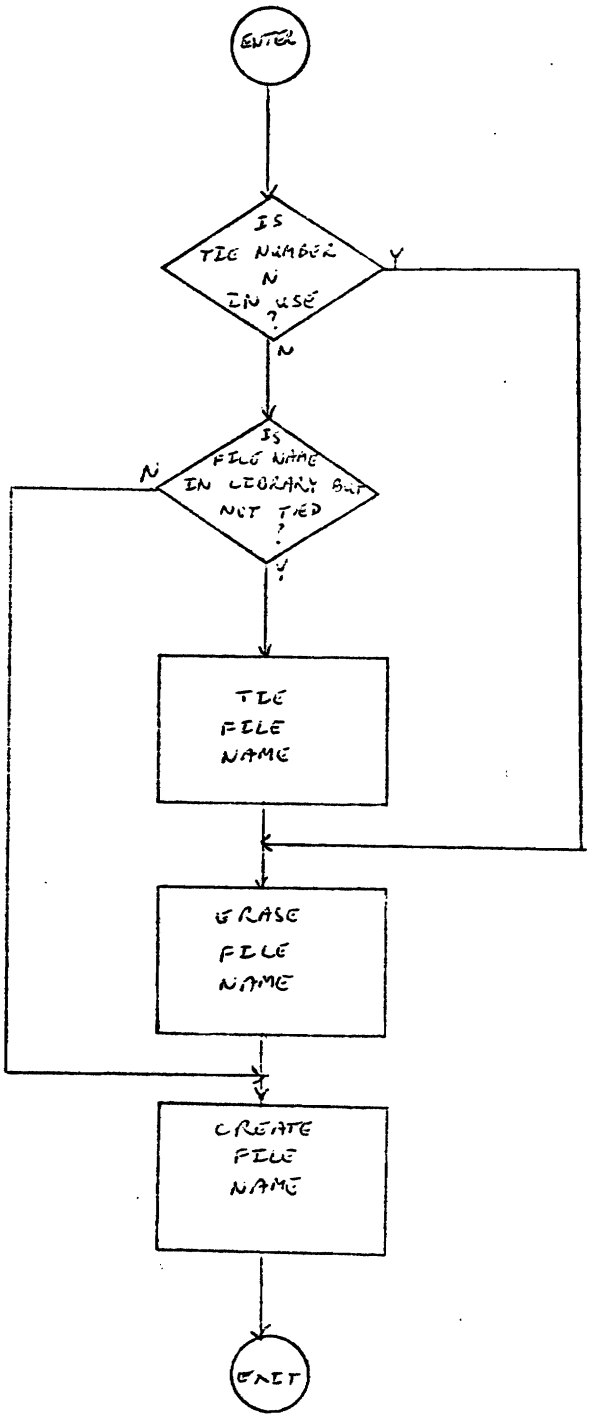
```
      ∇PROMPT[□]∇
    ∇ R←PROMPT;X;Y;I
[1]   →(¯1≠1↑J)↑L1
[2]   →(CR≠1↑EG←,GETL 'END      ')↑0,R←2
[3]   →L7,⍴EG←'END'
[4]  L1:Y←□READ 1,1↑J
[5]   ⍝IS THIS A NULLARY OPERATOR?
[6]   →(1<⍴Y)↑L2
[7]   →(CR≠1↑EG←,GETL Y,'      ')↑0,R←2
[8]   →(L8,L9,L10,L11)['↑↓○→'⍳Y]
[9]  L8:→L6,STACK←(□FI ACC),STACK
[10] L9:→L6,STACK←1↓STACK
[11] L10:→L6,⍴ACC←'0'
[12] L11:→L6,STACK←1⌽STACK
[13] ⍝UNARY OPERATOR. CHECK OPERAND
[14] L2:→(L3,L4,L5)['012'⍳Y[2]]
[15] ⍝OPERAND IS A CONSTANT
[16] L3:→(CR≠1↑EG←,GETL(X←LAST 1↑J),Y[1],'      ')↑0,R←
      2
[17]  →L41
[18] ⍝OPERAND IS FROM STACK
[19] L5:→(CR≠1↑EG←,GETL 'ω',Y[1],'      ')↑0,R←2
[20]  →L41,⍴X←,'E15.10' □FMT 1↑STACK
[21] ⍝OPERAND IS A VARIABLE
[22] L4:X←EG←,GETL '?',1↑Y
[23]  →(0=⍴I←(EG∈OPRS)/⍳⍴EG)↑0,R←2
[24]  →(∧/EG∈NUM)↑L41
[25]  'INVALID NUMERIC CHARACTER. RE-ENTER.'
[26]  →L4
[27] L41:□←ACC←XEQ ACC,Y[1],X
[28]  ACC←,'E15.10' □FMT ACC
[29] REPLACE:(Y,X,',',ACC,',') □REPLACE 1,J[1]
[30] L6:R←1
[31] L7:((□READ 2,J[2]),M) □REPLACE 2,J[2]
    ∇
```

```
      ∇SIG[□]∇
    ∇ R←SIG J;K;Y;X;S;Z;T;P;Q;N;O;E;TEMP;T1;S1
[1]   K←2⍴R←0
[2]   TEMP←B
[3]   ⍝J[1] IS COMPONENT NUMBER OF OPERAND AT BRANCH.
[4]   ⍝K[1] WILL HAVE VALUE OF COMPONENT NUMBER OF CONSECUTI
      VE OPERANDS.
[5]   L1:→(J[1]=(K←POINTER 1↑K)[1])↑0
[6]   ⍝SKIP NULLARY OPERATORS.
[7]    →(1=⍴Q←□READ 1,1↑K)↑L1
[8]   ⍝DISPLAY OPERAND AND ASK IF IT IS SIGNIFICANT.
[9]   L5:→('Y'=1↑GETL(O←LAST 1↑K),'  (Y/N)?')↑L2,T1← 1 0
[10]  ⍝DISPLAY INTERMEDIATE RESULT AND ASK IF IT IS SIGNIFIC
      ANT.
[11]   →('N'=1↑GETL(O←LAST(-1↑K)),'  (Y/N)?')↑L1,T1← 0 1
[12]  L2:PIVOTS←PIVOTS,K+T1[1]=0
[13]   →(¯1=J[1])↑L7,B←TEMP
[14]   →(0≠B)↑L6
[15]  ⍝P IS THE POINTER VECTOR WHICH PRECEDES THE OPERAND.
[16]   P←□READ 1,J[1]-1
[17]  ⍝REPLACE IT BY A BRANCH POINTER.
[18]   ((□SIZE 1)[2]) □REPLACE 1,J[1]-1
[19]  ⍝Y IS LITERAL STRING REPRESENTATION OF PIVOT VALUE.
[20]  L3:Y←GETL 'ENTER PIVOT VALUE.'
[21]  ⍝COMPARING TWO OPERANDS?
[22]   →('α'=1↑Y)↑L10
[23]   →(∨/~Y∈NUM)↑ERR3
[24]   →(1<⍴T←□FI E←Y)↑ERR5
[25]   T←T,0
[26]  ⍝T IS PIVOT VALUE IN NUMERIC FORM.
[27]  L4:X←1↑GETL 'ENTER RELATION.'
[28]   →(~X∈ROPRS)↑ERR6,S←0
[29]  ⍝Z IS CONDITIONAL EXPRESSION IN LITERAL STRING FORM.
[30]  ⍝CHECK THAT CONDITION IS EITHER TRUE FOR LAST EXAMPLE
      ONLY OR FALSE FOR LAST EXAMPLE ONLY.
[31]   Q[(Q=',')/⍳⍴Q←¯1↓2↓Q]←' '
[32]   Q←((⍴Q)⍴T1)/Q←□FI Q
[33]   Z←'(',E,X,(Q←,'E15.10' □FMT Q←((1↓□READ 2,K[
       2])∈1↓□READ 2,J[2])/Q),')',~',Y,X,Q
[34]  ⍝S IS USED AS SWITCH TO INDICATE WHICH BRANCH IS TRUE
      AND WHICH IS FALSE.
[35]   S←S+2×(∧/XEQ Z)
[36]   Z←'(~',E,X,Q,'),',Y,X,Q
[37]   →(0=S←S+∧/XEQ Z)↑0
[38]  ⍝SET UP BRANCH INFORMATION.
[39]   K[1]←((T1[1]=0)×(-K[1]))+(T1[1]=1)×K[1]
[40]   X □APPEND 1
```

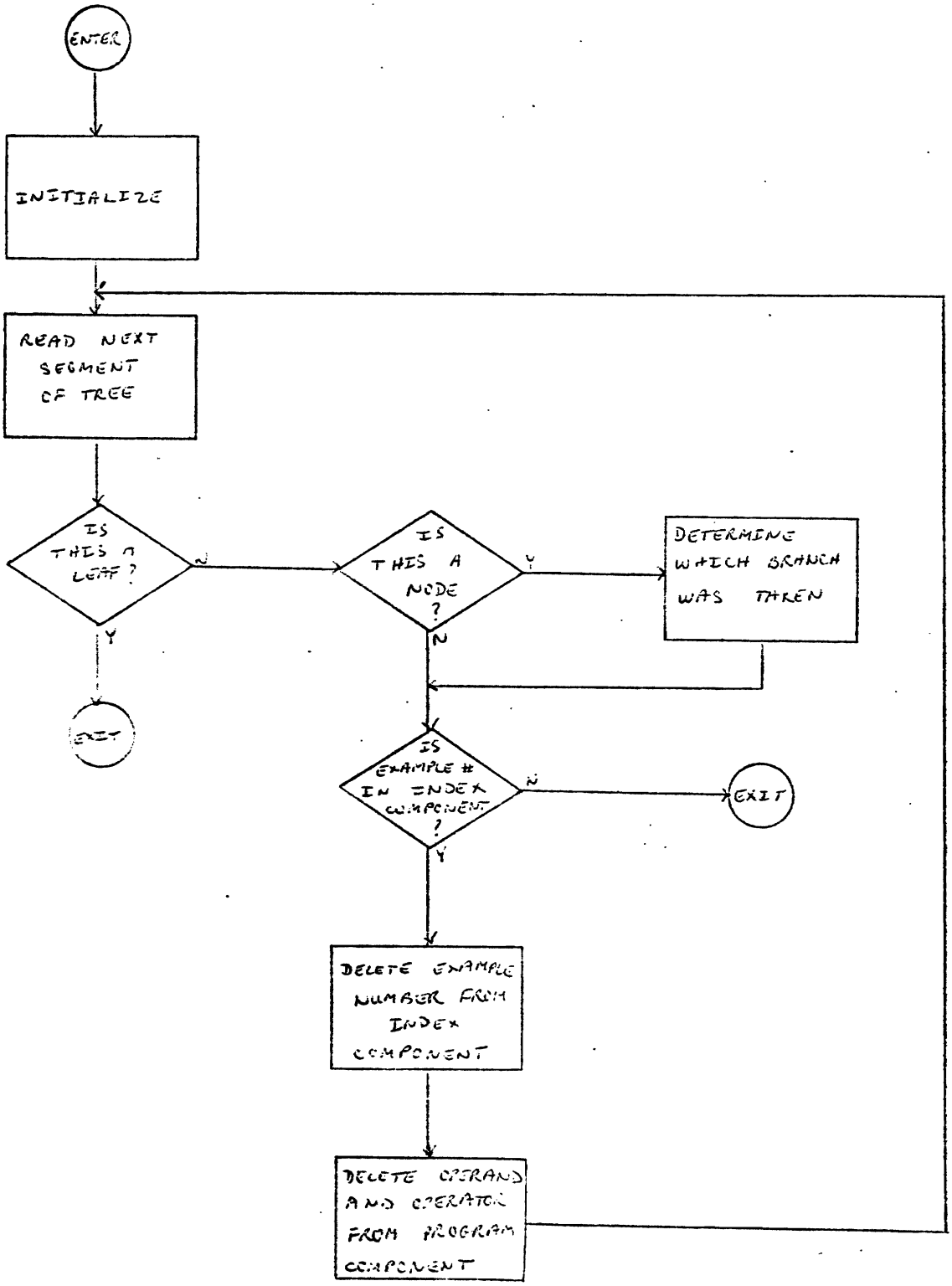
```
[41]  ((SΦJ[1],(⎕SIZE 1)[2]+1),(SΦJ[2],(⎕SIZE 2)[2]),(1↑P),
      K[1],T) ⎕APPEND 1
[42]  (0,M) ⎕APPEND 2
[43]  →0,R←1
[44] L6:P←⎕READ 1,B
[45] ⍝CHECK WHICH BRANCH WAS TAKEN
[46]  Y←(P[1 2]⍳J[1])-1
[47]  J←P[Y+1,3]
[48]  P[Y+1,3]←(⎕SIZE 1)[2],0
[49]  P ⎕REPLACE 1,B
[50]  →L3
[51] ERR3:→L3,⍴⎕←'INVALID NUMERIC CHARACTER.'
[52] ERR5:→L3,⍴⎕←'ONE VALUE ONLY.'
[53] ERR6:→L4,⍴⎕←'INVALID OPERATOR.'
[54] L10:N←,X[1]
[55]  →(T1[1]=1)↑L12
[56] L11:→(J[1]=(N←POINTER 1↑N)[1])↑0
[57] ⍝SKIP NULLARY OPERATORS
[58]  →(1=⍴⍴←⎕READ 1,N[1])↑L11
[59] ⍝DISPLAY OPERAND AND ASK IF IT IS SIGNIFICANT.
[60] L14:→('Y'=1↑GETL(Y←LAST N[1]),'  (Y/N)?')↑L13,S1← 1 0
[61] ⍝DISPLAY INTERMEDIATE RESULT AND ASK IF IT IS SIGNIFIC
      ANT.
[62] L12:→('N'=1↑GETL(Y←LAST(-N[1])),'  (Y/N)?')↑L11,S1← 0
      1
[63] ⍝MAKE T A TWO ELEMENT VECTOR POINTING TO SECOND SIGNIF
      ICANT OPERAND.
[64] L13:T←Y
[65]  PIVOTS←PIVOTS,N[1]+S1[1]=0
[66]  →L4,T←0,N[1]←((S1[1]=0)×(-N[1]))+(S1[1]=1)×N[1]
[67] L7:P←⎕READ 1,B
[68]  ((⎕SIZE 1)[2]) ⎕REPLACE 1,B
[69]  →L3
    ∇
```

APPENDIX II

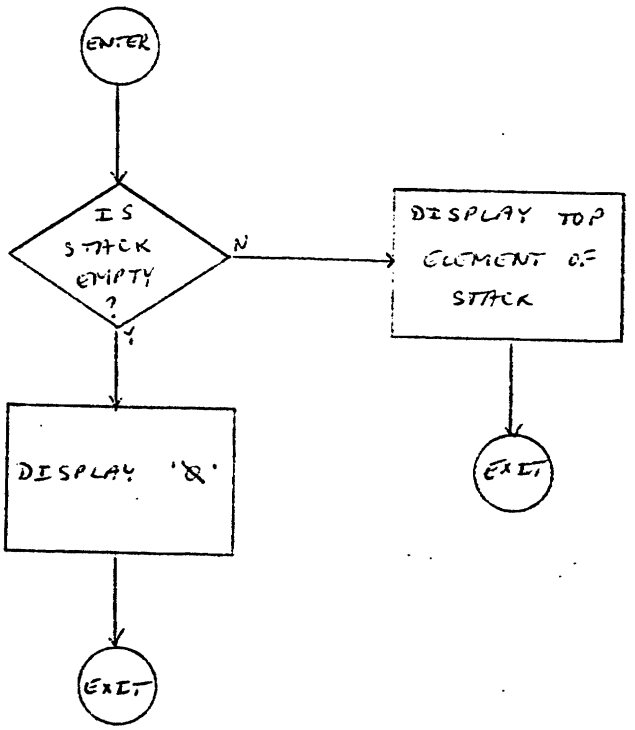
PROGRAM FLOWCHARTS



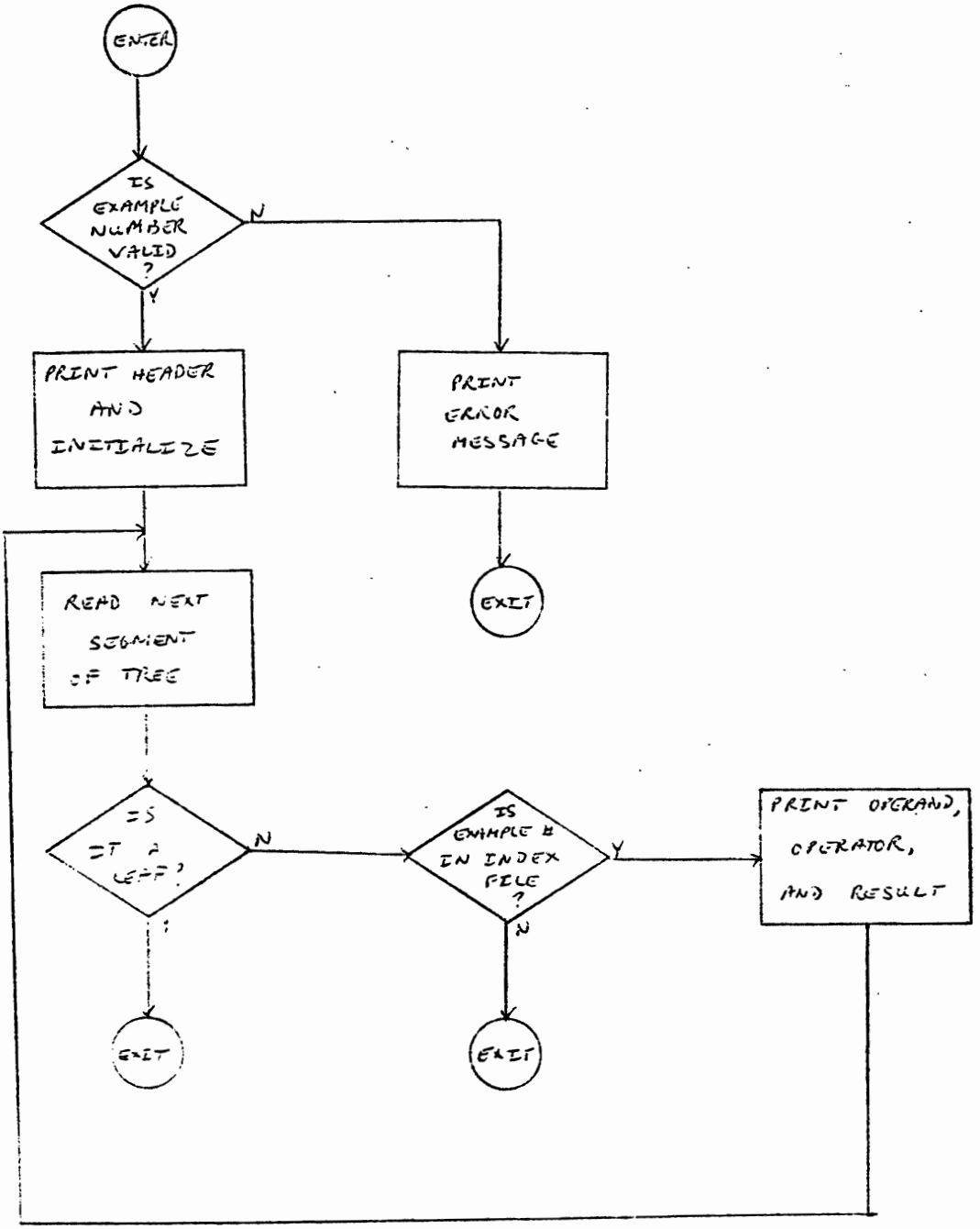
FNM ABSCREATE N


ENTER
INITIALIZE
READ NEXT SEGMENT OF TREE
IS THIS A LEAF ?
N
Y
IS THIS A NODE ?
Y
N
DETERMINE WHICH BRANCH WAS TAKEN
EXIT
IS EXAMPLE # IN INDEX COMPONENT ?
N
Y
EXIT
DELETE EXAMPLE NUMBER FROM INDEX COMPONENT
DELETE OPERAND AND OPERATOR FROM PROGRAM COMPONENT

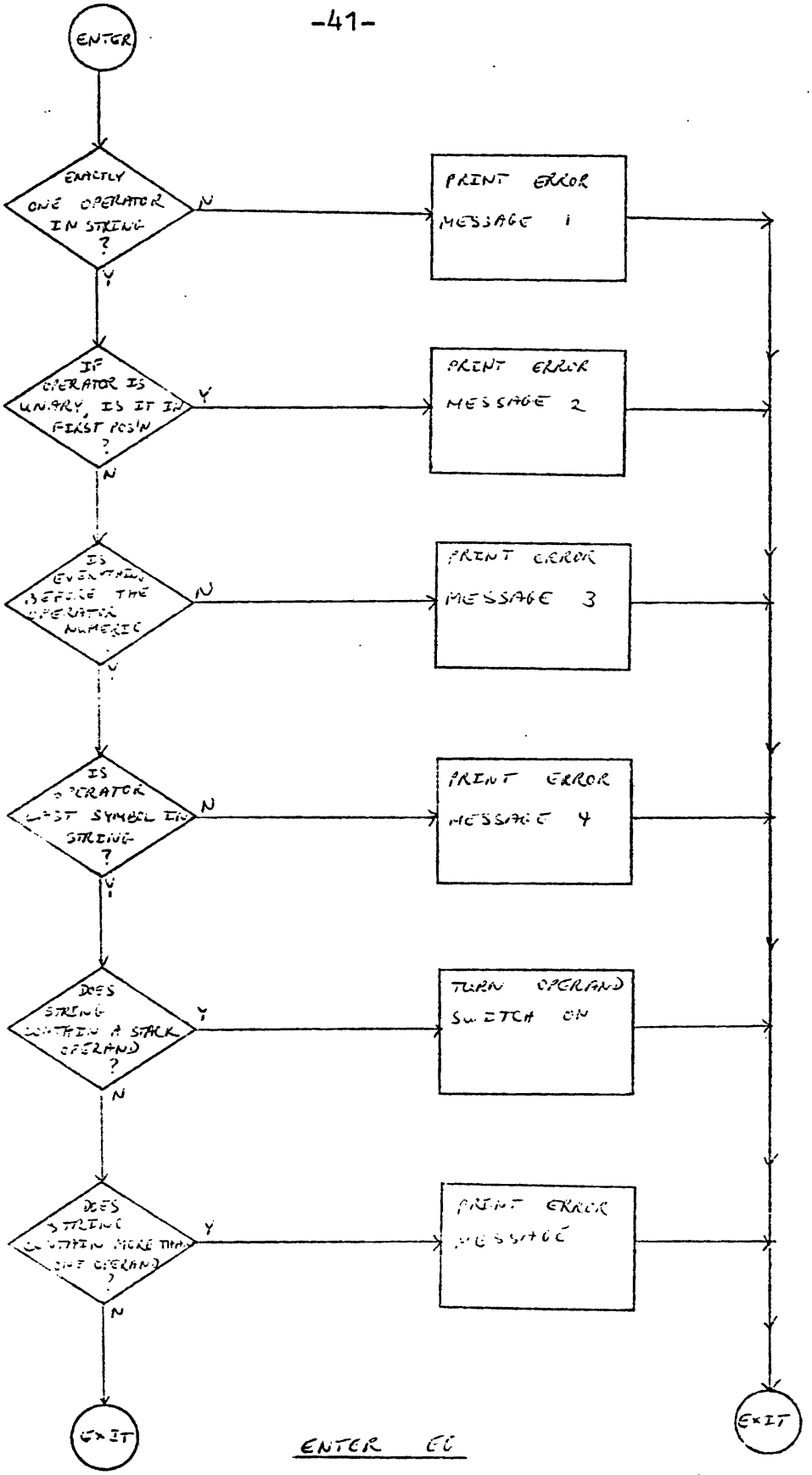
DELETE I


ENTER
IS STACK EMPTY ?
N
Y
DISPLAY TOP ELEMENT OF STACK
EXIT
DISPLAY '&'
EXIT

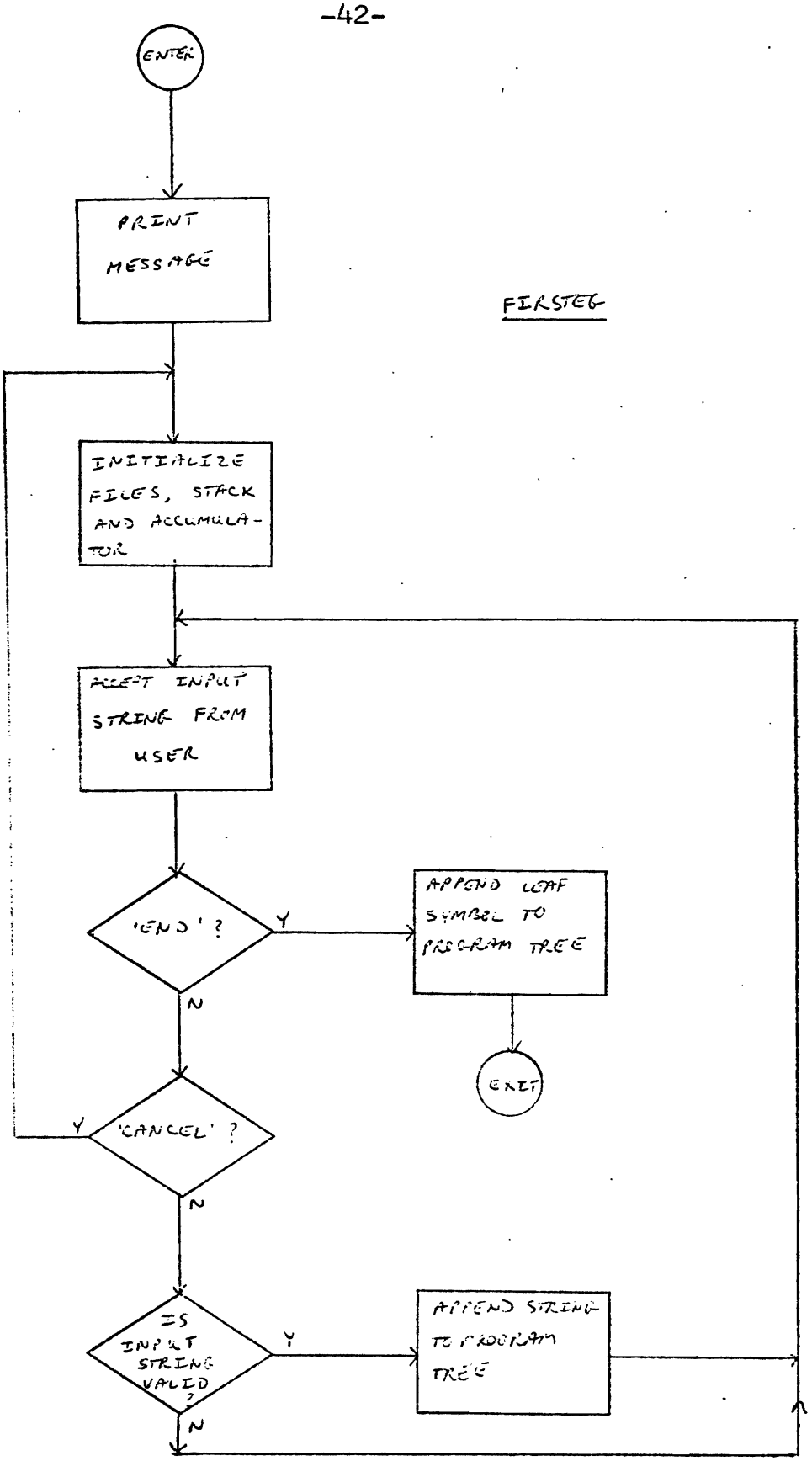
DISPLAY


ENTER
IS EXAMPLE NUMBER VALID ?
N
Y
PRINT HEADER AND INITIALIZE
PRINT ERROR MESSAGE
EXIT
READ NEXT SEGMENT OF TREE
IS IT A LEAF ?
N
IS EXAMPLE # IN INDEX FILE ?
Y
PRINT OPERAND, OPERATOR, AND RESULT
N
EXIT
EXIT

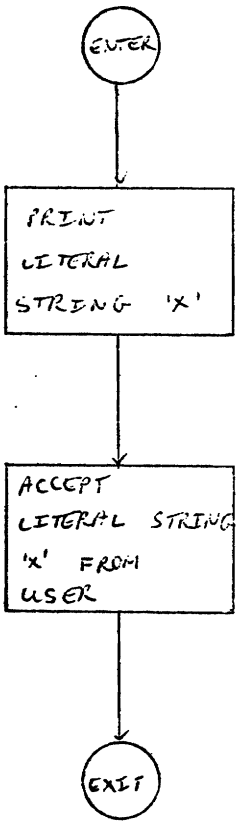
DUMP I


ENTER
EXACTLY ONE OPERATOR IN STRING ?
N
PRINT ERROR MESSAGE 1
Y
IF OPERATOR IS UNARY, IS IT IN FIRST POS'N ?
Y
PRINT ERROR MESSAGE 2
N
IS EVERYTHING BEFORE THE OPERATOR NUMERIC
N
PRINT ERROR MESSAGE 3
Y
IS OPERATOR LAST SYMBOL IN STRING ?
N
PRINT ERROR MESSAGE 4
Y
DOES STRING CONTAIN A STACK OPERAND ?
Y
TURN OPERAND SWITCH ON
N
DOES STRING CONTAIN MORE THAN ONE OPERAND ?
Y
PRINT ERROR MESSAGE
N
EXIT
EXIT

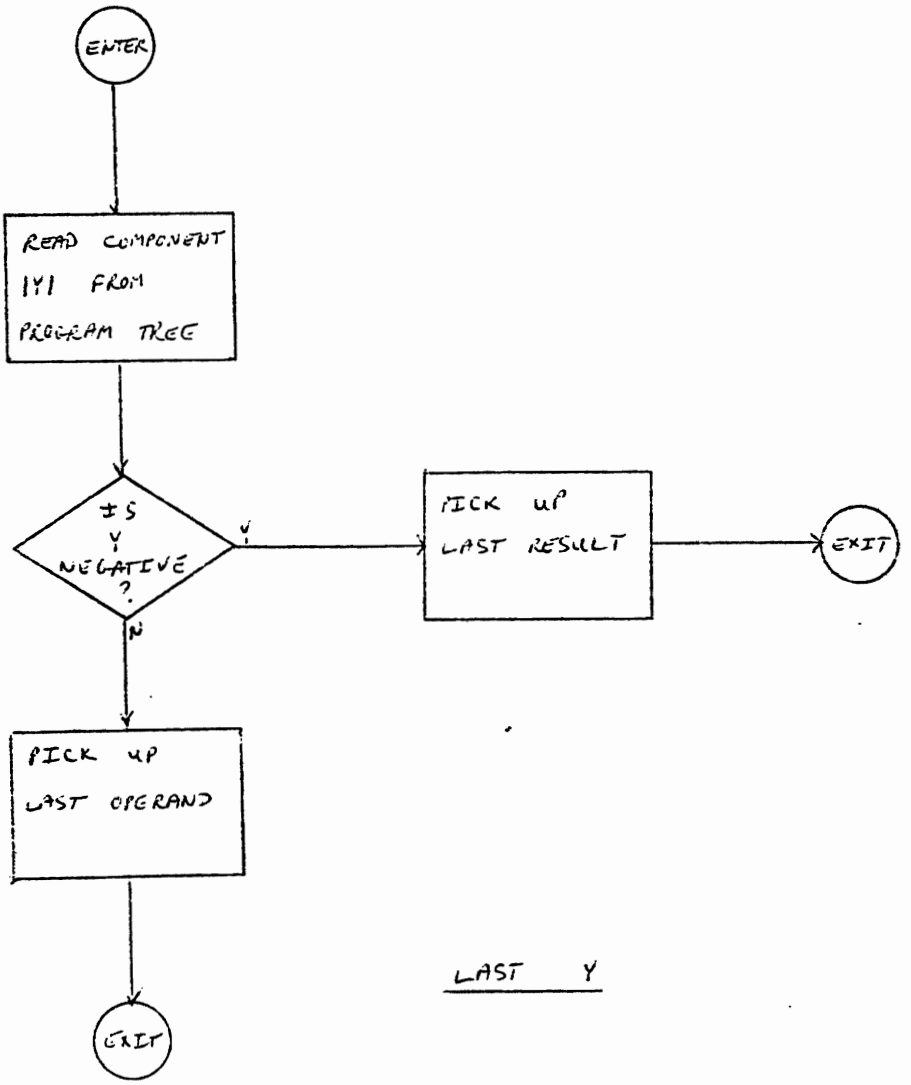
ENTER EC


ENTER
PRINT MESSAGE
INITIALIZE FILES, STACK AND ACCUMULATOR
ACCEPT INPUT STRING FROM USER
'END'?
Y
N
APPEND LEAF SYMBOL TO PROGRAM TREE
EXIT
'CANCEL'?
Y
N
IS INPUT STRING VALID?
Y
N
APPEND STRING TO PROGRAM TREE

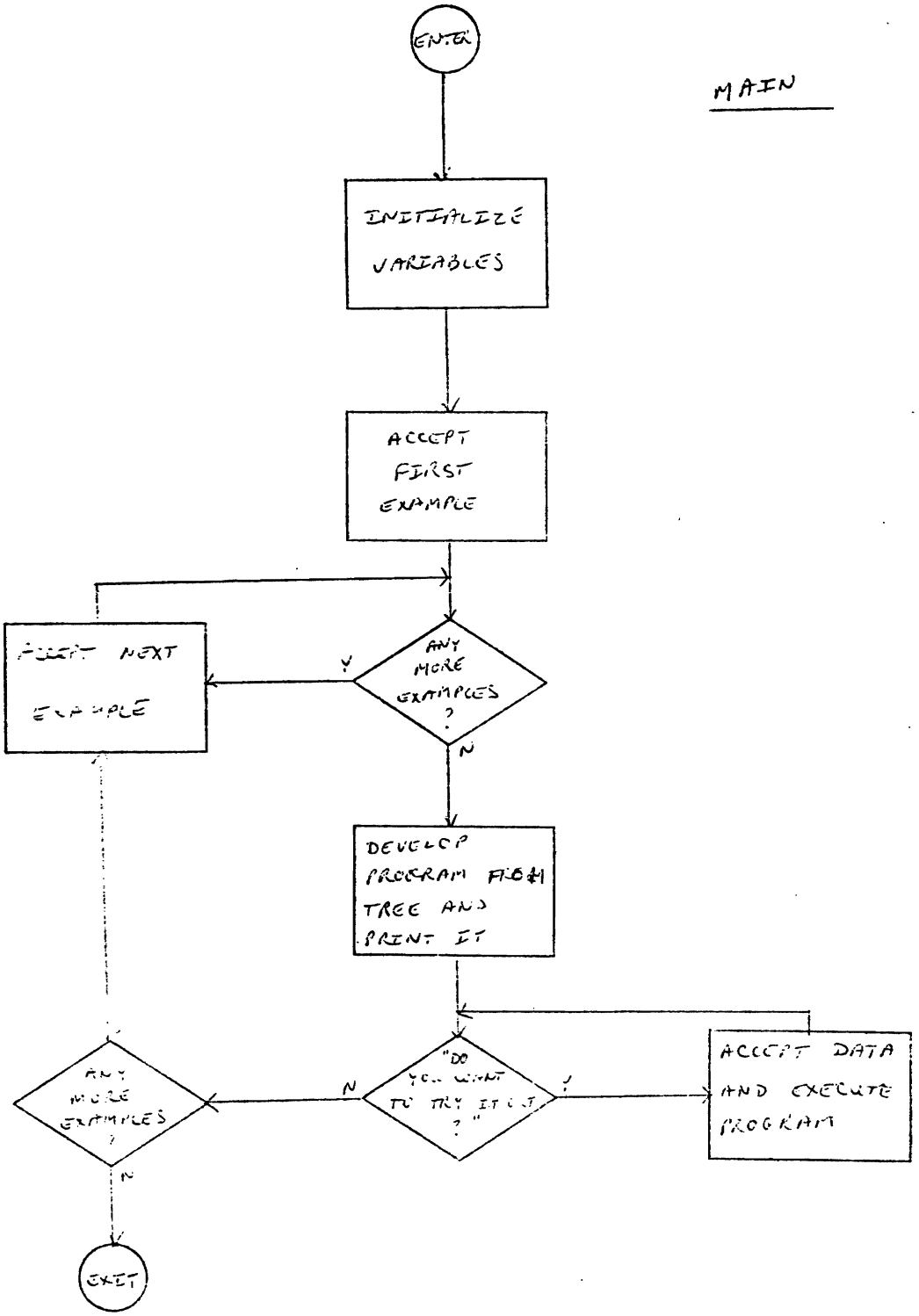
FIRSTEG

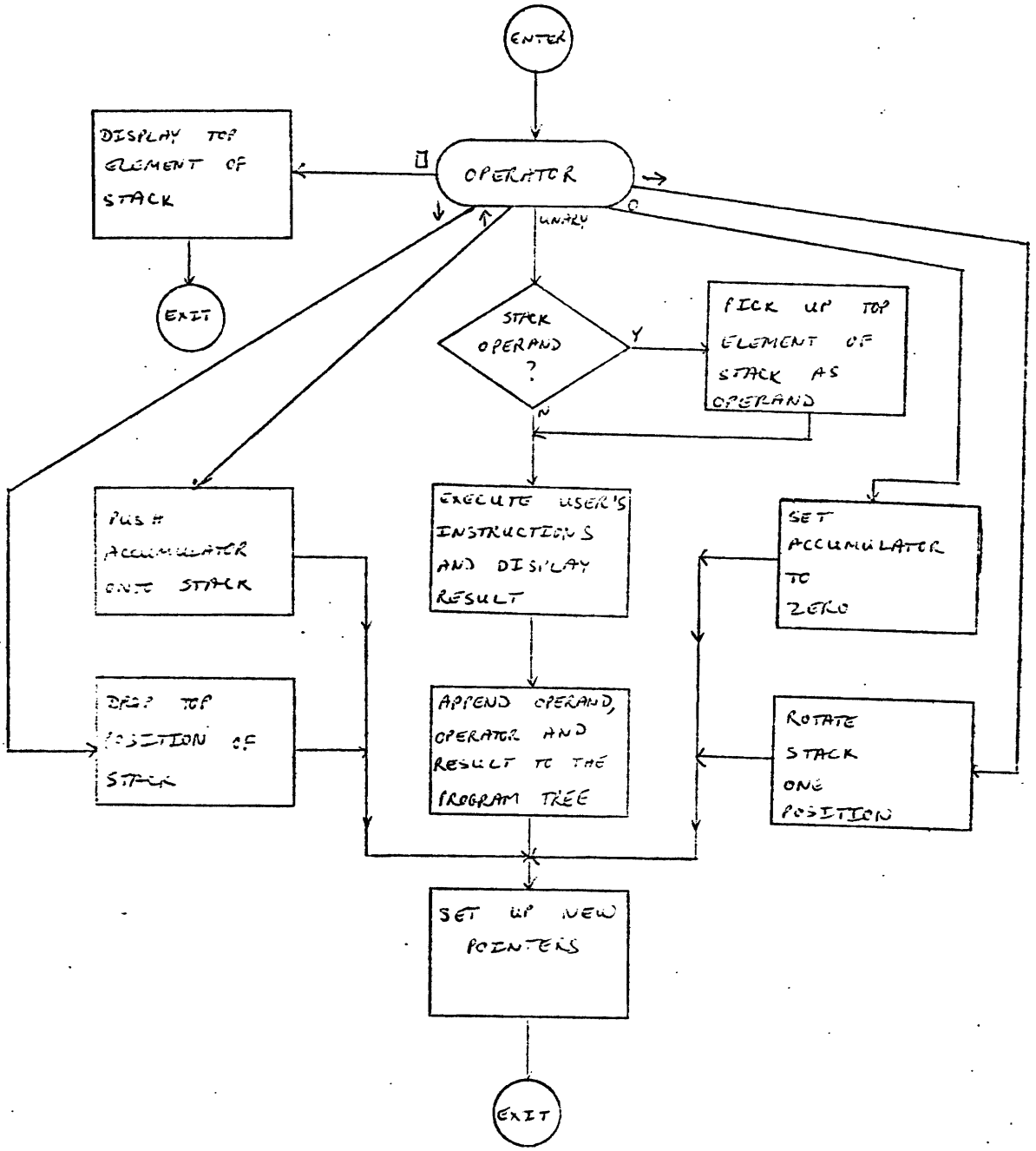

ENTER
PRINT
LITERAL
STRING 'X'
ACCEPT
LITERAL STRING
'X' FROM
USER
EXIT

GETL X

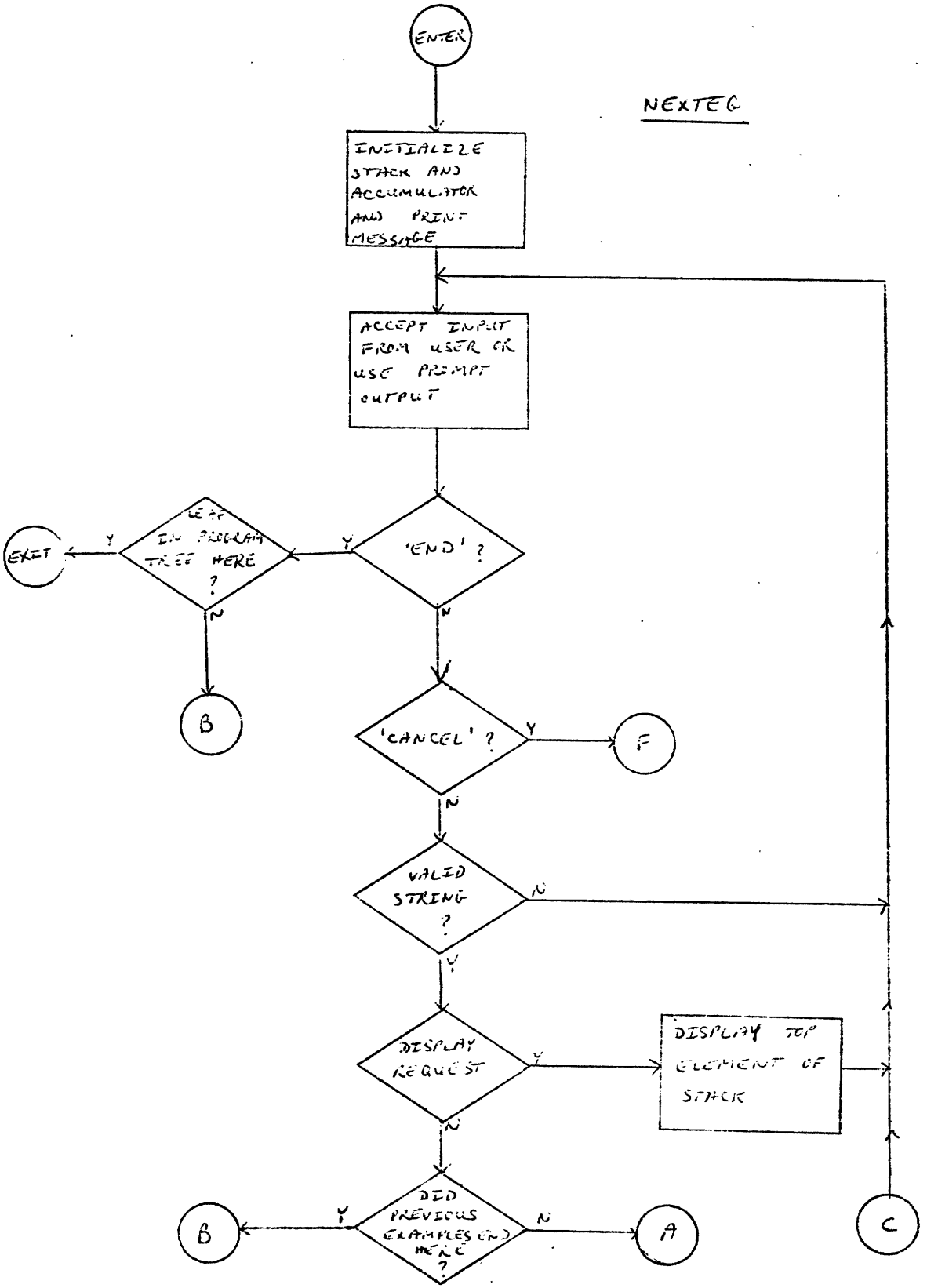

ENTER
READ COMPONENT
|Y| FROM
PROGRAM TREE
IS
Y
NEGATIVE
?
Y
N
PICK UP
LAST RESULT
EXIT
PICK UP
LAST OPERAND
EXIT

LAST Y

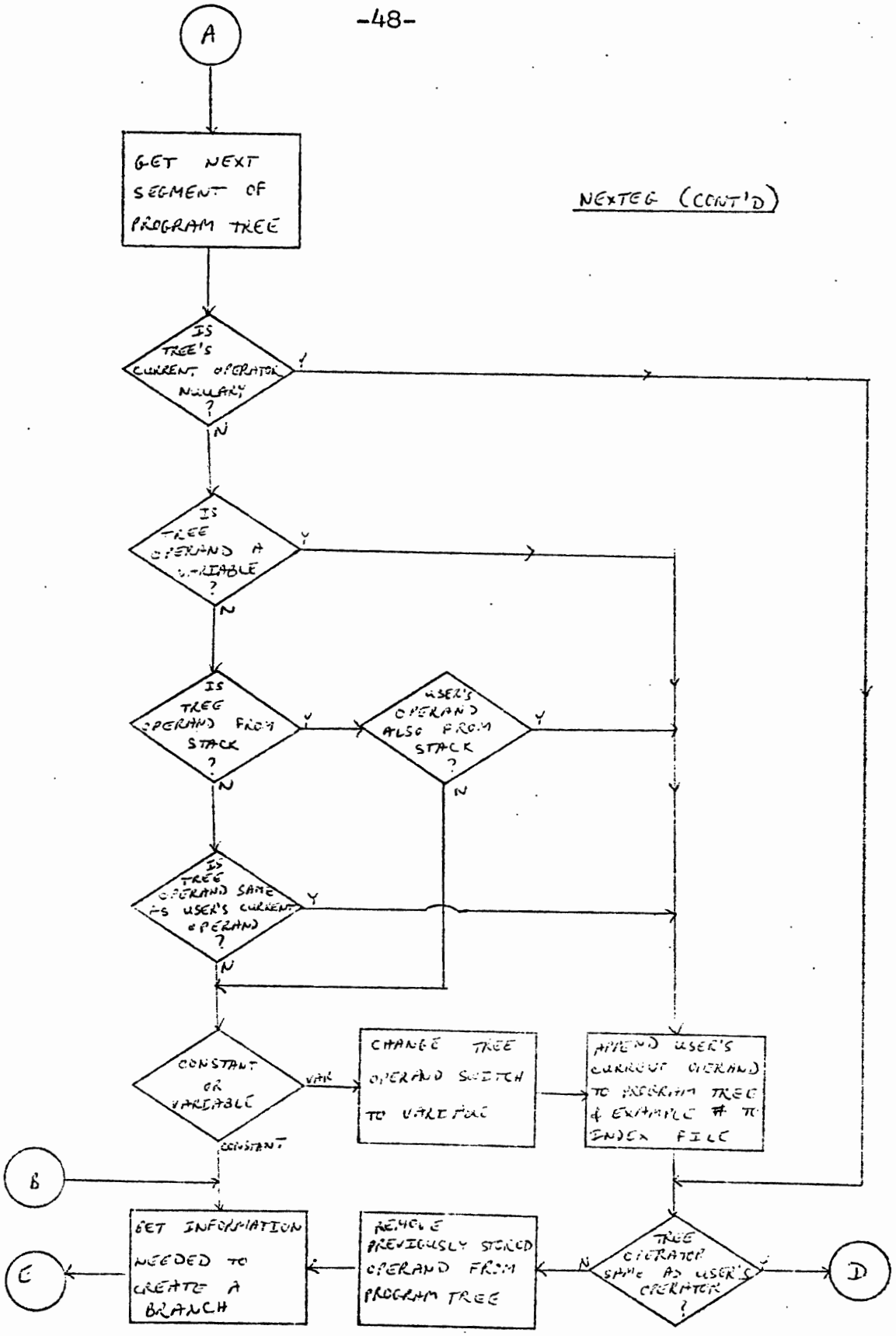

MAIN
ENTER
INITIALIZE VARIABLES
ACCEPT FIRST EXAMPLE
ANY MORE EXAMPLES ?
Y
ACCEPT NEXT EXAMPLE
N
DEVELOP PROGRAM FROM TREE AND PRINT IT
"DO YOU WANT TO TRY IT OUT ?"
Y
ACCEPT DATA AND EXECUTE PROGRAM
N
ANY MORE EXAMPLES ?
N
EXIT


ENTER
DISPLAY TOP ELEMENT OF STACK
OPERATOR
EXIT
UNARY
STACK OPERAND ?
Y
PICK UP TOP ELEMENT OF STACK AS OPERAND
N
PUSH ACCUMULATOR ONTO STACK
EXECUTE USER'S INSTRUCTIONS AND DISPLAY RESULT
SET ACCUMULATOR TO ZERO
DROP TOP POSITION OF STACK
APPEND OPERAND, OPERATOR AND RESULT TO THE PROGRAM TREE
ROTATE STACK ONE POSITION
SET UP NEW POINTERS
EXIT

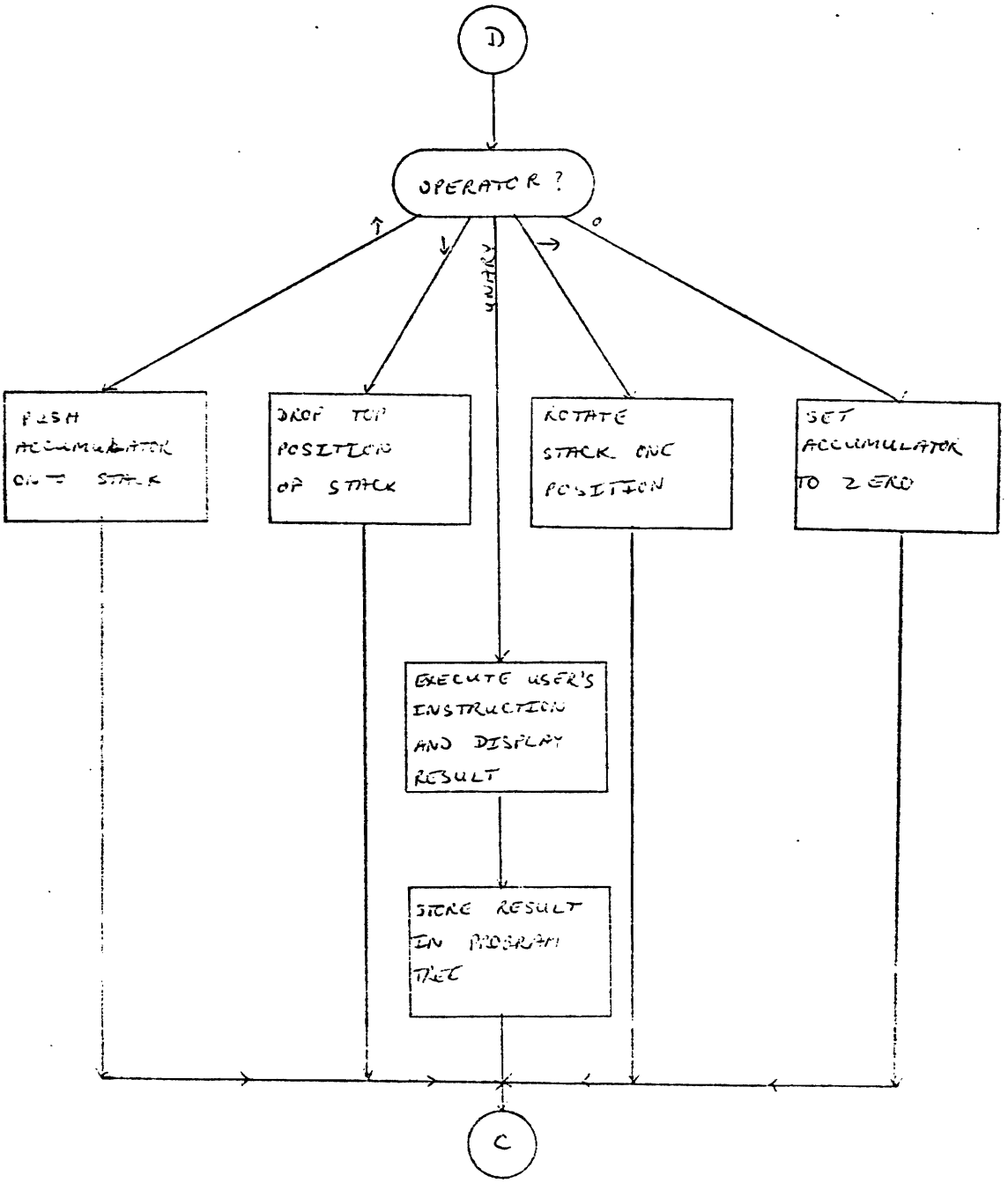
NEXT

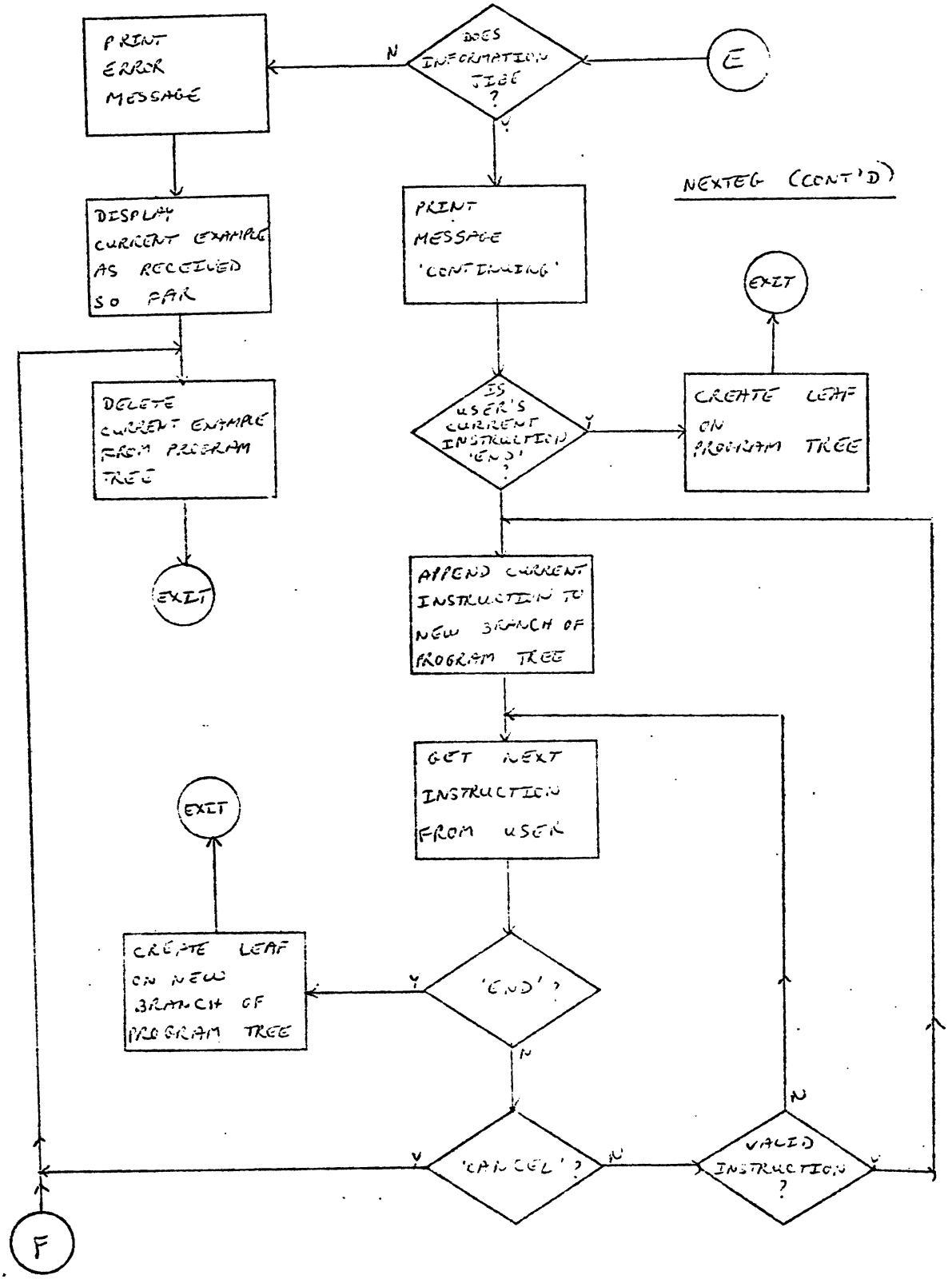

NEXTEG
ENTER
INITIALIZE STACK AND ACCUMULATOR AND PRINT MESSAGE
ACCEPT INPUT FROM USER OR USE PREMPT OUTPUT
'END' ?
Y
LEAF IN PROGRAM TREE HERE ?
Y
EXIT
N
B
N
'CANCEL' ?
Y
F
N
VALID STRING ?
N
Y
DISPLAY REQUEST
Y
DISPLAY TOP ELEMENT OF STACK
N
DID PREVIOUS EXAMPLES END HERE ?
Y
B
N
A
C

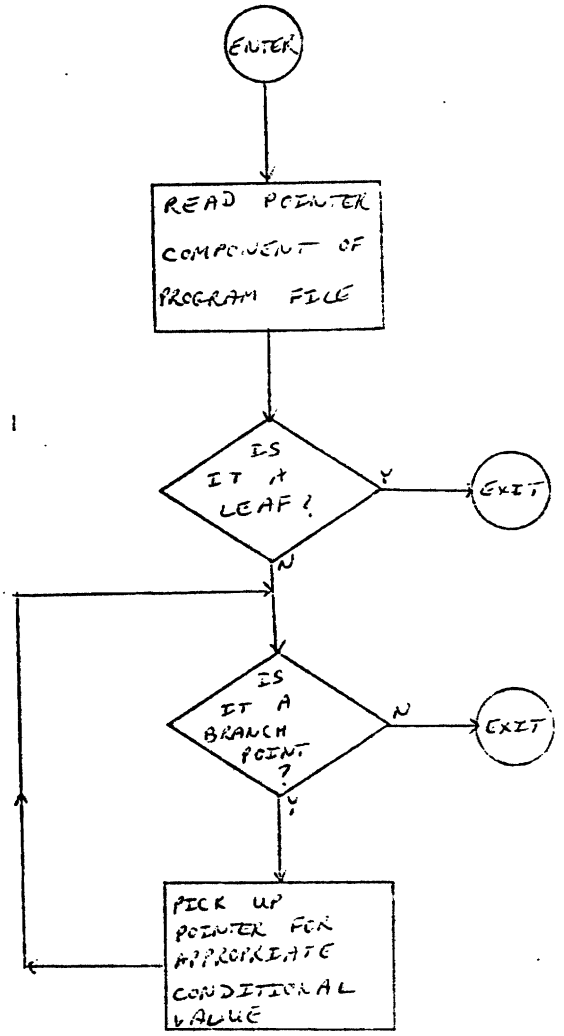
NEXTEG (CONT'D)



NEXTEG (CONT'D)

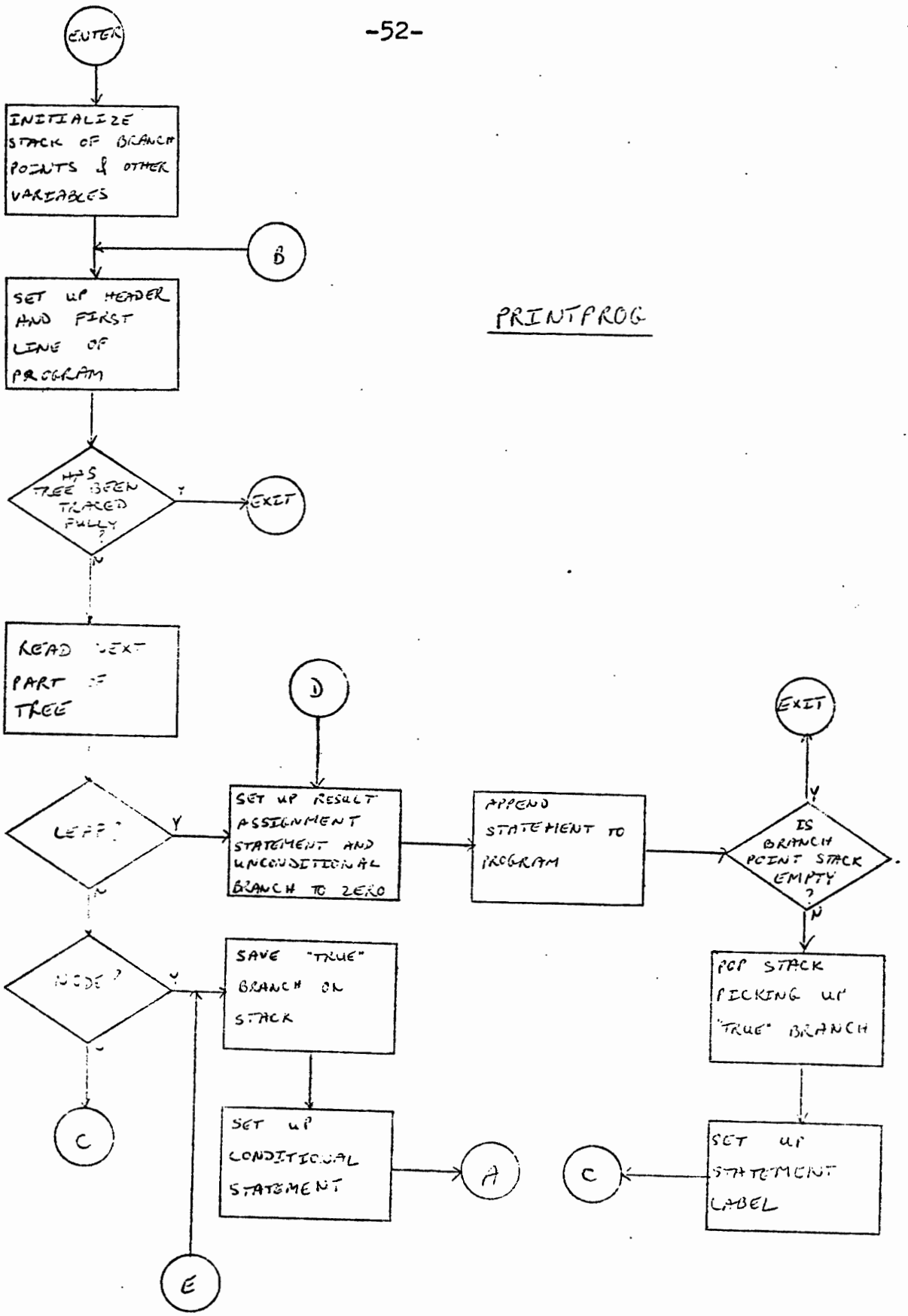


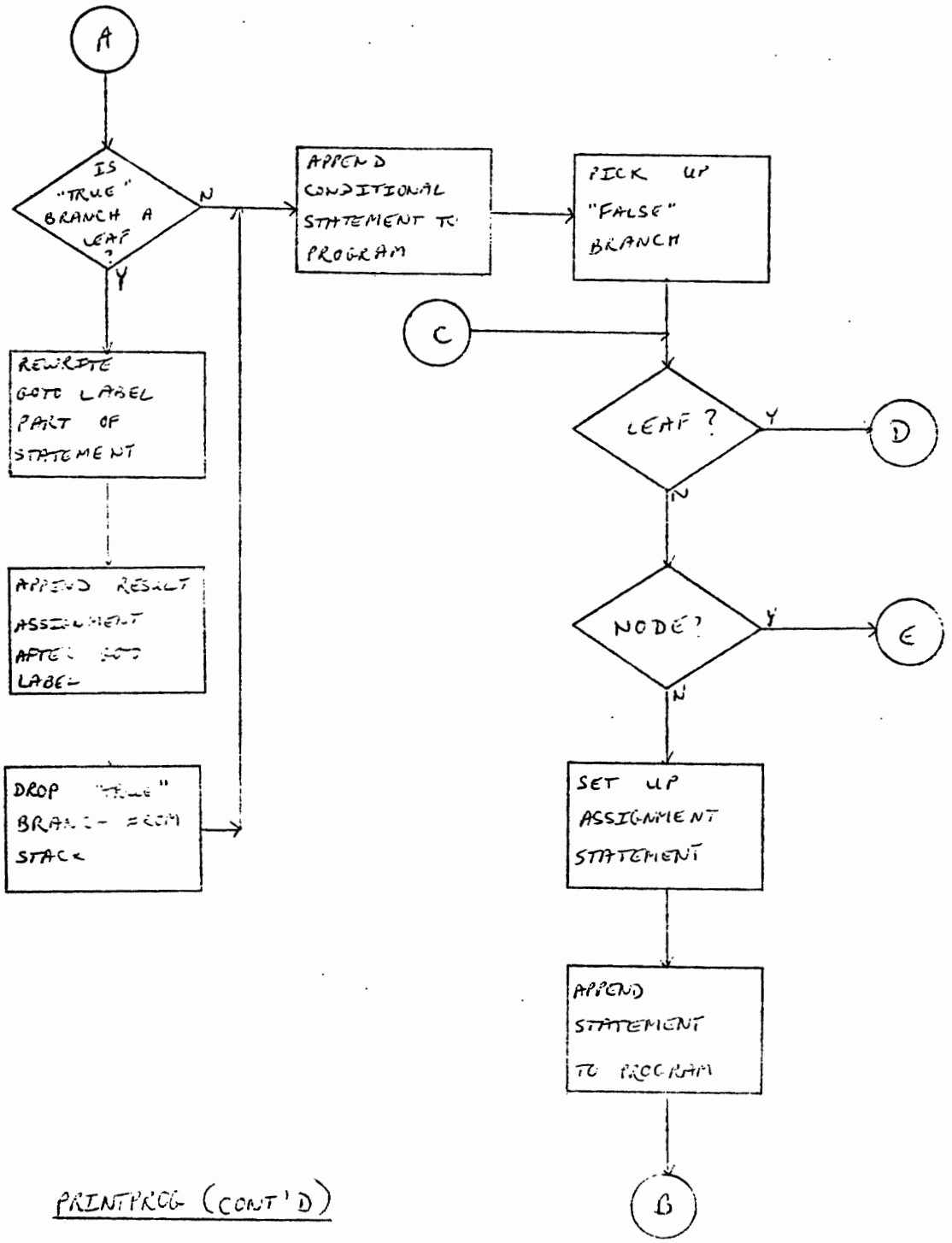

PRINT ERROR MESSAGE
N
DOES INFORMATION JIBE ?
E
Y
NEXTEG (CONT'D)
DISPLAY CURRENT EXAMPLE AS RECEIVED SO FAR
PRINT MESSAGE 'CONTINUING'
EXIT
DELETE CURRENT ENTAMPLE FROM PROGRAM TREE
IS USER'S CURRENT INSTRUCTION 'END' ?
Y
CREATE LEAF ON PROGRAM TREE
EXIT
APPEND CURRENT INSTRUCTION TO NEW BRANCH OF PROGRAM TREE
GET NEXT INSTRUCTION FROM USER
EXIT
CREATE LEAF ON NEW BRANCH OF PROGRAM TREE
Y
'END' ?
N
N
Y
'CANCEL' ?
N
VALID INSTRUCTION ?
Y
F


ENTER
READ POINTER COMPONENT OF PROGRAM FILE
IS IT A LEAF?
Y
EXIT
N
IS IT A BRANCH POINT?
N
EXIT
Y
PICK UP POINTER FOR APPROPRIATE CONDITIONAL VALUE

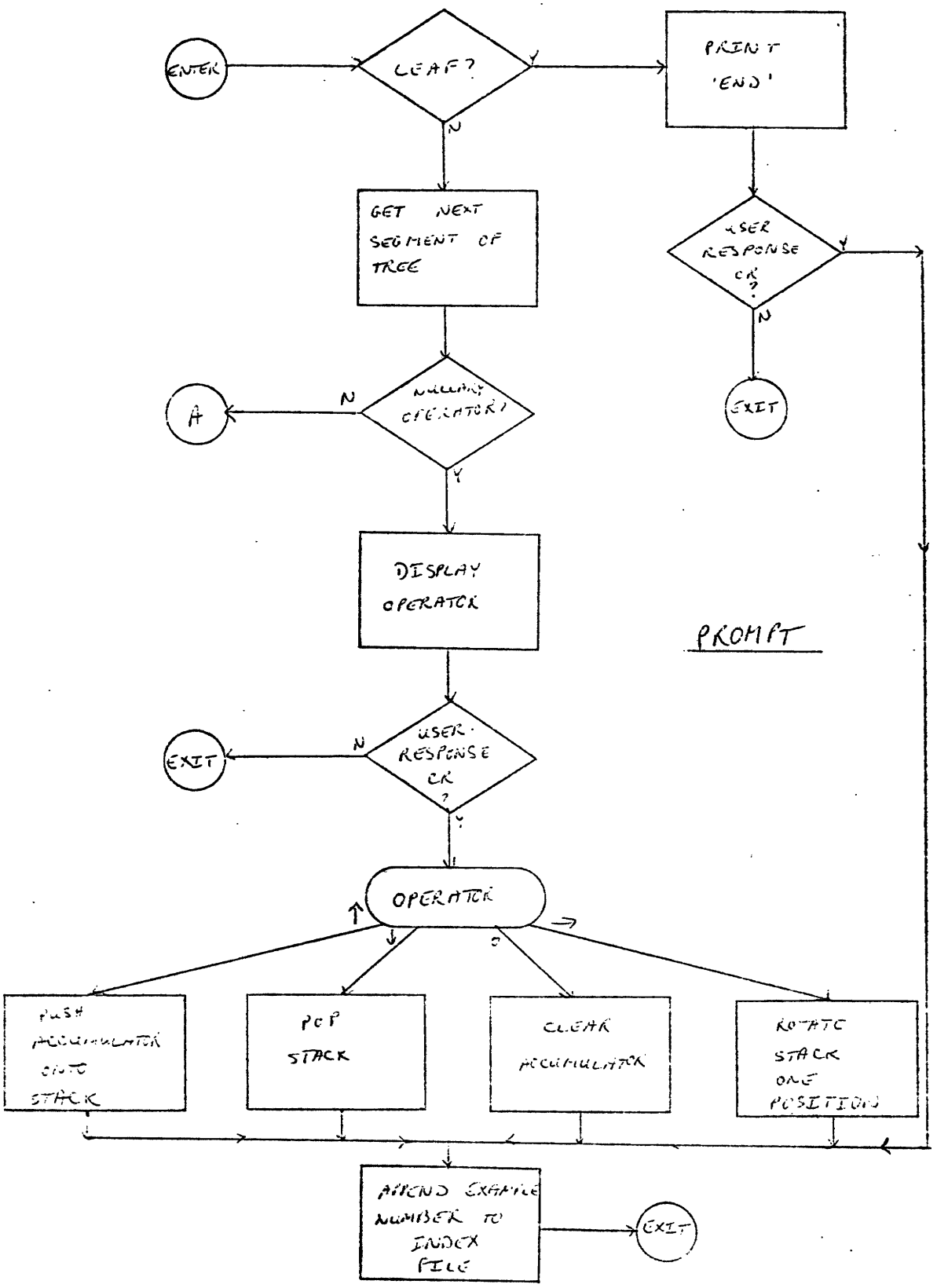
POINTER

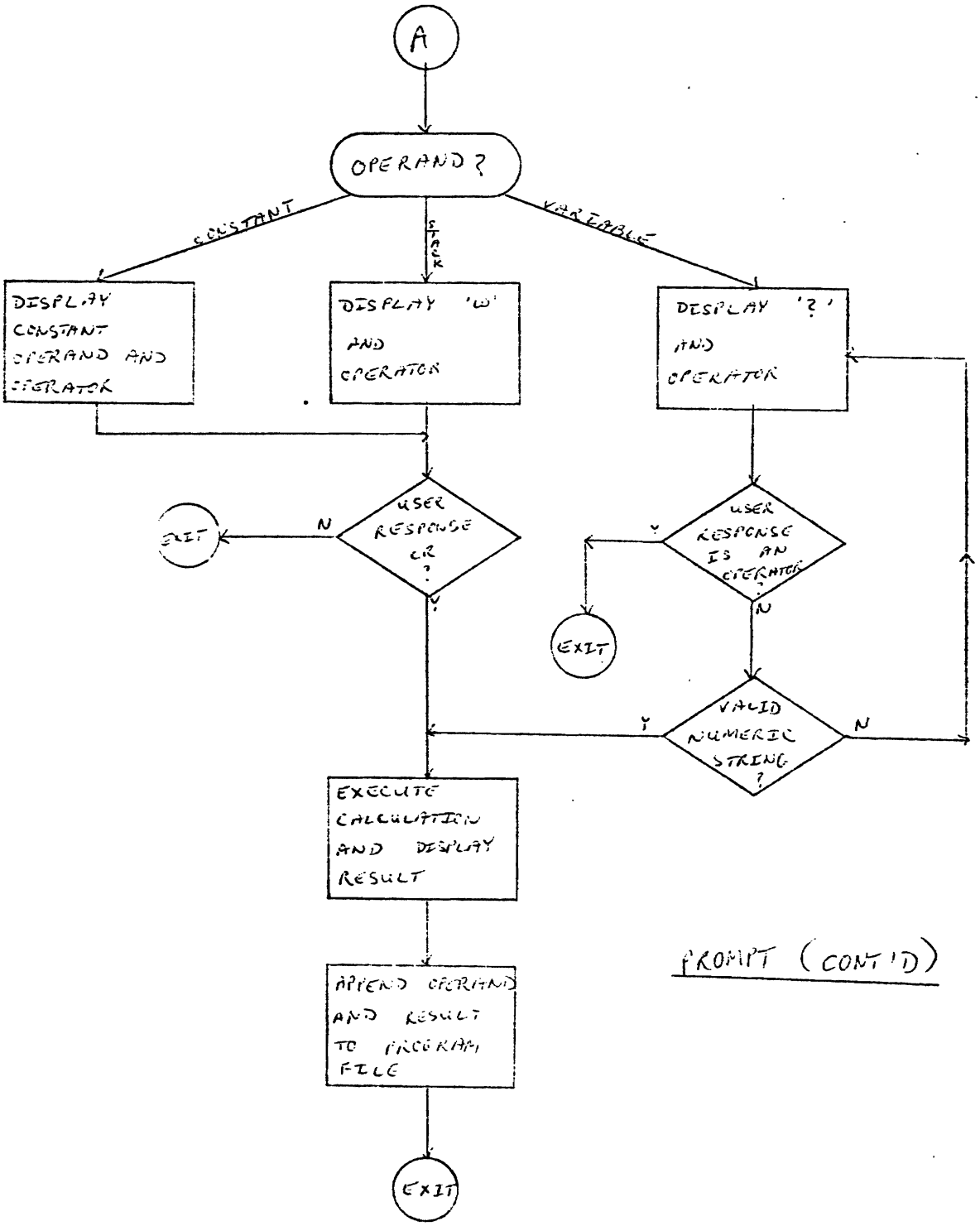
PRINTPROG


ENTER
INITIALIZE STACK OF BRANCH POINTS & OTHER VARIABLES
B
SET UP HEADER AND FIRST LINE OF PROGRAM
HAS TREE BEEN TRACED FULLY ?
Y
EXIT
N
READ NEXT PART OF TREE
D
LEFT ?
Y
SET UP RESULT ASSIGNMENT STATEMENT AND UNCONDITIONAL BRANCH TO ZERO
APPEND STATEMENT TO PROGRAM
EXIT
Y
IS BRANCH POINT STACK EMPTY ?
N
N
NODE ?
Y
SAVE "TRUE" BRANCH ON STACK
POP STACK PICKING UP "TRUE" BRANCH
C
SET UP CONDITIONAL STATEMENT
A
C
SET UP STATEMENT LABEL
E



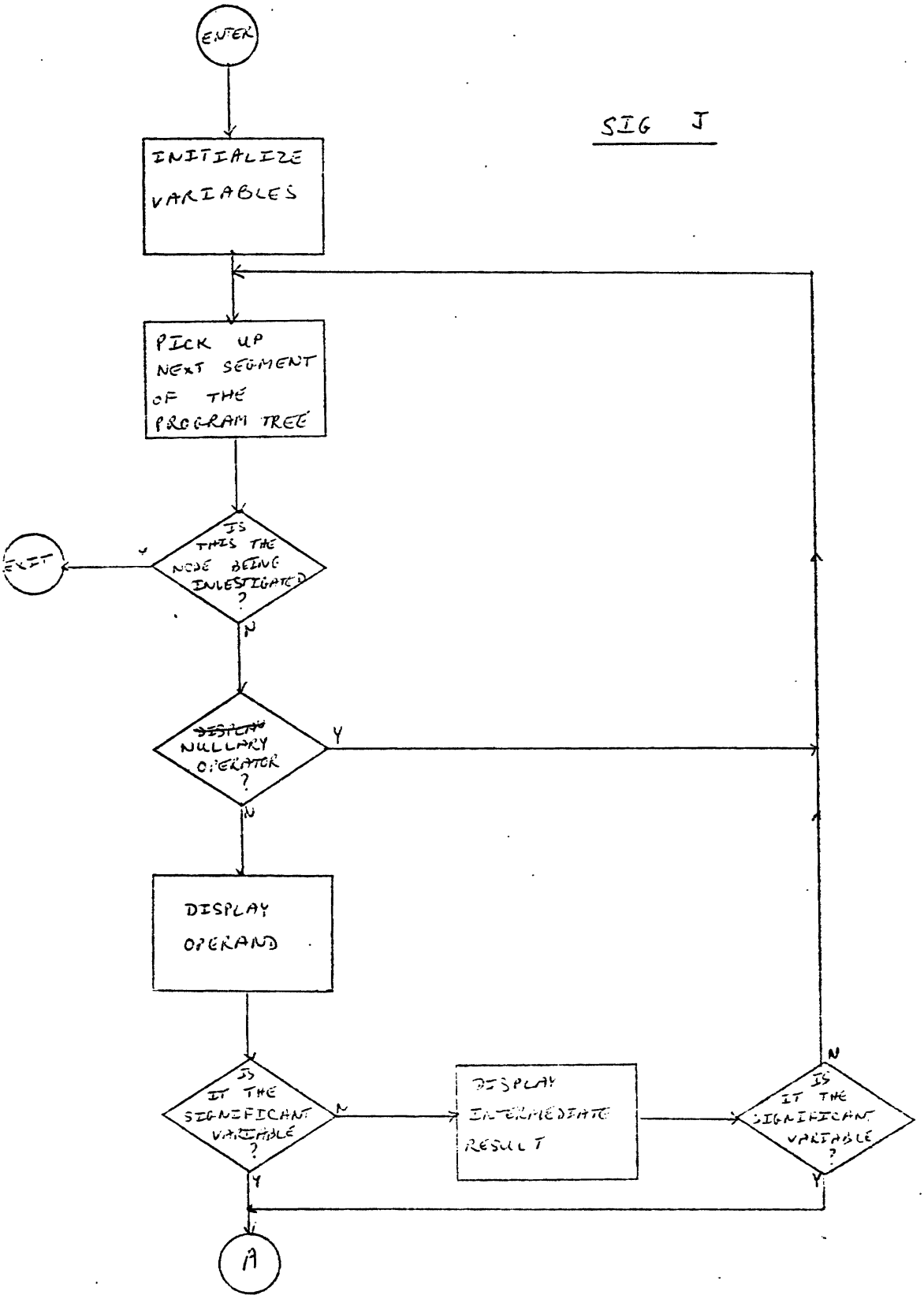
PRINTPROG (CONT'D)


ENTER
LEAF?
Y
PRINT 'END'
N
GET NEXT SEGMENT OF TREE
USER RESPONSE OK?
Y
N
EXIT
A
N
NUMERIC OPERATOR?
Y
DISPLAY OPERATOR
PROMPT
EXIT
N
USER RESPONSE OK?
Y
OPERATOR
PUSH ACCUMULATOR ONTO STACK
POP STACK
CLEAR ACCUMULATOR
ROTATE STACK ONE POSITION
APPEND EXPRESSION NUMBER TO INDEX FILE
EXIT

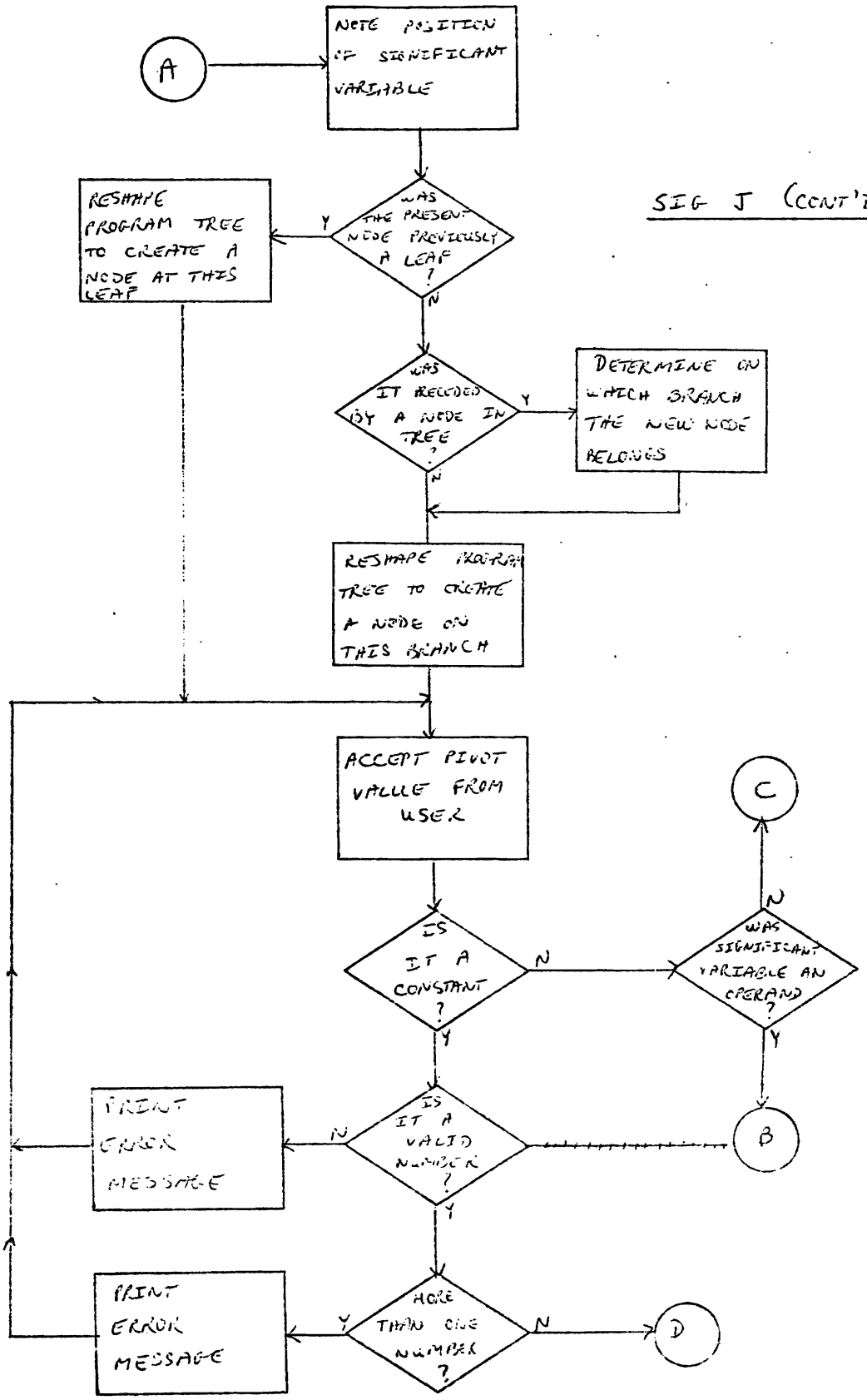


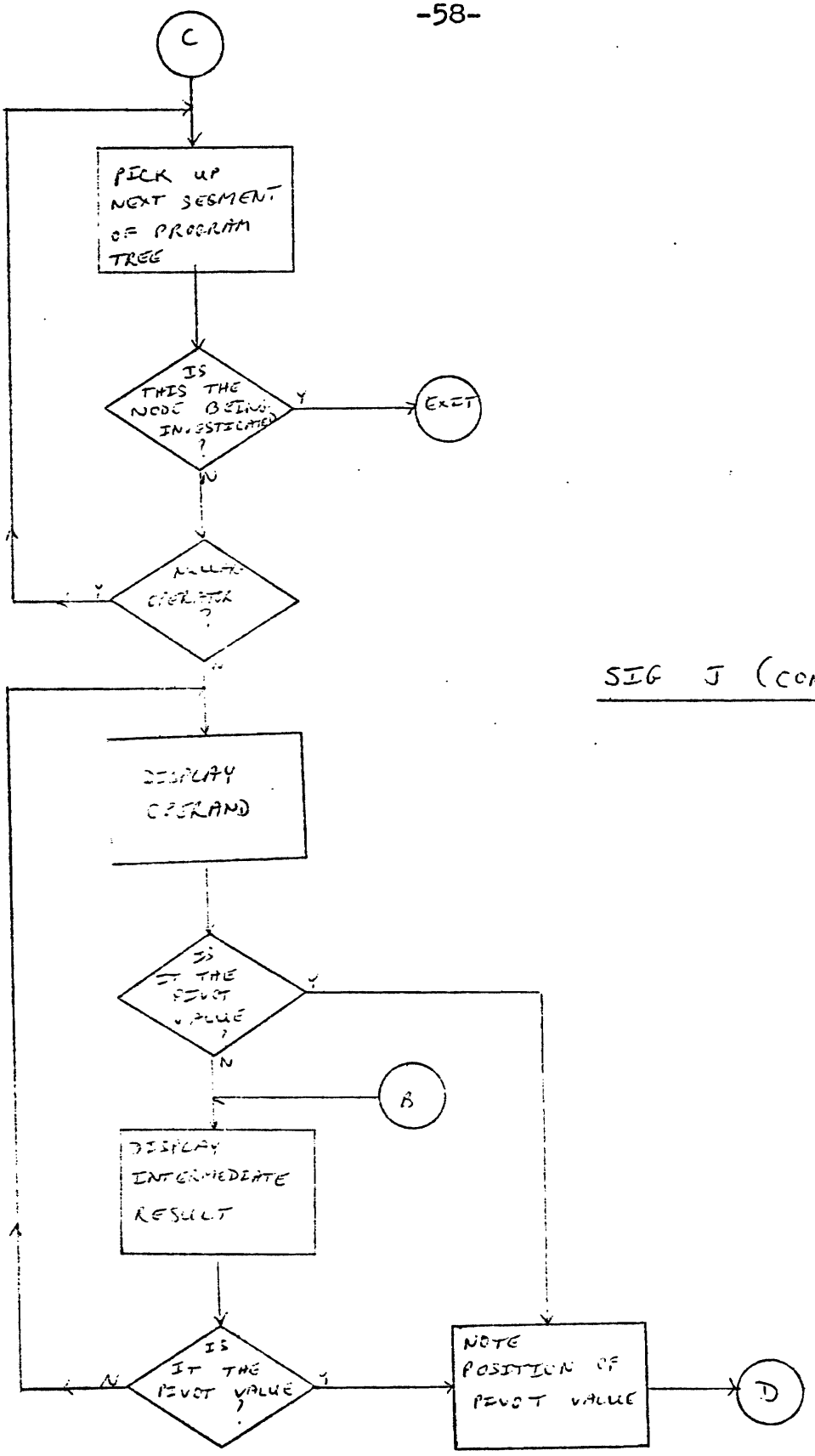
PROMPT (CONT'D)

SIG J

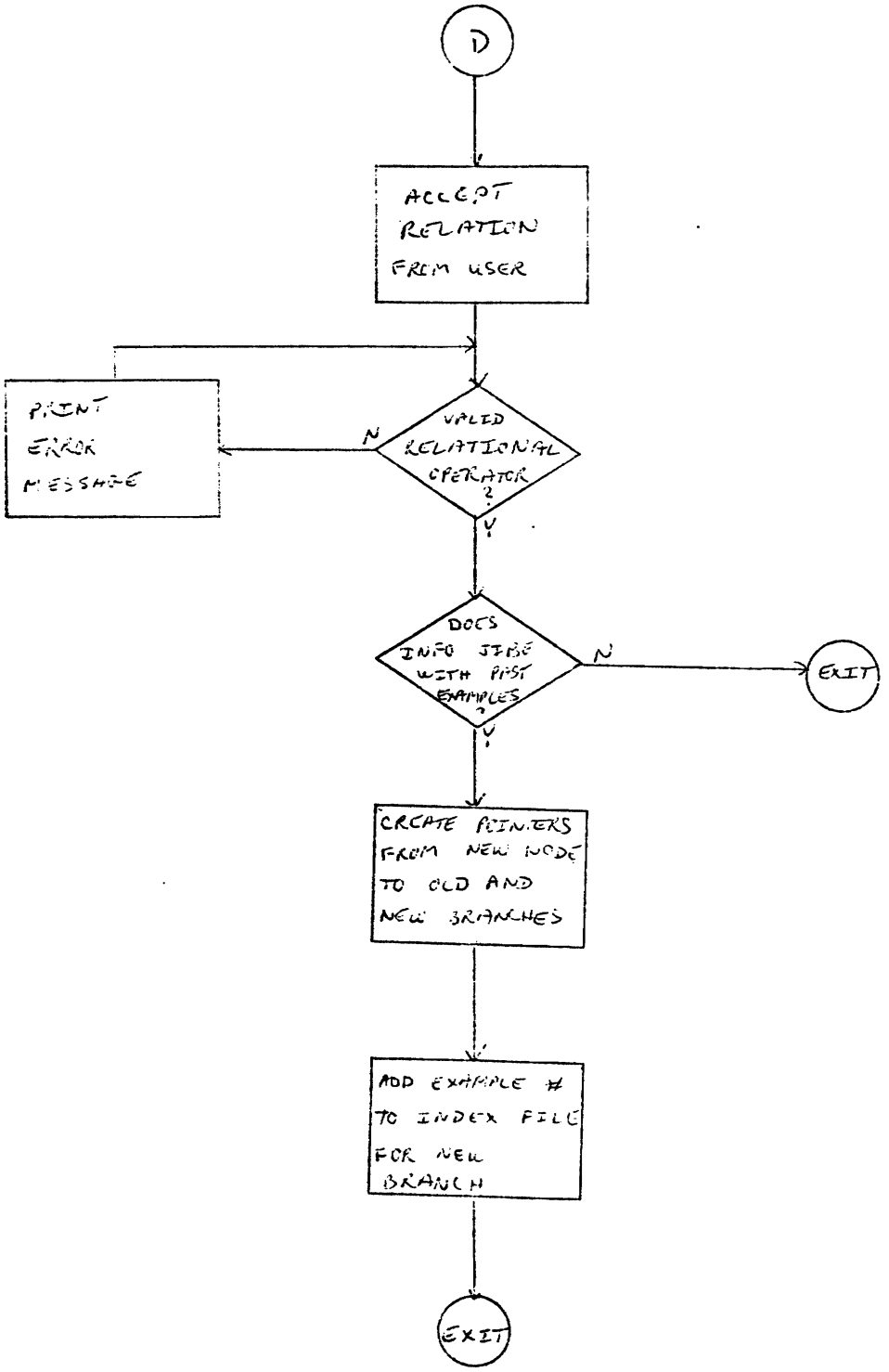

ENTER
INITIALIZE VARIABLES
PICK UP NEXT SEGMENT OF THE PROGRAM TREE
IS THIS THE NODE BEING INVESTIGATED ?
Y
EXIT
N
NULLARY OPERATOR ?
Y
N
DISPLAY OPERAND
IS IT THE SIGNIFICANT VARIABLE ?
N
DISPLAY INTERMEDIATE RESULT
IS IT THE SIGNIFICANT VARIABLE ?
N
Y
Y
A

SIG J (CONT'D)


A
NOTE POSITION OF SIGNIFICANT VARIABLE
WAS THE PRESENT NODE PREVIOUSLY A LEAF ?
Y
RESHAPE PROGRAM TREE TO CREATE A NODE AT THIS LEAF
N
WAS IT RECORDED BY A NODE IN TREE ?
Y
DETERMINE ON WHICH BRANCH THE NEW NODE BELONGS
N
RESHAPE PROGRAM TREE TO CREATE A NODE ON THIS BRANCH
ACCEPT PIVOT VALUE FROM USER
IS IT A CONSTANT ?
N
WAS SIGNIFICANT VARIABLE AN OPERAND ?
N
C
Y
Y
IS IT A VALID NUMBER ?
N
PRINT ERROR MESSAGE
B
Y
HOW MANY DIGITS IN NUMBER ?
Y
PRINT ERROR MESSAGE
N
D


C
PICK UP NEXT SEGMENT OF PROGRAM TREE
IS THIS THE NODE BEING INVESTIGATED ?
Y
EXIT
N
Y
DISPLAY OPERAND
IS IT THE PIVOT VALUE ?
Y
N
B
DISPLAY INTERMEDIATE RESULT
IS IT THE PIVOT VALUE ?
N
Y
NOTE POSITION OF PIVOT VALUE
D

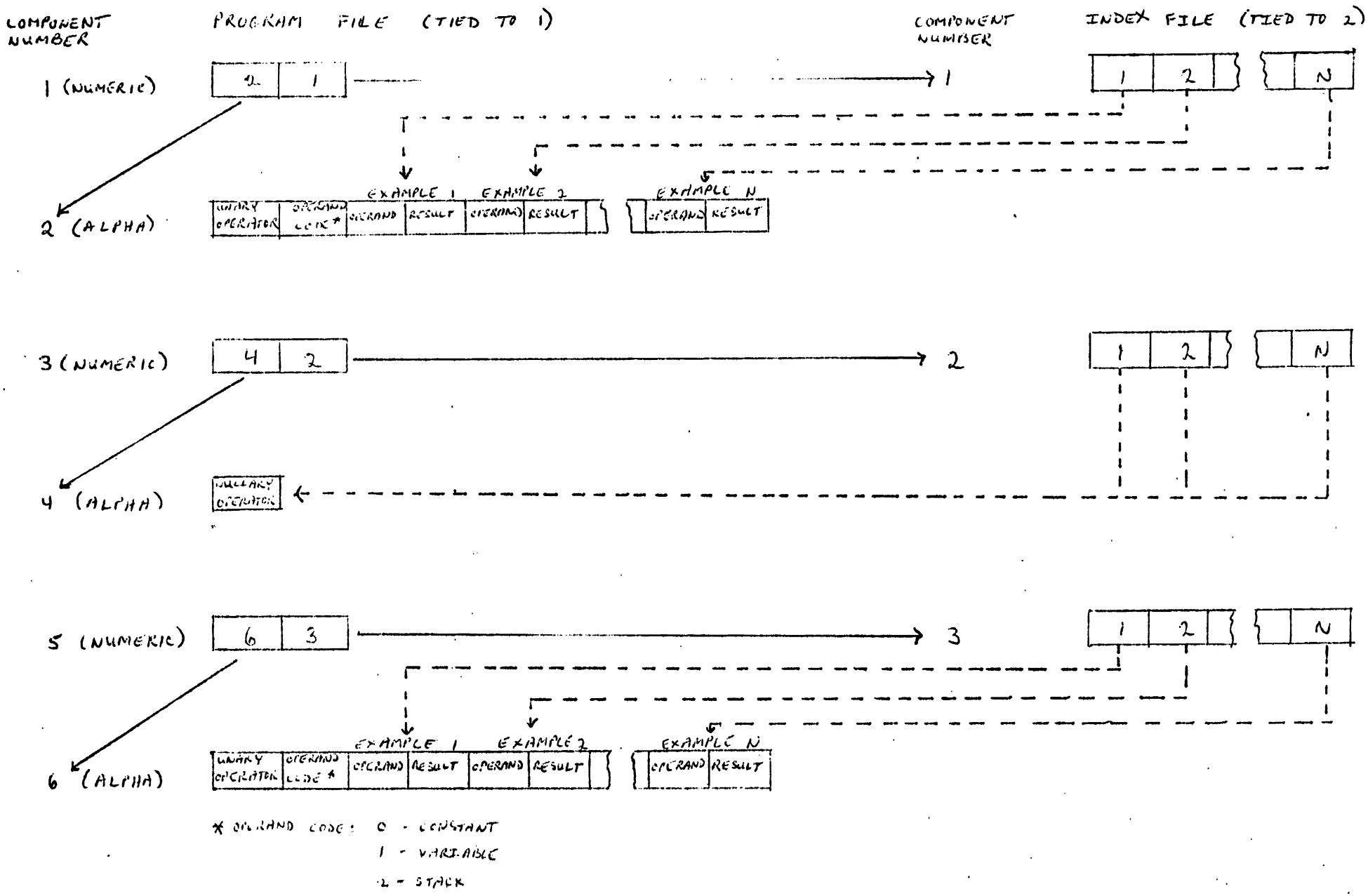
SIG J (CONT'D)


D
ACCEPT RELATION FROM USER
PRINT ERROR MESSAGE
N
VALID RELATIONAL OPERATOR ?
Y
DOES INFO JIBE WITH PAST EXAMPLES ?
N
EXIT
Y
CREATE POINTERS FROM NEW NODE TO OLD AND NEW BRANCHES
ADD EXAMPLE # TO INDEX FILE FOR NEW BRANCH
EXIT

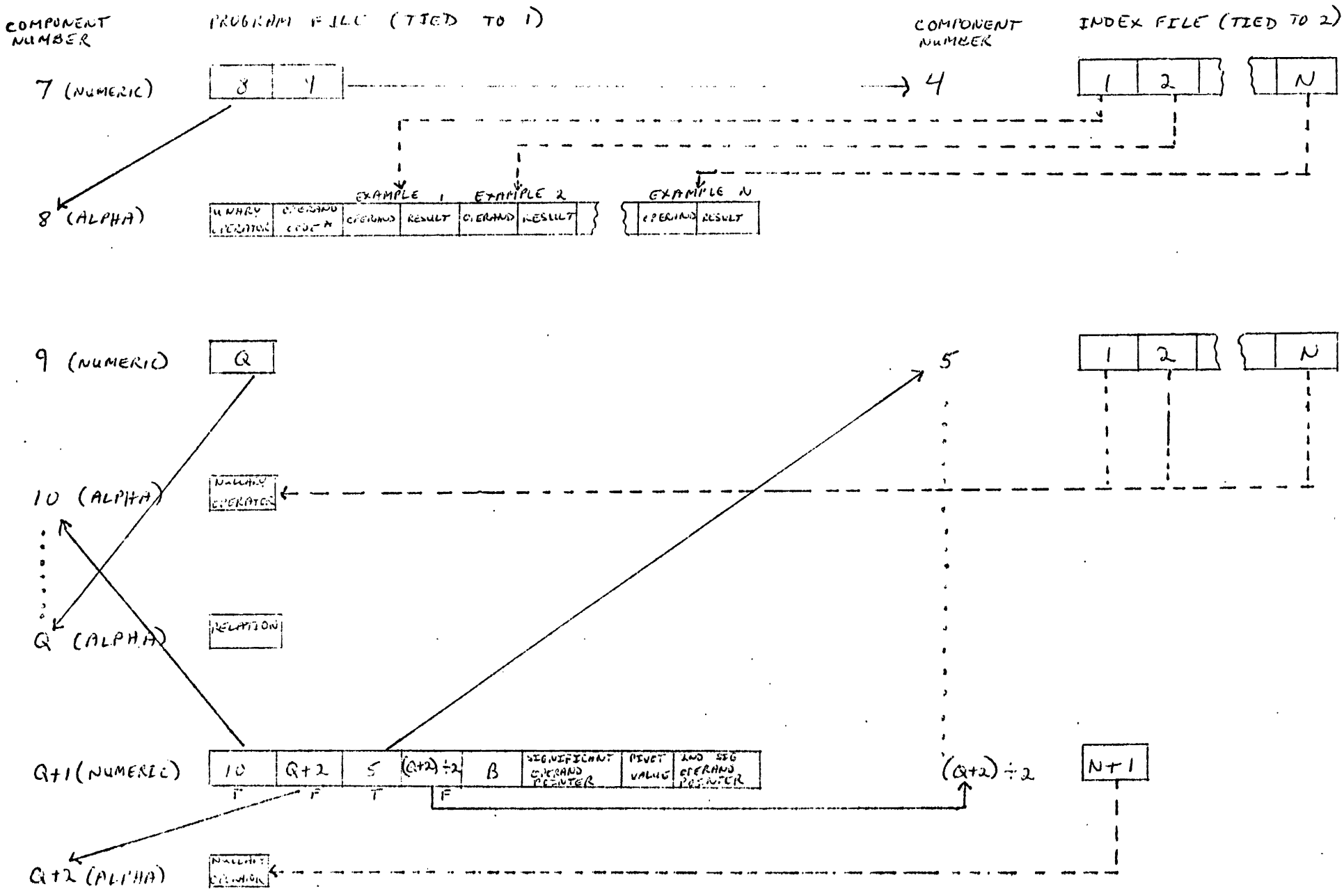
APPENDIX III

FILE LAYOUTS

LAYOUT OF PROGRAM AND INDEX FILES AND THEIR RELATIONSHIP



LAYOUT OF PROGRAM AND INDEX FILES AND THEIR RELATIONSHIP, CONTINUED



APPENDIX IV

FILE DESCRIPTIONS

The file PROG is tied to 1 and is composed of four types of components:

I Pointers

II Operation Traces

III Relations

IV Branch Pointers

TYPE I: Pointers (numeric)

The first element of the component is a pointer whose value is the number of the component in file PROG which contains the next segment of the trace. It is generally the next component in the file. A value of minus one for this pointer indicates that the end of the trace has been reached. The second element of the component, if it exists, is a pointer whose value is the number of the component in file INDEX which contains significant information relating to the next segment of the trace (see the description of the INDEX file). If the second element is missing, the first element indicates a branch point at the next point in the trace.

TYPE II: Operation Traces (alphabetic)

The first element is an operator. If the operator is nullary, there are no more elements in this component.
The second element, if it exists, indicates the type of operand used with the operator. A value of zero indicates a

constant operand; a value of one indicates a variable operand; a value of two indicates a stack operand.
The remainder of the component is composed of pairs of strings of indefinite length separated by commas. The first string of each pair is the literal representation of an operand used with this operator; the second string is the literal representation of the result obtained from the operator and the aforementioned operand. There are as many operand-result pairs as there were examples which passed through this trace.

TYPE III: Relations (alphabetic)
The only element of this component is the relational operator involved at a branch point.

TYPE IV: Branch Pointers (numeric)
The first four elements of this component correspond to type I components discussed earlier. The first pair of elements are the pointers of the true and false paths of a trace. That is, if the value of the conditional statement is true, the trace continues at the component number whose value is contained in the first element of the pair; if the value of the conditional statement is false, the trace continues at the component number whose value is contained in the second element of the pair. A value of minus one

indicates that the end of a trace has been reached. The fifth element of this component points to the closest preceding nonbranch point in the trace.

The next two elements of this component correspond to the second element of type I data when true and false values are respectively obtained for the conditional statement. A value of zero is equivalent to the non-existence of this element.

The sixth element of the component points to the first operand which participates in the conditional. If the value of the pointer is positive, the operand is indicated; if it is negative, the result is indicated.

The seventh and eighth elements of the component point to the second operand participating in the conditional. If the eighth element is a zero, then the value of the second operand is given in the seventh element; if the eighth element is non-zero, it points to an operand within PROG. As in element six, if it is positive, the pointer indicates an operand; if it is negative, it indicates a result.

The file INDEX is tied to 2 and is composed of only one type of component.

Each component consists of a list of numbers. These numbers correspond to the example numbers represented in the operation trace components of PROG.

APPENDIX V

INPUT TO THE MODEL

OPERANDS

Type	Description
constant	any type acceptable to APL
⍵	indicates that the top element of the stack is to be used
⍺	indicates that another program variable is to be used
END	indicates the example is complete
CANCEL	indicates the user wishes to erase the current example

OPERATORS

Nullary:

↑	Push the accumulator onto the stack
↓	Pop the stack--i.e., allow the top element of the stack to disappear
→	Rotate the stack by one position; the top element goes to the bottom
c	Clear the accumulator
□	Display the top element of the stack

Unary:

+	Add the operand into the accumulator and store the result in the accumulator
-	Subtract the operand from the accumulator and store the result in the accumulator
x	Multiply the operand by the accumulator and store the result in the accumulator
÷	Divide the accumulator by the operand and store the result in the accumulator
*	Raise the accumulator to the power of the operand and store the result in the accumulator

Binary:

$<$	Test whether the significant variable is less than the pivot value
$\leq$	Test whether the significant variable is less than or equal to the pivot value
=	Test whether the significant variable is equal to the pivot value
$>$	Test whether the significant variable is greater than the pivot value
$\geq$	Test whether the significant variable is greater than or equal to the pivot value
$\neq$	Test whether the significant variable is unequal to the pivot value

NOTE: If the model prompts the user by printing a nullary operator or an operand and unary operator, the user may respond with a carriage return which is equivalent to accepting the prompt as the next step in the example, or may override the prompt by entering some other valid computational step.

APPENDIX VI

PROGRAM MESSAGES AND THEIR MEANINGS

BEGIN ENTERING FIRST EXAMPLE

Initialization of the files is complete. The user may begin entering his example.

TOO MANY OPERANDS. RE-ENTER

A computational step has too many operands in it. This may result from a blank or comma embedded in the operand.

WRONG NUMBER OF OPERATORS. RE-ENTER

There must be exactly one operator.

NO OPERAND. RE-ENTER

A unary operator has been entered without an operand.

INVALID NUMERIC CHARACTER. RE-ENTER

The operand is an invalid APL constant.

OPERATOR NOT LAST SYMBOL. RE-ENTER

The computational step is syntactically incorrect.

ANY MORE EXAMPLES(Y/N)?

Asks if the user wishes to enter any more examples. Entering the letter N indicates no; any other response will cause the model to expect another example.

ENTERING EXAMPLE xx

Indicates that the model is ready to accept the next example and tells the user which example is about to be accepted.

?

Indicates the operator which the model expects next takes a variable operand. Waits for an operand or an override.

CONSTANT OR VARIABLE (C/V)?

Asks whether the operand which was just entered is a constant or a variable. Entering the letter C indicates a response of constant. Any other response will be interpreted as a variable.

CONTINUING

The user may continue entering his example from the point at which he left off.

DOES NOT TALLY WITH PAST EXAMPLES

Information supplied by the user is not consistent with past examples. The current example will be displayed then deleted.

(Y/N)?

The model is displaying the value of a variable operand used

earlier in the example and is asking whether it is the significant variable. The user may respond with the letter Y for yes or N for no.

ENTER PIVOT VALUE

The user may enter either a valid APL constant or the letter alpha to indicate that the pivot value is another variable operand.

ENTER RELATION

The user is requested to enter the suitable binary operator.

INVALID NUMERIC CHARACTER

Re-enter the pivot value.

ONE VALUE ONLY

Re-enter the pivot value.

INVALID OPERATOR

Re-enter the relational operator.

DO YOU WANT TO TRY IT OUT?(Y/N)

The program has been created. The user may respond with a Y for yes or an N for no.

EXAMPLE xx

Precedes the example being displayed.

INVALID EXAMPLE NUMBER

A request was made to display an example which doesn't exist.

APPENDIX VII

SAMPLE RUNS

Hero's Algorithm for the Area of a Triangle

MAIN $\Delta = \sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{1}{2}(a+b+c)$

a, b, c are the lengths of the sides of the triangle

BEGIN ENTERING FIRST EXAMPLE.

```
      3+
3
      ↑
      o
      5+
5                 } ENTER a,b,c IN
      ↑             STACK
      o
      7+
7
      ↑
      →
      □
5
      ω+
12
      →
      −           } CALCULATE S AND
3                   SAVE IN STACK
      ω÷
15
      2÷
7.5
      ↑
      →
      □
3
      ω÷>
```

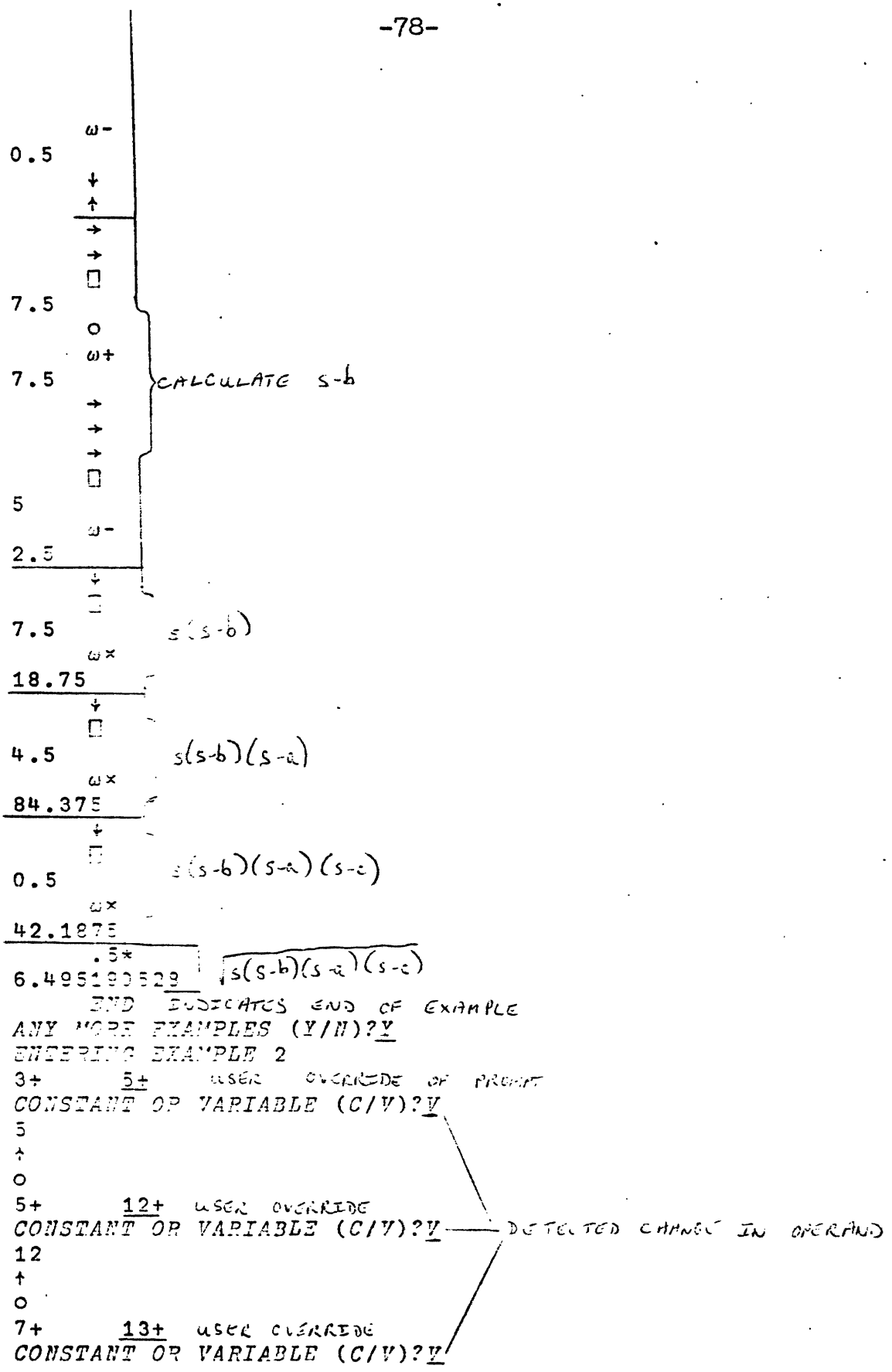
WRONG NUMBER OF OPERATORS. RE-ENTER.

```
      ω−          } CALCULATE S-a AND
4.5                 SAVE IN STACK
      ÷
      ↑
      →
      →
      →
      □
7.5
      o
      ω+          } CALCULATE S-c AND
7.5                 SAVE IN STACK
      →
      →
      □
7
```

NOTE

USER'S INPUT IS INDENTED OR UNDERLINED

NUMBERS IN LEFT MARGIN SHOW VALUE OF ACCUMULATOR AFTER A UNARY OPERATION HAS BEEN PERFORMED, OR SHOW VALUE OF TOP ELEMENT OF STACK AFTER A □ REQUEST.



END INDICATES END OF EXAMPLE

```
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 2
3+      5+      USER OVERRIDE OF PROMPT
CONSTANT OR VARIABLE (C/V)?V
5
↑
O
5+      12+     USER OVERRIDE
CONSTANT OR VARIABLE (C/V)?V
12
↑
O
7+      13+     USER OVERRIDE
CONSTANT OR VARIABLE (C/V)?V
```

DETECTED CHANGE IN OPERAND

```
13
↑
→
ω+
25
→
ω+
30
2÷
15
↑
→
ω-
10
↓
↑
→
→
→
O
ω+
15
→
→
ω-
2
↓
↑
→
→
O
ω+
15
→
→
→
ω-
3
↓
ω×
45
↓
ω×
450
↓
ω×
900
.5*
30
END
```

THIS ENTIRE PAGE WAS GENERATED BY THE PROMPT MECHANISM.

```
ANY MORE EXAMPLES (Y/N)?N
DO YOU WANT TO TRY IT OUT?(Y/N)Y
R←PROG;T;S
S←⍳T←0
T←T+⍞                    THIS IS THE APL PROGRAM
S←T,S                    DEVELOPED BY THE SIMULATION
T←0                      BASED ON THE TWO EXAMPLES GIVEN
T←T+⍞                    BY THE USER.
S←T,S                    INITIAL AND FINAL '∇'S ARE
T←0                      ABSENT IN ORDER TO MEET THE
T←T+⍞                    REQUIREMENTS OF 3∆FD, A SYSTEM
S←T,S                    PROGRAM.
S←1↓S
T←T+1↑S
S←1↓S
T←T+1↑S
T←T÷2
S←T,S
S←1↓S
T←T-1↑S
S←1↓S
S←T,S
S←1↓S
S←1↓S
S←1↓S
T←0
T←T+1↑S
S←1↓S
S←1↓S
T←T-1↑S
S←1↓S
S←T,S
S←1↓S
S←1↓S
T←0
T←T+1↑S
S←1↓S
S←1↓S
S←1↓S
T←T-1↑S
S←1↓S
T←T×1↑S
S←1↓S
T←T×1↑S
S←1↓S
T←T×1↑S
T←T*.5
→0,R←T
⍞:              REQUESTS INPUT
      3          2
```

□:

4 b

□:

5 c

6

DO YOU WANT TO TRY IT OUT?(Y/N)N

ANY MORE EXAMPLES (Y/N)?N

MORE EXAMPLES MAY BE ENTERED AFTER TESTING. IF ADDITIONS ARE MADE, THE SIMULATION RECYCLES FOR MORE TESTING.

Calculation of the Final Temperature of a Mixture of Two Known Quantities of Ice and Water. Temperature and Mass of Container are Ignored.

```
      MAIN
BEGIN ENTERING FIRST EXAMPLE.
      10+          10 GMS. OF ICE AT 100° K.
10                 15 GMS. OF ICE AT 200° K.
      ↑
      100×
1000
      ↑
      O
      15+
15
      ↑          CALCULATE AND SAVE TOTAL
      200×       NUMBER OF CALORIES PRESENT
3000
      →          C=(10×100)+(15×200)
      □+
4000
      ↓
      ↑
      →
      O
      □+
10                CALCULATE TOTAL MASS OF ICE
      ↓           M=10+15
      □+
25
      →
      ↑
      →
      O
      □+          CALCULATE FINAL TEMPERATURE OF ICE
4000              T=4000÷25
      →
      □÷
160
      END
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 2
10+      15+
CONSTANT OR VARIABLE (C/V)?V
15
↑
100×     75×
```

```
CONSTANT OR VARIABLE (C/V)?V
1125
↑
o
15+      10+
CONSTANT OR VARIABLE (C/V)?V
10
↑
200×      350×
CONSTANT OR VARIABLE (C/V)?V
3500
→
ω+
4625
↓
↑
→      o
15   (Y/N)?N
1.5000000000E1      (Y/N)?N
75   (Y/N)?N
1.1250000000E3      (Y/N)?N
10   (Y/N)?N
1.0000000000E1      (Y/N)?N
350  (Y/N)?Y
ENTER PIVOT VALUE.273
ENTER RELATION.>
CONTINUING
      →
      →
      ω+
10
      80×
800
      →
      ω+
5425
      ↓
      ↑
      →
      o
      ω+
15
      ↓
      ω+
25
      ↓
      ↑
      →
      o
```

15 GMS. OF ICE AT 75° K
10 GMS OF WATER AT 350° K.

THE USER IS PRESENTED WITH ALL OPERANDS AND INTERMEDIATE RESULTS USED IN THE EXAMPLE TO THIS POINT. THE USER MUST INDICATE WHICH OF THE VALUES HAS CAUSED HIM TO CHANGE PROCEDURE. PRESENTATION ENDS WHEN THE USER INDICATES A PARTICULAR VALUE.

SINCE THE TEMPERATURE OF THE SECOND QUANTITY IS GREATER THAN 273° K, WE MUST INCLUDE THE NUMBER OF CALORIES USED FOR THE CONVERSION OF ICE TO WATER.

$$C = (15 \times 75) + (10 \times 350) + (10 \times 80)$$

```
      ω+
5425
      →
      ω÷
217
      END
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 3
?+10          10  GMS.  OF  ICE  AT  75° K
10            15  GMS.  OF  WATER  AT  350° K
↑
?×75
750
↑
O
?+15
15
↑
?×350
5250
→
ω+
6000
↓
↑
O
→
→
ω+
15
80×
1200
→
ω+
7200
↓
↑
→
O
ω÷
10
÷
ω+
25
↓
↑
→
O
ω+
```

```
7200
→
ω÷
288
END      0
10   (Y/N)?N
1.0000000000E1     (Y/N)?N
75   (Y/N)?N
7.5000000000E2     (Y/N)?N
15   (Y/N)?N
1.5000000000E1     (Y/N)?N
350  (Y/N)?N
5.2500000000E3     (Y/N)?N
7.5000000000E2     (Y/N)?N
6.0000000000E3     (Y/N)?N
1.5000000000E1     (Y/N)?N
1.5000000000E1     (Y/N)?N
80   (Y/N)?N
1.2000000000E3     (Y/N)?N
6.0000000000E3     (Y/N)?N
7.2000000000E3     (Y/N)?N
1.0000000000E1     (Y/N)?N
1.0000000000E1     (Y/N)?N
1.5000000000E1     (Y/N)?N
2.5000000000E1     (Y/N)?N
7.2000000000E3     (Y/N)?N
7.2000000000E3     (Y/N)?N
2.5000000000E1     (Y/N)?N
2.8800000000E2     (Y/N)?Y
ENTER ERROR VALUE.273
ENTER RELATION.≥
CONTINUING
      273÷
273
      END
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 4
?+10
10
↑
?×75
750
↑
0
?+30
30
↑
?×400
12000
→
```

SINCE THE FINAL TEMPERATURE IS GREATER THAN 273° K. AND LESS THAN 353° K, WE RESET IT TO 273° K. (SLUSH)

10 GMS OF ICE AT 75° K

30 GMS OF WATER AT 400° K.

```
ω+
12750
↓
↑
O
→
→
ω+
30
80×
2400
→
ω+
15150
↓
↑
→
O
ω+
10
↓
ω+
40
↓
↑
→
O
ω+
15150
→
ω÷
378.73
O      30-
10   (Y/N)?Y
1.000000000E1     (Y/N)?N
75   (Y/N)?Y
7.500000000E2     (Y/N)?N
30   (Y/N)?Y
3.000000000E1     (Y/N)?N
400   (Y/N)?Y
1.200000000E4     (Y/N)?N
7.500000000E2     (Y/N)?N
1.275000000E4     (Y/N)?N
3.000000000E1     (Y/N)?N
3.000000000E1     (Y/N)?N
80   (Y/N)?Y
2.400000000E3     (Y/N)?N
1.275000000E4     (Y/N)?N
```

SINCE THE FINAL TEMPERATURE IS GREATER THAN 353° K, WE REDUCE IT BY 80° TO ACCOUNT FOR THE CONVERSION OF ICE TO WATER.

```
1.5150000000E4     (Y/N)?N
1.0000000000E1     (Y/N)?N
1.0000000000E1     (Y/N)?N
3.0000000000E1     (Y/N)?N
4.0000000000E1     (Y/N)?N
1.5150000000E4     (Y/N)?N
1.5150000000E4     (Y/N)?N
4.0000000000E1     (Y/N)?N
3.7875000000E2     (Y/N)?Y
ENTER PIVOT VALUE.353
ENTER RELATION.>
CONTINUING
298.75
     END
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 5
?+15
15                 15 GMS. OF WATER AT 350° K
↑
?×350              = GMS OF ICE AT 75° K.
5250
↑
o
?+10
10
↑
?×75
750
→
ω+
6000
↓
↑
→      o                          THIS TIME THE FIRST
15  (Y/N)?N                       TEMPERATURE IS GREATER
1.5000000000E1     (Y/N)?N        THAN 273° K
350   (Y/N)?Y
ENTER PIVOT VALUE.273
ENTER RELATION.>
CONTINUING
      →
      ω+
15
      80×
1200
      →
      →
      ω+
7200
```

```
      ↓
      ↑
      →
      O
      ⍵+
15
      ↓
      ⍵+
25
      ↓
      ↑
      →
      O
      ⍵+
7200
      →
      ⍵÷
288
      O
      273+
273
      END
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 6
?+30
30
↑                30  GMS. OF WATER AT 400° K.
?×400
12000            10  G-MS. OF ICE AT 75° K.
↑
O
?+10
10
↑
?×75
750
→
⍵+
12750
↓
↑
O
→
⍵+
30
80×
2400
→
→
```

```
ω+
15150
↓
↑
→
○
ω+
30
↓
ω+
40
↓
↑
→
○
ω+
15150
→
ω÷
378.75
○      80-
30   (Y/N)?N
3.000000000E1     (Y/N)?N
400   (Y/N)?N
1.200000000E4     (Y/N)?N
10   (Y/N)?N
1.000000000E1     (Y/N)?N
75   (Y/N)?N
7.500000000E2     (Y/N)?N
1.200000000E4     (Y/N)?N
1.275000000E4     (Y/N)?N
3.000000000E1     (Y/N)?N
3.000000000E1     (Y/N)?N
80   (Y/N)?N
2.400000000E3     (Y/N)?N
1.275000000E4     (Y/N)?N
1.515000000E4     (Y/N)?N
3.000000000E1     (Y/N)?N
3.000000000E1     (Y/N)?N
1.000000000E1     (Y/N)?N
4.000000000E1     (Y/N)?N
1.515000000E4     (Y/N)?N
1.515000000E4     (Y/N)?N
4.000000000E1     (Y/N)?N
3.787500000E2     (Y/N)?Y
ENTER PIVOT VALUE.353
ENTER RELATION.>
CONTINUING
298.75
     END
```

SAME AS EXAMPLE 4

```
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 7
?+10
10
↑                10  GMS.  OF  WATER  AT  290° K.
?×290
2900             10  GMS.  OF  WATER  AT  300° K.
↑
o
?+10
10
↑
?×300
3000
→
ω÷
5900
↓
↑
o
→
→
ω+
10
80×
800
→
ω+
6700
↓
↑
→
o
ω+
10
↓      30×
10   (Y/N)?N                    BOTH  QUANTITIES  ARE  WATER
1.000000000E1     (Y/N)?N
290   (Y/N)?Y
ENTER PIVOT VALUE.273
ENTER RELATION.>
CONTINUING
300
      →
      →
      ω+
7500
      ↓
      ↑
```

```
      →
      O
      ω+
10
      ↓
      ω+
20
      ↓
      ↑
      →
      O
      ω+
7500
      →
      ω÷
375
      30-
295
      END
ANY MORE EXAMPLES (Y/N)?Y
ENTERING EXAMPLE 8
?+10
10
↑                 10 GMS. OF WATER AT 350° K.
?×350
3500              15 GMS. OF ICE AT 75° K.
↑
O
?+15
15
↑
?×75
1125
→
ω+
4625
↓
↑
O
→
ω+
10
30×
800
→
→
ω+
5425
↓
```

```
↑
→
o
ω+
10
↓
ω+
25
↓
↑
→
o
ω+
5425
→
ω÷
217
o      END
10   (Y/N)?N                    THE FINAL TEMPERATURE IS LESS
1.000000000E1      (Y/N)?N      THAN 273° K
350   (Y/N)?N
3.500000000E3      (Y/N)?N
15   (Y/N)?N
1.500000000E1      (Y/N)?N
75   (Y/N)?N
1.125000000E3      (Y/N)?N
3.500000000E3      (Y/N)?N
4.625000000E3      (Y/N)?N
1.000000000E1      (Y/N)?N
1.000000000E1      (Y/N)?N
80   (Y/N)?N
8.000000000E2      (Y/N)?N
4.625000000E3      (Y/N)?N
5.425000000E3      (Y/N)?N
1.000000000E1      (Y/N)?N
1.000000000E1      (Y/N)?N
1.500000000E1      (Y/N)?N
2.500000000E1      (Y/N)?N
5.425000000E3      (Y/N)?N
5.425000000E3      (Y/N)?N
2.500000000E1      (Y/N)?N
2.170000000E2      (Y/N)?Y
ENTER PIVOT VALUE.273
ENTER RELATION.≥
CONTINUING
ANY MORE EXAMPLES (Y/N)?N
DO YOU WANT TO TRY IT OUT?(Y/N)Y
```

```
R←PROG;T;S;016;08;093;047;093;047;06;03;0147;074;0
S←⍳T←0
03←T←□
S←03,S
T←03×06←□
S←T,S
T←0
T←T+□
S←T,S
T←T×016←□
S←1⌽S
T←T+1↑S
S←1↓S
S←T,S
→(2.730000000E2>016)↑L104
T←0
S←1⌽S
S←1⌽S
T←T+1↑S
T←T×80
S←1⌽S
T←T+1↑S
S←1↓S
S←T,S
S←1⌽S
T←0
T←T+1↑S
→(2.730000000E2>06)↑L76
T←T×80
S←1⌽S
S←1⌽S
T←T+1↑S
S←1↓S
S←T,S
S←1⌽S
T←0
T←T+1↑S
S←1↓S
T←T+1↑S
S←1↓S
S←T,S
S←1⌽S
T←0
T←T+1↑S
S←1⌽S
T←T+1↑S
T←T-80
→0,R←T
L76:S←1↓S
```

```
T←T+1↑S
S←1↓S
S←T,S
S←1ϕS
T←0
T←T+1↑S
S←1ϕS
O93←T÷1↑S
→(2.730000000E2>O93)↑0,R←O93
→(3.530000000E2>O93)↑L96
T←O93-80
→0,R←T
L96:T←0
T←T+273
→0,R←T
L104:→(2.730000000E2>O6)↑L26
T←0
S←1ϕS
T←T+1↑S
T←T×30
S←1ϕS
S←1ϕS
T←T+1↑S
S←1↓S
S←T,S
S←1ϕS
T←0
T←T+1↑S
S←1↓S
T←T+1↑S
S←1↓S
S←T,S
S←1ϕS
T←0
T←T+1↑S
S←1ϕS
O147←T÷1↑S
→(3.530000000E2>O147)↑L196
T←O147-80
→0,R←T
L196:→(2.730000000E2>O147)↑0,R←O147
T←0
T←T+273
→0,R←T
L26:S←1ϕS
T←0
T←T+1↑S
S←1↓S
T←T+1↑S
```

```
S←1↓S
S←T,S
S←1φS
T←0
T←T+1↑S
S←1φS
T←T÷1↑S
→0,R←T
□:
        20
□:
        194
□:
        10
□:
        210
199.3333333
DO YOU WANT TO TRY IT OUT?(Y/N)N
ANY MORE EXAMPLES (Y/N)?N
```

APPENDIX VIII

OTHER APPROACHES TO THE PROBLEM

Many disciplines other than compouter and information science are attempting to grapple with the process of communication and with its underlying structure. The various approaches used in these disciplines may help us come to a better understanding of the problem.

The subject of this thesis was suggested to me by Dr. John W. Carr III. He described the problem as the induction of an incompletely specified finite state machine from traces of its output function (data or test operations) and inputs (T,F). The finite state machine when induced was to be able to accept (generate) the language of all traces. In this thesis the program output by the simulation is the finite state machine which has been inferred from the traces left by the user on the programmable device.

Although it may not be immediately apparent, modern mathematical linguistics is deeply engaged in investigation of the same problem. The notion of an acceptor or generator of valid strings of a language has been used by Noam Chomsky in his theory of transformational grammars for natural language. Each native speaker of a language constructs a "grammar" which allows him to recognize and produce valid sentences of that language. Chomsky presented three possible models of a grammar to account for human language capabili-

ties: a finite state grammar, a phrase structure grammar, and a transformational grammar, each one more powerful than its predecessor. The finite state grammar is directly analogous to the finite state machine above. However, it is an inadequate grammar for the systematic analysis of natural language since it cannot deal with dependencies holding between non-adjacent word (e.g., He who hesitates is lost). A phrase structure grammar is context free. It is generally represented in linguistic literature by tree structures. Chomsky was dissatisfied with the power of phrase structure grammars in dealing with certain types of structural ambiguity in natural language. He claims that in a certain sense transformational grammars which are context sensitive, are not only more powerful but simpler. In Mathematical Models in Linguistics Maurice Gross outlines the basic mathematical concepts underlying Chomsky's theories. He discusses the relationship between various types of grammar and their counterparts in automata theory. (Turing machines correspond to rewriting systems, finite automata to K-grammars, push-down automata to C-grammars, nondeterministic linear bounded automata to context sensitive grammars)

Another approach to this problem is through algebra and its relationship to automata theory. Any machine has

an associated semigroup (Theorem in Binary Systems, McNaughton Bruck). Carr has discussed in his Notes on Languages and Semigroups a variety of reasons for approaching the problem of grammatical inference from this angle. A detailed discussion of semigroups can be found in Introduction to Discrete Structures by Preparata and Yeh.

One of the earliest attempts at solving this learning problem was made by E. B. Hunt. It is his terminology which I have used extensively in this thesis. He analyzes concept learning or inductive reasoning as the discovery and utilization of a classification rule. (recognition and generation by a grammar) He points out that concepts can be represented as sequential decision rules (trees) and investigates different strategies for developing such trees under various constraints such as limited memory and under various methods of presentation of sample strings from the concept.

This discussion barely scratches the surface of the variety of approaches taken to solve essentially the same problem--induction, inference, learning. There is a short bibliography included in this thesis which can be used as an entry point into the extensive literature available.

APPENDIX IX

BIBLIOGRAPHY

Carr, John W. III

Inference of Computer Programs from their Traces

internal publication, U. of Pennsylvania, 1974

Carr, John W. III

The Moore Machine Model of a Program Machine

internal publication, U. of Pennsylvania, 1974

Carr, John W. III

Notes on Languages and Semigroups

internal publication, U. of Pennsylvania, 1973

Carr, John W. III

Notes on Machine Decomposition

internal publication, U. of Pennsylvania, 1973

Carr, John W. III

Proof that a Set of Productions Generates a Corresponding Semigroup of its Derivation Classes

internal publication, U. of Pennsylvania, 1973

Carr, John W. III

A Tentative Protocol for the Construction of a Substitutional Grammar

internal publication, U. of Pennsylvania, 1973

Epstein, R.

Notes for CIS 680

internal publication, U. of Pennsylvania, 1974

Fu, K. S.

A Survey of Grammatical Inference

Gross, Maurice and A. Lentin

Introduction to Formal Grammars

Gauthier-Villars, Paris, 1970

Gross, Maurice

Mathematical Models in Linguistics

Prentice-Hall Inc., Englewood Cliffs, N.J., 1972

Hockett, Ch. F.

Language, Mathematics, and Linguistics

Mouton & Co., The Hague, 1967

Hunt, Earl B.

Concept Learning an Information Processing Problem

John Wiley and Sons, Inc., N.Y. and London, 1962

Hunt, Earl B. and J. Marins and Philip J. Stone

Experiments in Induction

Academic Press, N.Y. and London, 1966

Lyons, John

Chomsky

Wm. Collins & Co. Ltd., London, 1970

Miller, George Arthur

Finite State Machine Induction

Master's Thesis, Moore School of Electrical Engineering

U. of Pennsylvania, 1969

Miller, G. A., E. Galanter and K. H. Pribham

Plans and the Structure of Behaviour

Holt, N.Y., 1960

Pao, Tsyh-Wen Lee

A Solution of the Syntactical Induction Inference Problem for a Non-Trivial Subset of Context-Free Language

Ph.D. Dissertation, U. of Pennsylvania, 1969

Preparata, F. P. and R. T. Yeh

Introduction to Discrete Structures

Addison-Wesley Publishing Co., 1973

Simon, H. A. and L. Siklossy (eds.)

Representation and Meaning: Experiments with Information Processing Systems

Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972