13/2/76

"AN EXPERIMENT IN MULTIPROCESSING COMPILATION"

by

GEORGE RICHARD WILEY

B.Sc.(Sydney), Dip.N.A.A.C.(Sydney)

A thesis
presented for the degree
of
Master of Science
in the
University of Adelaide.

Based on work performed in the
Computer Centre, South Australian Institute of Technology
in association with the
Department of Computing Science, University of Adelaide.

March, 1975.

TABLE OF CONTENTS

SUMMARY vi

STATEMENT OF AUTHENTICITY viii

MAIN TEXT

1. INTRODUCTION 1

2. DESCRIPTION OF THE CONTROL DATA CORPORATION 6400 COMPUTER SYSTEM 5

2.0 Introduction 5

2.1 The Central Processor 5

2.2 The Central Memory 8

2.3 The Peripheral Processors 8

3. DESCRIPTION OF THE SCOPE OPERATING SYSTEM 14

3.0 Introduction 14

3.1 The Central Memory Resident 19

3.2 The Peripheral Processor Resident 23

3.3 The Monitor 24

3.4 The Circular Buffer Input/Output Program - CIO 26

4. OVERVIEW OF THE MULTIPROCESSING FORTRAN COMPILATION SYSTEM 30

4.0 Introduction 30

4.1 The Compilation Process 31

4.2 Multiprocessing Aspects of the Compilation Process 33

4.3 Large Source Subprograms 37

5. CENTRAL MEMORY COMPILATION SYSTEM TABLES 40

5.0 Introduction 40

5.1 Miscellaneous Information 40

5.1.0 The Access Switches 40

5.1.1 The Processor Status Table 44

5.1.2 The Processor Communication Buffer Pointers 45

5.1.3 The INPUT Buffer Parameters 46

5.1.4 The DO Table Pointers 46

5.1.5 The Error Table Pointers 47

5.1.6 The Name Table Pointers 47

5.1.7 The Statement Label Table Pointers 47

5.1.8 The Program PP4 Parameters 47

5.1.9 The File Environment Tables 48

5.1.10 The Error Message Table 48

5.1.11 The Central Memory Table Full Flag Word 48

5.1.12 The Subprogram Name and Header Flags 49

5.1.13 The COMMON Block Name Table 49

5.2 The Processor Communication Buffer 50

5.3 The Name Table 51

5.3.1 Subroutine Name 52

5.3.2 Function Name 52

5.3.3 Subroutine Subprogram (Header) Name 52

5.3.4 Function Subprogram (Header) Name 52

5.3.5 Variable Name 53

5.3.6 Statement Function Name 53

5.3.7 Namelist Name 54

5.4 The Statement Label Table 54

6. PROGRAM GSR - GENERAL STATEMENT RECOGNIZER 56

6.0 Introduction 56

6.1 Explanatory Notes to the Flow-Chart 57

7. PROGRAM PCR - PROCESSOR CALL ROUTINE 100

7.0 Introduction 100

7.1 Explanatory Notes to the Flow-Chart 101

8. PROGRAM DSA - DECLARATIVE STATEMENT ANALYSER 107

8.0 Introduction 107

8.1 Explanatory Notes to the Flow-Chart 108

9. PROGRAM FEA - FORTRAN EXPRESSION ANALYSER 161

9.0 Introduction 161

9.1 Explanatory Notes to the Flow-Chart 163

10. PROGRAM MSA - MISCELLANEOUS STATEMENT ANALYSER 209

10.0 Introduction 209

10.1 Explanatory Notes to the Flow-Chart 210

11. PROGRAM CTM - CENTRAL MEMORY TABLE MOVER 264

11.0 Introduction 264

11.1 Explanatory Notes to the Flow-Chart 265

12. PROGRAM PP4 - WINDUP ROUTINE 272

12.0 Introduction 272

12.1 Explanatory Notes to the Flow-Chart 274

13. COMPILATION SYSTEM TESTING 297

13.0 Introduction 297

13.1 The Assembler 298

13.2 The Simulator 298

13.3 An Example 300

14. CONCLUSION 311

14.0 The Multiprocessing Compiler in Retrospect 311

14.1 The Generation of Object Code 312

14.2 Alternative Approaches 314

14.3 Acknowledgements 316

TABLES

1. CENTRAL PROCESSOR OPERATION CODES 317

2. PERIPHERAL PROCESSOR OPERATION CODES 319

APPENDICES

1. FLOYD PRODUCTION LANGUAGE PROGRAM 321

2. THE SOURCE LANGUAGE 327

3. THE COMPILATION ERROR NUMBERS, CODES AND MESSAGES 330

REFERENCES 334

SUMMARY

This thesis describes an experiment in multiprocessing compilation using a set of independent, highly parallel processors, each having its own memory as well as access to a common memory.

The thesis commences with a brief survey of the development of parallelism in hardware and software. This first chapter goes on to specify the nature of, and reasons for, the experiment which was undertaken. The next two chapters are devoted to a description of the hardware and software chosen for the experiment. This is followed by an overview of the multiprocessing FORTRAN precompiler which was evolved as a result of the experiment. This latter chapter describes how the precompilation task is spread amongst the seven programs which constitute the system and how these programs communicate with each other via the common memory. The organizational aspects of, and the restrictions on, the concurrent running of different programs as well as multiple copies of the same programs is also described. A description of the information stored in the common memory is next given. The mechanism through which this information is accessed by the multiple processors is described in detail. This access mechanism is compared and contrasted with the work of Dijkstra and others in this field.

Each of the next seven chapters provides a description of the main features of one of the seven constituent programs and includes a flow-chart, cross-referenced with additional explanatory notes where necessary. The next chapter briefly reviews the experimental method adopted. It also gives an example of how the system performs the precompilation process by means of a detailed analysis of twenty system "snapshots" taken during the processing of a

sample FORTRAN source program. The thesis concludes with a brief review of the multiprocessing aspects of compilation and a discussion of some possible alternative approaches.

STATEMENT OF AUTHENTICITY

This is to certify that this thesis contains no material which has been accepted for the award of any other degree or diploma in any University. To the best of my knowledge and belief, it contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signed

(G.R. WILEY)

1. INTRODUCTION

Since the development of the first stored-program computer (EDSAC), much further research has been dedicated to increasing the speed with which computational tasks can be performed by computing devices. This has been achieved by speeding up the various components of these devices as well as reorganising the components and/or the devices themselves so that operations previously carried out sequentially were subsequently performed concurrently. There are numerous examples of the latter. Early in computer hardware development, serial adders were replaced by parallel units. Then input/output operations were separated from the computational operations of the processor so that both could proceed simultaneously and this development culminated in the use of separate controlling processors for input/output and computation such as in the CDC 6000 Series and ICL 1906A systems. Instruction execution times have effectively reduced by overlapping the execution of several instructions by the provision of multiple arithmetic units as in the CDC 6600 system. Inefficiencies due to the tremendous differential in the speeds of computation and input/output were reduced by multiprogramming several programs together.

However, as it was becoming increasingly apparent that the ultimate speed limit for single processor systems was rapidly being reached, attention has most recently been turned to systems involving several processors. A number of such systems have been experimentally developed and of these the ILLIAC IV, produced at the University of Illinois, is the best example. The latter system consists of an array of 256 processors, each of which simultaneously executes the same instruction (stored

in a common memory) on its own set of data in its own (small) memory. Commercially available multiprocessing systems are much simpler and are usually the outgrowth of a multiprogramming system to which additional processors have been added - the CDC 6500 system being an example. In this type of system, each processor is independent of the others and generally does not cooperate with these other processors in carrying out its task. This is, however, not completely true of the CDC 6400 and CDC 6600 systems which are single-processor systems from the user program point of view and multiprocessor systems from the operating system view since system and input/output functions are carried out in parallel by ten identical independent processors, called peripheral processors.

The majority of research into multiprocessing systems has been into areas such as the detection and exploitation of parallelism in computational algorithms [9, 13, 14, 18, 19], the efficient scheduling of multiprocessor systems [1, 2, 11, 16] and the problems of cooperation between processors [5, 6, 12, 15, 17, 20]. Only Ellis (7) seems to have investigated the possibility of multiprocessing the compilation task. Yet compilation can be a major processing load at installations such as universities and therefore worthy of being seriously considered for multiprocessing. Ellis (7) describes two parallel compilation techniques both of which are completely dependent on an experimental, highly specialised parallel processing device of the ILLIAC IV type, developed at Bell Telephone Laboratories. Ellis writes: "It is generally claimed that the machine on which a compiler is to be implemented should have little effect on the compiler design".

Yet he appears to have ignored this in his research, apparently without even attempting to develop a more machine-independent technique, but brushes it aside by continuing: "Compilation on a parallel machine is a blatant exception to this rule, so it is worthwhile to investigate how one can take advantage of the parallel computer architecture in the compiler design". This is in marked contrast to the work of the present thesis which is based on a set of general-purpose, highly independent parallel processors.

This type of multiprocessor hardware configuration is increasingly being produced by various manufacturers. For example, the set of ten peripheral processors and central memory of the CDC 6400 system provides hardware of this type and was therefore adopted as the vehicle for experimenting with multiprocessor compilation in this thesis. However, since the peripheral processors of the CDC 6400 are not normally accessible to users of the system and since there are no diagnostic facilities available for programs running in these processors, the experiment was carried out on a simulator (see 13.1 and 13.2) written to run on an ICL 1903A system. For the purposes of the experiment, it was decided to develop a FORTRAN compiler in which as many phases of the compilation task as possible are being performed concurrently. To further demonstrate the feasibility of the concept, the implemented compiler was to be compatible with Control Data's RUN compiler (3) and SCOPE operating system (4). However, it was not intended to implement object code generation since the latter would contribute little to the experiment but would greatly increase the size of the experimental task. In this

sense, the system as implemented might be referred to as a "precompiler" although in the remainder of the thesis the more usual term "compiler" is used.

The remaining chapters describe the system which developed as a result of this experiment. Chapters 2 and 3 describe the hardware and operating system on which the multiprocessing compiler was implemented. Chapter 4 outlines the system itself, the components of which are fully described in Chapters 5 through 12 inclusive. The testing of the system is discussed in Chapter 13 whilst the final chapter outlines a possible implementation of object code generation and also discusses some alternative approaches.

2. DESCRIPTION OF THE CONTROL DATA CORPORATION 6400 COMPUTER SYSTEM

2.0 Introduction

The Control Data Corporation 6400 computer system is composed of one central processor and ten peripheral processors, all of which have access to a large central memory having sixty bits per word. In addition to this, each peripheral processor has its own 4K memory of twelve bits per word. The latter memory is accessible only to the peripheral processor to which it is attached. All ten of these peripheral processors can execute programs independently of each other or the central processor.

In effect, the central processor is a very high speed arithmetic device capable of multiprogramming a number of programs stored in the central memory. The peripheral processors perform operating system functions and control input/output channels thereby providing a flexible arrangement for very high speed communication with a variety of input/output devices. The instruction set of the peripheral processors (see table 2 and 2.3) enables them to control the running of the central processor and therefore the central processor may be regarded as the "slave" of the peripheral processors.

2.1 The Central Processor

The central processor has three groups of eight registers each, designated the X, A and B registers and referred to as X0, X1, ..., X6, X7, A0, A1, ..., A6, A7, B0, B1, ..., B6, B7. The X registers are each sixty bits long and are used as temporary data result registers. Both the A and B registers are eighteen bits wide and are used for

central memory addressing and indexing, respectively. Whenever an eighteen bit number is stored in any of registers A1 to A5, the contents of the memory location whose address corresponds to that eighteen bit number is stored in the corresponding X register. Similarly, whenever the contents of A6 or A7 are altered, the contents of the corresponding X register is stored in the memory location whose address is given by the A register contents. The B0 register is hard-wired to be permanently zero.

The arithmetic unit performs the one's complement addition and subtraction of sixty bit and eighteen bit (for addressing purposes) integers. It further provides facilities for the addition, subtraction, division and multiplication of floating-point numbers. It also performs left (circular) and right (end-off with sign extension) shift operations and the basic logical operations of logical product, logical sum and logical difference. It is worth noting that no facilities are available for direct multiplication or division of integers. This must be performed via floating-point.

The central processor has the capability of multiprogramming up to seven user programs held in the central memory (see 3.0). User program integrity is ensured by dynamic memory protection which is provided by two hardware registers; the reference address (RA) register and the field length (FL) register and all memory references are checked against FL and modified by RA (see 2.2).

The central processor instructions fall into three categories : those that reference working registers

only (fifteen bits long), those that reference both working registers and central memory locations (thirty bits long) and those that load constants into working registers (thirty bits long). The arithmetic and logical instructions operate only on the X registers. There are two groups of conditional branch instructions; those which are dependent upon the contents of the X registers and those which compare the contents of pairs of B registers, including B0 which is always zero.

One to four instructions are packed into a memory word, but an instruction cannot be split across two words. Also branch instructions transfer control to the first instruction in a word. To accommodate these situations, it is necessary to fill out the remainder (if any) of the current instruction word with fifteen bit "no operation" instructions and insert the next instruction at the beginning of the next instruction word.

If the central processor executes an instruction with a zero operation code, it stops and can only be restarted by a peripheral processor. Any peripheral processor has the capability to do this, but the SCOPE operating system provides that only one peripheral processor, the monitor (see 3.3), monitors the performance of the central processor.

From table 1, it will be seen that the central processor is incapable of performing input/output operations. All such functions can only be performed by peripheral processors as they alone have access to input/output channels (see 3.4).

2.2 The Central Memory

The central memory is organised into 32K, 65K or 131K words of sixty bits each, in eight, sixteen or thirty-two banks of 4096 words each. These banks are logically independent and consecutive addresses occur in different banks. Banks may be phased into operation at one hundred nanosecond intervals whilst a memory access takes one microsecond to complete. The memory is accessible to the central processor as well as the ten peripheral processors. Memory access requests are satisfied under a priority system in which the central processor is given preferential treatment. The ten peripheral processors share common read and write paths to the central memory. Each path passes through a device known as a pyramid which assembles or disassembles, as appropriate, five, twelve bit peripheral memory words into one central memory word. The process takes five peripheral processor memory cycles, that is five microseconds, to complete but up to four peripheral processors can share the same pyramid provided their initial memory access requests do not enter the pyramid faster than at one microsecond intervals apart.

2.3 The Peripheral Processors

There are ten peripheral processors, each with its own 4K memory of twelve bit words with a cycle time of one microsecond. In addition, each peripheral processor has access to the central memory (see 2.2).

Each processor has its own eighteen bit accumulator which is used as one of the operands in almost all instructions. The instructions themselves are either one word (twelve bits) or two words (twenty-four bits) long. One word instructions have a six bit operation code and a six bit operand or operand address. The latter can only access the first sixty-four words of memory. Two word instructions also have a six bit operation code but specify a twelve bit quantity in addition to the six bit quantity and these are used together to form either an eighteen bit operand or a twelve bit direct address with a six bit address modifier location.

There are four address modes available. In the "no address" mode, the six bit (one word instruction case) or eighteen bit (two word instruction case) operand is used directly. In the "indirect address" mode, the six bit quantity specifies an address the contents of which is the address of the desired operand. In the "indexed address" mode, the address of the required operand is generated from the twelve bit quantity to which is added the contents of the location specified by the non-zero six bit quantity. In the "direct address" mode, the operand address is given by the six bit quantity (one word instruction case) or the twelve bit quantity (two word instruction case) if the six bit quantity is zero.

Table 2 contains a list of peripheral processor instructions and some of these are worthy of further comment.

The arithmetic unit has facilities for six, twelve and eighteen bit addition and subtraction using one's complement arithmetic, but there is no hardware available for multiplication and division of integers nor floating-point operations.

The single shift instruction performs left circular and right end-off with sign extension shifting with positive or negative constant operand, respectively. Variable length shifting facilities are not available and can only be achieved by instruction generation.

The "exchange jump" instruction (see 3.0) is used to control the multiprogramming facility of the central processor (see 2.1). The "read program address" instruction provides a means of monitoring the running of the central processor.

The central memory access instructions transfer either a single central memory word or a block of words from the peripheral memory to the central memory or vice versa. All central memory references are in terms of absolute addresses since peripheral processor access to the central memory is not subject to memory protection hardware. However, the SCOPE operating system provides a subroutine (see peripheral processor resident, 3.2) which checks that a central memory address is within the field length and adds in the reference address of the central processor job to which the peripheral processor is attached.

All conditional branch instructions specify a six bit relative address, that is, if the condition is true processor control is transferred to an address which

must be within thirty-one locations of the address at which the branch instruction is stored.

There is no stop or halt instruction. Thus, a peripheral processor program which has reached completion must call an operating system subroutine, which will alert the controlling peripheral processor, the monitor, to the fact that this processor wishes to "drop" itself. Meanwhile, the peripheral processor enters an operating system provided idle loop during which it will periodically scan a special word in central memory (its input register, see 3.1) into which the monitor will insert the details of a program which it is to load into itself from the system library and start executing.

By design (see 2.0), the peripheral processors are small, slow computing devices unsuited to carrying out long and complex computations or even performing involved character manipulations. These observations are borne out by an inspection of the available peripheral processor instructions given in table 2 and the fact that there are only 3588 words of peripheral processor memory available for program and data storage, since locations 0_8 to 74_8 can only be used for indexing purposes and intermediate data storage whilst location 75_8 to 772_8 are reserved for the SCOPE operating system. Thus, to code even the simplest operations requires large numbers of instructions with the result that programs can quickly grow in size to outstrip the limited memory space available.

Two examples will help to emphasise the problem of the lack of power of the instruction set.

1) Consider that the leftmost bit of central memory location TABLE must be tested by a peripheral processor program and that within the program there is a five word area beginning with a location called BUFFER which can be used as a scratch area. The following code will transfer control to a location called ON if the required bit is non-zero.

```
LDC TABLE             load central memory relative address
RJM R.TFL             call system routine to convert it to absolute
CRM BUFFER,CONST1     read in the central memory word
LDM BUFFER            load first byte
SHN -11               shift out unwanted bits
ZJN *+3               skip next instruction if bit is zero
LJM ON                otherwise jump to required location
```

where CONST1 is a word, in the first sixty-one locations of peripheral memory, containing the integer value one and R.TFL is a system routine entry point in the peripheral processor resident (see 3.2). The code occupies twelve words of storage.

2) Consider the problem of the addition of two positive eighteen bit quantities with the eighteen bit result being stored in the memory. Each quantity is stored in two consecutive memory locations and the two operands are stored in location pairs starting at NO1 and NO2 while the result is to be stored at the location pair starting at NO3.

The following code will perform the desired computation:

```
LDM NO1+1       load lower twelve bits of first operand
ADM NO2+1       add in lower twelve bits of second operand
STM NO3+1       store lower twelve bits of result
SHN -12         shift out lower twelve bits of result
ADM NO1         add in upper six bits of first operand
ADM NO2         add in upper six bits of second operand
STM NO3         store upper six bits of result
```

This code requires thirteen words of storage.

It is worthy of note that during the development of this project, it became necessary, for the reasons discussed above, to rewrite the code of one program and split the code of another into two separate programs.

3. DESCRIPTION OF THE SCOPE OPERATING SYSTEM

3.0 Introduction

The SCOPE operating system, of which version 3.2 is described here, is a file-oriented system; all information within the system is held in the form of one or more files, and in particular, each user computing task is designated a job file. A job file takes the form of a deck of cards (the job deck) which begins with a job card containing job parameters such as job name, time limit and central memory field length and terminates with an end-of-file card (6-7-8-9 punched in column 1). As for all files, a job deck consists of one or more logical records, each terminated by an end-of-record card (7-8-9 punched in column 1). The first record of a job file is a set of control cards; each such control card represents an instruction to SCOPE to load and run a central processor (or peripheral processor) program, to initiate a system function such as a file allocation request or a file handling request (such as REWIND file, BACKSPACE logical record).

Figure 3-1 shows a typical job deck for a FORTRAN compilation and execution. The job card specifies a job name of JOB123, priority of 6, central processor time limit of 400 seconds and central memory field length of 4500 octal words. The control card record consists of a single card, being a call to the FORTRAN compiler, which will, on successful completion, automatically initiate execution of the generated object code. RUN is the name of the FORTRAN

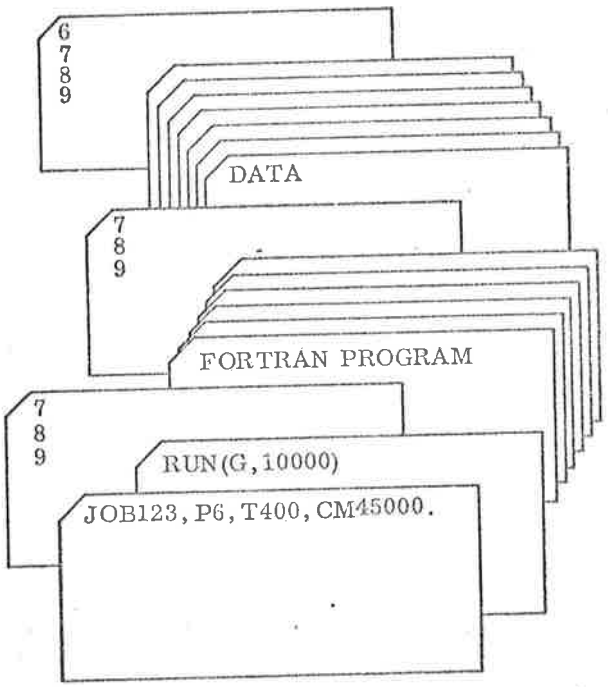

6
7
8
9
DATA
7
8
9
FORTRAN PROGRAM
7
8
9
RUN(G,10000)
JOB123,P6,T400,CM45000.

Figure 3-1

compiler. The second record comprises the FORTRAN source program and the third record is the source of "card-reader" data to be read by the object program during execution (if any).

The job file is created from the job deck as soon as it enters the system via the card-reader. SCOPE stores this job file on disc and enters in its file table the job name together with a job priority calculated from the job card parameters and the current state of the job stack. The priorities of all jobs in the job stack increase as new jobs enter the system and when a new job is to be initiated, SCOPE selects the job with highest priority which will fit the available free central memory storage, after shuffling jobs already in core if necessary.

When SCOPE eventually begins the execution of a job, it causes the first record of its file to be read into a control card buffer, thus positioning the job file, which is now renamed INPUT, at the beginning of the second logical record (if any) and executes each control card sequentially starting with the first. Thereafter, the file INPUT becomes the immediate source of card input and three other disc files are also automatically assigned (if referenced) to each job. They are: OUTPUT, the immediate destination of printer output and PUNCH and PUNCHB, the immediate destinations of coded and binary output, respectively. As each control card is read and interpreted by SCOPE, the control card image together with other systems generated messages is inserted into yet another automatically assigned file called the DAYFILE file. A job terminates when the

last control card has been processed or a fatal error is flagged and, in either case, the contents of DAYFILE is appended to OUTPUT, after which the contents of INPUT is irretrievably lost while the files OUTPUT, PUNCH and PUNCHB (if they exist) are queued for output when a peripheral of the required type becomes available. They are subsequently erased. In this way, when the user receives his printout, the last page will contain a detailed history (the DAYFILE) of the flow of his job through the system.

The SCOPE operating system itself comprises an area of central memory low core, called the Central Memory Resident, and a large number of peripheral processor programs held in a disc library.

The Central Memory Resident holds system tables such as equipment status tables, filename/file status tables and two central processor programs - one is merely a processor idle loop and the other moves jobs around in central memory so that all free core space becomes contiguous. In addition, there are eight areas, numbered 0 to 7, called control points. Each central processor program is assigned to a control point, and since control point 0 is reserved for the running of either of the two central processor programs described above, a maximum of seven user jobs can reside in the central memory simultaneously. However, one or more of these seven control points could be assigned (by the operator via the operator's console) to a program which controls the card-reader in order to create job files or to another program which drives printers and punches in order to dispose of output files queued to these devices.

The peripheral processor programs which make up the SCOPE operating system fall into two categories - those which reside in dedicated peripheral processors and those which are held in the library on disc and are loaded into free peripheral processors as required. The monitor program, MTR, has overall control of the system. Its chief task is to monitor the running of the central processor and load peripheral processor programs from the library on request from central processor programs and other peripheral processor programs. The other operating system peripheral processor program that resides in a dedicated processor is the display console control program, DSD. This program controls the operation of the twin cathode ray tube display console which is used to display system messages to the operator and accept commands which are then passed on to the monitor for further action. The remaining peripheral processor programs are of the type that are loaded from the systems library on demand. The only program in this category, which is of interest to this thesis, is CIO. This program, when attached to a job by the monitor, performs all input/output operations between the data channels and a circular buffer within the field length of the job. Program CIO can handle any type of file since it loads its high end of core within an overlay to handle the required peripheral type. Due to the nature of the pointers which are associated with this circular buffer, it is possible for the central processor program and program CIO to be both accessing different parts of the buffer at the same time without conflict.

The area from 75_8 to 772_8 of every peripheral processor's memory contains a set of routines known as the peripheral processor resident. This set of operating system routines provide facilities for making requests to the monitor, loading peripheral processor programs and overlays, modifying central memory addresses by RA and checking against the FL of the job to which the peripheral processor is attached. There are also facilities for communicating with program DSD in order to display messages on the operator's console.

3.1 The Central Memory Resident

This area of central memory provides communication between the various peripheral processors and contains all the information shared by the system. Both upper and lower memory are used for system storage and both areas are variable length at execution time. The following is a brief description of the central memory resident areas which concern this thesis.

3.1.1 The central memory resident pointer area contains various flags and pointers to tables stored elsewhere in central memory.

3.1.2 The peripheral processor communication area contains ten eight-word areas, one for each peripheral processor, through which the peripheral processors communicate with each other, the monitor and the display console control program. Each eight-word area comprises a one-word input register, a one-word output register and a six-word message buffer. Each peripheral processor has pointers to its input

register, output register and message buffer in peripheral processor memory locations 75_8, 76_8 and 77_8, respectively.

When a peripheral processor is idle, its resident program periodically scans its input register. As soon as the monitor has a task for that processor, it inserts the name of the appropriate program in that processor's input register. When the idle processor recognises the request, it loads the program into itself and starts executing it. Monitor regularly scans the output register of each active peripheral processor. When a peripheral processor requires monitor assistance, it places a code in its output register. Monitor detects the request during its scan of the output registers and processes it. When the request has been accepted, the requesting processor is informed of the fact by having its output register cleared by monitor.

3.1.3 Each job in central memory is assigned to a control point. The control point area contains information about the job such as its name, length, starting address in central memory, time used so far, details about input/output equipment assigned to it, and a control card buffer containing the control card record of the job. The control point area also holds a sixteen-word section called an exchange package, which contains all of the information to

start or resume the execution of a central processor program. This consists of the program address, the reference address (RA), the field length (FL) and the current values of the A, B and X registers. The format of the exchange package is shown in figure 3-2. Monitor forces the central processor to resume the running of an interrupted program by executing an exchange jump instruction. This instruction causes the central processor to save the current information about the program, which was running when the exchange jump was executed, in its exchange package. The central processor then enters the information about the program, which is to be run next, from the program's exchange package into the appropriate registers, thereby exchanging the two programs.

3.1.4 The central processor resident area contains the two programs mentioned in 3.0, which run at control point 0. They are the storage move program and the idle loop.

3.1.5 The equipment status table contains one entry for each peripheral device attached to the system.

3.1.6 The channel status table is maintained by the monitor to contain data channel reservation information. A channel can be reserved by a peripheral processor via a monitor request. It is released directly by the peripheral processor concerned without monitor intervention.

				Words
	Program Address(P)	A0(Address Registers)		0
	Reference Address(RA)	A1	B1(Increment Register)	1
	Field Length(FL)	A2	B2	2
	Exit Mode(EM)	A3	B3	3
	RA - ECS	A4	B4	4
	FL - ECS	A5	B5	5
		A6	B6	6
		A7	B7	7
		X0 (Operand Registers)		10
		X1		11
		X2		12
		X3		13
		X4		14
		X5		15
		X6		16
		X7		17

Figure 3-2

bits 59 17 6 0

file name	not used	code	Name (display code left adjusted) and operation	
not used	FIRST		Beginning address	
not used	IN		Current input address	of circular buffer
not used	OUT		Current output address	
not used	LIMIT		Last address + 1	

Figure 3-3

3.1.7 The file name/file status table is composed of as many three-word entries as there are files in the system. A table entry is set up at the time that a file is created, but is not accessible to the user. This table provides a linkage between the user program and all required data channels and functions.

3.2 The Peripheral Processor Resident

In the SCOPE operating system, the monitor and the display console control program permanently reside in two of the ten peripheral processors. The remaining processors form a pool of processors to which monitor may assign tasks as required. The pool processors have no fixed assignments; any processor may be assigned to the execution of any system routine. In fact, it is possible that more than one processor may be executing the same routine at the same time.

All the processors contain a small resident program (the peripheral processor resident) which handles the communications between pool processor programs and the monitor and loads and initiates the execution of programs, as directed by the monitor. Communication is performed via the peripheral processor communication area (see 3.1). Monitor assigns a task to a pool processor by placing the request in the processor's input register. The name of the program which is to be loaded and executed appears in the upper eighteen bits of the input register. This name consists of three characters, such as CIO, GSR and CTM. The request remains in the input register until the task

is completed. On task completion, the peripheral processor program drops itself by advising monitor that it has finished. Monitor flags the processor inactive by clearing its input register. The processor then enters an idle loop during which it periodically scans its input register (see 3.1.2). The resident routine, which handles requests to monitor via the processor's output register (see 3.1.2), waits for the output register to be cleared. This ensures that monitor has at least accepted the request before the resident routine returns control to the program. The resident routine which converts central memory addresses from relative to absolute, after checking that the address is within the field length, uses values of RA and FL held locally within the peripheral processor resident. These values must be reset after central memory storage has been moved.

3.3 The Monitor

Monitor resides permanently in a dedicated peripheral processor from dead-start time and has overall command of the system. It does not, however, perform operating system tasks directly but rather assigns them to available pool processors. Operating system tasks are never initiated by the monitor but result from requests made by central and peripheral processor programs and the operator via the operator's display console. These tasks may be requests to load peripheral processor programs from the system library. Alternatively, they may be more complex requests for which monitor requires several such peripheral

processors to be loaded and executed.

If no peripheral processor is immediately available, the monitor inserts an entry into a peripheral processor request stack held within its own memory. When a peripheral processor subsequently becomes free, it is assigned a new program from the entry at the top of this request stack. When a peripheral processor requests monitor to load a program into another peripheral processor, it may specify a delay before the expiration of which the requested program must not be loaded. For this type of request, the monitor inserts an entry into a delay stack which is also held within monitor's own memory. When the appropriate time comes, monitor moves entries from the delay stack to the peripheral processor request stack.

The method by which peripheral processor programs request monitor action has already been described (see 3.1.2).

A central processor program may obtain monitor action by entering the name of a peripheral processor program in central memory location RA+1. Monitor periodically scans location RA+1 of the currently running program for such requests. When RA+1 is non-zero, monitor passes the value to a peripheral processor for action. Requests such as input/output (carried out by program CIO, see 3.4) are processed in this manner. When RA+1 has been cleared by monitor, the running program may assume that the request has been accepted, though not necessarily completed. The program may then request monitor to suspend it until the requested peripheral processor program has completed its

task. Any parameters associated with a request to monitor must be put in the lower thirty-six bits of location RA+1 by the calling program. The format of the parameter list is dependent upon the peripheral processor program called. System requests can be generated by the operator via the operator's display console. The operator types in a command on the console keyboard and when this is accepted by the display console program, DSD, it is syntactically analysed and, if valid, broken down into one or more monitor functions. Program DSD then issues these monitor functions in the usual way via its output register. Messages can be sent to the operator's console by peripheral processor programs. The message is inserted into the peripheral processor's message buffer and a special function code is loaded into the output register. This code is ignored by the monitor during its regular output register scan, since monitor knows it to be directed towards program DSD. The latter program will, in fact, recognise the display request during its regular scan of the output registers of active peripheral processors. The contents of the message buffer is then displayed on one or other of the twin screens of the operator's console in the appropriate format.

3.4 The Circular Buffer Input/Output Program - CIO

Program CIO performs input/output between a file and a circular buffer in central memory. The requesting program must specify a file name and an operation code, plus information about the buffer to program CIO, which then performs the requested operation.

Before calling program CIO, circular buffer parameters must be set up in a file environment table (FET). The format of a file environment table is shown in figure 3-3 and contains information such as the file name, the start address of the buffer (called FIRST), the current buffer input address (IN), the current buffer output address (OUT) and the last address of the buffer plus one (LIMIT) as well as a code to indicate the required operation. The buffer and the file environment table must both be within the field length of the job and all addresses are relative to the reference address (RA) of the job.

A central processor program calls program CIO by entering the request via RA+1 (see 3.3). The operation code in the first word of the FET is even and one is added to it by program CIO when it has performed the requested operation. Program CIO also updates IN and OUT in the FET according to the function performed.

File operations provided by program CIO include read and write, backspace, and write end of record mark. It should be noted that because card-reader and line printer files are simulated (or off-lined) on disc, it is possible to backspace and rewind such files.

In a CIO write operation, data is transferred out of the buffer in sixty-four central memory word blocks only. However, write end record will empty the buffer completely by writing a short block, if necessary. Similarly for a CIO read operation, data is inserted into the buffer in sixty-four word blocks, except that the last block of a

logical record may be shorter. The status code returned by program CIO indicates whether the end of record or end of file condition terminated a read operation.

Reference to figure 3-3 will show that the last seven bits of the first word of the FET is used to transmit an operation code to program CIO and for the latter to return a status code (for the read operation only) to the central processor program.

Program CIO performs input/output in the following way. For input, CIO reads data into the buffer beginning at IN. It continues reading as long as storage is available in the buffer or until the data is exhausted. IN is advanced by one as each data word is inserted into the buffer. When IN = OUT - 1, the buffer is full and the operation is complete. If IN≥OUT, program CIO inserts data words into the buffer while progressively advancing IN to LIMIT - 1. Then IN is automatically reset to FIRST by program CIO before it continues the read operation. The maximum buffer capacity for a read operation is LIMIT-FIRST-1. For output, program CIO writes data words from the buffer beginning at OUT. Only complete sixty-four word blocks are written to the output file and no partial or short block is written unless the write end of record operation is specifically requested in the call to program CIO. Unless such a request was made, program CIO continues extracting from the buffer until there is insufficient data in the buffer for a full sixty-four word block. OUT is advanced by one as each word is removed from the buffer.

When OUT = LIMIT - 1, program CIO automatically resets OUT to FIRST and continues the write operation. When program CIO has completed the requested operation, it signals completion by adding one to the operation code in the lower seven bits of the first word of the FET and drops itself.

4. OVERVIEW OF THE MULTIPROCESSING FORTRAN COMPILATION SYSTEM

4.0 Introduction

The compilation system consists of seven peripheral processor programs which operate on data stored in the central memory. This central memory area holds the usual compiler tables such as the name table and statement label table as well as tables unique to the system. Amongst the latter are processor communication buffers to allow data to be passed from one program to another via central memory. There is also a processor status table whose entries indicate which compilation system programs are currently in use and what their status is. A full description of this information and its organisation within the central memory area is given in Chapter 5.

The seven programs distribute the compilation task amongst themselves as follows:

Program GSR controls the extraction of source statements from the INPUT buffer and performs statement recognition

Program DSA completes the compilation of all declarative statements other than ENTRY and statement function definition statements

Program FEA processes all statements involving FORTRAN expressions

Program MSA compiles all other statement types such as READ and GOTO statements

Program PCR determines, and causes the required number of copies of programs FEA and MSA to be loaded, at certain stages during the compilation process

Program CTM expands central memory tables when they are almost full

Program PP4 checks the nesting of DO loops, searches the statement label table for references to undefined labels and produces a listing of the current subprogram.

These programs are described in more detail in Chapters 6 to 12 and their role in the overall compilation process is expanded in the following sections.

4.1 The Compilation Process

The compilation of the first subprogram of a source program commences with the loading of program GSR, which starts by requesting the system peripheral program CIO to fill the INPUT buffer (see 3.4 and 5.1.9) from the user's job file (see 3.0). Since this input operation will require a significant time lapse for completion, program GSR will drop itself after requesting to be reloaded by the monitor after a delay. When the INPUT buffer has been filled, the reloaded program GSR continuously extracts statements from the buffer, recognises them, and passes them on to the next program which will further process them by converting them into one or more entries which are then inserted into the appropriate processor communication buffer. During this preliminary processing, valid statement label fields are inserted into the statement label table,

multiple statement lines (in which statements are separated by the $ character) are split into separate statements and IF statements are divided into an expression part, which is passed on to program FEA for further processing, and either a statement number list part, which is passed to program MSA, or a statement part, which is then recognised and passed to the appropriate program. The first occurrence of a header statement causes program DSA to be loaded. Program GSR continues in this way, periodically requesting program PCR to be loaded, until either it detects the end of the source subprogram or the end of the source itself in which case it flags the appropriate error. In either case, program GSR requests program PCR to be loaded for the last time for the current subprogram and then drops itself.

On being loaded, program DSA commences extracting and processing communication buffer entries representing declarative statements until all such statements have been processed. Then, if necessary, it requests monitor to load programs FEA and/or MSA to process entries that have been entered into their respective communication buffers by program GSR in the interim. For a BLOCK DATA subprogram however, it checks instead that program GSR has detected the end of the subprogram and then calls program PP4 before dropping itself.

Programs FEA and MSA keep extracting and processing entries from their respective communication buffers until empty.

They each then check whether or not to load program PP4 before dropping themselves. If either of these programs drop themselves prematurely because program GSR has not been able to insert entries into their buffers at least as fast as they have extracted them, there will eventually be further entries in these buffers. This will cause program PCR to request the appropriate programs to be once again loaded in order to clear the buffers. This will continue until the buffers are finally cleared and program PP4 loaded.

When program PP4 is eventually loaded, it checks the validity of nested DO loops and looks for statement numbers referenced but not defined. Then, after producing a listing of the current source subprogram which is still intact in the INPUT buffer (see also 4.3), program PP4 checks for the occurrence of another subprogram. If the current subprogram was the last, the compilation run is terminated via the monitor. If, however, there is another subprogram, program PP4 reinitializes the central memory tables and requests further source to be inserted into the INPUT buffer. It then calls program GSR once again to commence the compilation process for the new source subprogram.

4.2 Multiprocessing Aspects of the Compilation Process

Multiprocessing of the compilation system is made possible by the organisation of the necessary control information in the central memory table area. The concepts of the processor communication buffers and the processor status table have already been introduced. However, the integrity

of all central memory tables is absolutely essential and must be maintained at all times. Nevertheless simultaneous access to a table by two or more programs is likely to cause table corruption. Therefore table access is restricted to a single program at any one time by means of a set of access switches controlled by the accessing programs themselves. A full description of the accessing mechanism is given in 5.1.0.

The problem of the validation of nested DO loops is solved by the use of a DO table stored in central memory and is further described later in this section. Since the source listing is not produced until the end of the processing of the subprogram, all generated error numbers must be saved in an error table. As these errors will not be generated in line number order, the contents of the error table will require sorting before final processing by program PP4.

Since each program (except program PCR, see 7.0) currently in use must have an entry in the processor status table and as this table has space for ten entries, as many as ten programs can be in use at any time. However, it is not physically possible for more than eight programs to be actually loaded and running since the SCOPE operating system requires two of the ten peripheral processors to be dedicated to it (see 3.0). Operating system action on peripheral processor allocation requests when none are available is described in 3.3.

Any number of copies of programs FEA and MSA may be running concurrently, provided that all declaratives have been processed by program DSA. The latter requirement

is due to the fact that erroneous errors could be generated, or actual errors missed, if either of programs FEA and MSA were to process a name which appears in a declarative which had not yet been compiled by program DSA. Therefore, these two programs are not loaded by program PCR if program DSA is still running. In fact, program DSA will load them if their communication buffers are non-empty, immediately before dropping itself. Program GSR is loaded only once per source subprogram and may not run concurrently with itself. There are two important reasons for this. Firstly, there is the problem of detecting the end of the current subprogram in such a way that other currently loaded copies of program GSR do not overshoot it. Secondly, since the original source of the current subprogram must be preserved until the windup routine, program PP4, has listed it and since this is normally done by keeping it in tact in the circular INPUT buffer, special action (which is described in 4.3) must be taken when a source subprogram is detected which is too large to be entirely held in the buffer. This condition will be detected by one copy of program GSR and must be communicated to the other copies, if any. Two possible solutions to these problems are discussed in Chapter 14 but have not been further investigated. Program GSR causes program DSA to be loaded as soon as an acceptable header statement is detected for the current subprogram. The latter program is also only loaded once per subprogram and like program GSR, remains loaded until its task is completed. Only a single copy

of program DSA is loaded and runs concurrently with program GSR. There are two reasons for this. Firstly, since there is only one copy of program GSR, multiple copies of program DSA would extract statements from their communication buffer faster than program GSR could make insertions; thus, forcing the programs to idle or to drop themselves after arranging for their subsequent recall. Secondly, if multiple copies of program DSA were permitted, all name table accesses would have to be made via the central memory table access request mechanism (see 5.1.0) which would increase table access overhead. This overhead would be further increased by clashes for access to the communication buffer, the error table and even the name table itself.

Program PCR is brought to execution by program GSR after every eighth insertion into a processor communication buffer as well as for the last time for the current source subprogram, just before program GSR drops itself.

Program CTM will be requested by the first program to detect that the main area of one or more central memory tables is full. The requesting program sets a flag (in central memory) which all other programs periodically check. When this flag is found to be set, all programs will drop themselves to allow program CTM exclusive access to the entire central memory table area. When program CTM has successfully completed central memory table expansion, it requests the monitor to reload all programs that were active at the time the table became full. Chapter 11 contains a full description of the central memory table expansion procedure.

The windup routine, PP4, is the last program to be executed for the current source subprogram. Its tasks are those which cannot be undertaken until all statements have been processed. For example, since the order in which source statements are processed is indeterminate, it is not possible to check the validity of the nesting of DO statements until all statements have been processed. Therefore, when the syntactic analysis of a DO statement has been completed by program MSA, an entry is generated in the DO table (see 5.1.4). When eventually all DO statements are processed in this way and information about all defined statement numbers has been entered in the statement label table, program PP4 will be able to determine DO nesting validity by scanning the DO table.

4.3 Large Source Subprograms

The listing of the current subprogram is produced at the end of the compilation process by program PP4. This is so that an error code can be printed on a line immediately following the source line containing the error with the full text of the corresponding error message and a line number reference to every occurrence of that error produced at the end of the listing. This is achieved by holding intact the entire source of the subprogram in the INPUT buffer throughout the compilation process. However, there arises the problem that the source subprogram may be too large to fit into the INPUT buffer. In this case, the beginning of the subprogram in this circular buffer must be overwritten by further source before the compilation of the subprogram can be completed and its

listing produced.

The problem was resolved in the following way. At the beginning of processing of each subprogram, the source line number of the header statement, relative to the start of the source file, is saved in central memory. If, after program CIO has completed filling the INPUT buffer, the end of the source file has been detected by program CIO, the buffer cannot contain what will be termed a large subprogram. Otherwise a mechanism is initiated which will set a flag as soon as the buffer contains less than 300_8 words. This ensures that the flag will be set as soon as it is possible that the next source statement is not wholly contained in the buffer, since the largest possible statement is 20 cards × 9 buffer words per card, that is, 180_{10} or 264_8 words long, excluding interspersed comment cards. Clearly, if the flag is not set before the end of the current subprogram is detected, then the subprogram is not large. The flag is checked before the extraction of each source statement is commenced and if set, causes program GSR to request system program CIO to load the next buffer full of source. Program GSR then requests to be reloaded by monitor after a suitable delay for input/output completion and drops itself. When the buffer has been refilled and program GSR reloaded, the above procedure continues until eventually the end of this large subprogram is found. Before program GSR finally drops itself, it requests program CIO to backspace over the entire source file.

If, after loading, program PP4 detects that the current subprogram is large, it immediately requests program CIO to insert the first buffer full into the INPUT buffer from what is now the start of the source file. However, program PP4 continues executing in the normal way until it is about to produce the listing. It now skips over all source lines preceding the header statement of the current large subprogram and then commences producing the listing as usual.

If, during the listing of a large subprogram, program PP4 detects that the INPUT buffer is less than one-third full, it requests program CIO to again fill the buffer but continues to produce the listing. Thus, while program CIO is inserting words into the buffer, program PP4 is extracting from it, always checking that it is not catching up to the other program. Should this happen, program PP4 dumps its data areas into central memory, recalls itself with a suitable delay and then drops itself. It will subsequently resume execution, retrieve its dumped data areas from central memory and continue listing the large source subprogram. The procedure is repeated until the listing is completed.

5. CENTRAL MEMORY COMPILATION SYSTEM TABLES

5.0 Introduction

The central memory holds all information which is required to be accessible to more than one peripheral processor program. The greater part of this information is stored in table form whilst the remainder, as well as the flags required to gain access to these tables, is grouped together under the title "miscellaneous information". Since some of the tables are variable-length, central memory as a whole is therefore variable-length. The minimum and initial memory allocation is 4K words but the memory itself has a fixed-length segment which precedes the variable-length segment. The former segment comprises the miscellaneous information (200_8 words), the INPUT buffer (3000_8 words) and three processor communication buffers (700_8 words each). The variable-length segment comprises the DO table (200_8 words initially), error table (100_8 words), name table (400_8 words) and statement label table (200_8 words). When required, an OUTPUT buffer occupies the high end of core (see note X of 12.1).

5.1 Miscellaneous Information

The information which comprises this area will be briefly described in the following sections.

5.1.0 The Access Switches

Because the contents of a given table could become corrupted by the simultaneous access by two or more programs, these tables must be protected from such

multiaccessing. Therefore each such table is allocated an access switch in this area. When the access switch corresponding to a certain table is zero, that table is not currently being accessed. Otherwise the value of the access switch represents the processor number of the program currently accessing that table. When a program requires access to a table, it reads in the access switch for that table and if non-zero, the program executes a long idle loop and then again tests the state of the access switch. When the access switch is zero, the program enters its processor number (see 5.1.1) into the access switch and writes it back into central memory. The time required to read the access switch into peripheral memory, set it and then write it back is twenty-one microseconds. The program then enters an idle loop of thirty-five microseconds duration to ensure that if any other program is simultaneously writing into the same access switch, sufficient time will have elapsed to resolve a possible conflict. The program then re-reads the access switch and compares the contents with its own processor number. If there is agreement, then this program has successfully gained access to the required table. Otherwise the program enters a long idle loop and subsequently retries the access request operation. When the program has finished with a table, it clears the corresponding access switch, thus making it available to some other program. Access switches are provided for the

three processor communication buffers, the DO table, the error table, the name table, the statement label table, the processor status table and the central memory table full flag word. The processor count word is accessible through the processor status table access switch.

The access switch concept used here is in some respects similar to Dijkstra's binary semaphore [5,6]. However, the effective use of this semaphore requires the existence of an indivisible operation "swap" which causes the semaphore and a private variable (in peripheral memory) to exchange their values. The "test and set" instruction of the IBM 360/67 and UNIVAC 1108 systems (15) provides this type of operation. Two other hardware implementations of Dijkstra's "swap" operation are described by Hill (12) and Zelkowitz (20). However, since the CDC 6000 series hardware does not provide this facility, it is not possible to implement Dijkstra's binary semaphore directly. As it is possible that two or more peripheral processors may be trying to set the same access switch during the twenty-one microsecond period in which the first of these processors is setting the switch, it is necessary to provide some way to resolve the resulting conflict. The latter is achieved by providing each processor with a unique number (that is, the processor number), so that after sufficient time has elapsed the access switch will indicate which processor has successfully

gained access. The thirty-five microsecond delay includes a fourteen microsecond margin to allow for the worst possible case of central memory hardware access clash.

The accessing mechanism adopted avoids the problem of the deadly embrace [5,6] or deadlock (17), since there are only two situations in which a given processor requires access to more than one table simultaneously. One of these situations is peculiar to program GSR. If, during a scan through the statement label table, the program detects a doubly defined statement number, access to the error table will be required. The second situation may arise if a program detects that a table (that is, the DO, error, name or statement label table) is full. In this case, the program will require access to the central memory table full flag word. Shoshani and Bernstein (17) describe an involved method in which more than one processor could be simultaneously accessing the same table in such a way that only one of these processors is accessing a given entry in the table at any point in time. However, the implementation of this method in the context of this thesis presents numerous problems due to the unavailability of Dijkstra's indivisible "swap" operation. It would mean that each entry in each table would have to be expanded to accommodate

an access switch of the type already discussed. This would have the effect of greatly increasing the amount of central memory store as well as the table search time for a given processor. The advantage of the resulting multiaccessibility is further reduced by the fact that in the present work the total data is spread across seven tables. This implies that the likelihood of multiaccess to a given table being required is minimal as compared to its likelihood in the data base organisation of Shoshani and Bernstein.

5.1.1 The Processor Status Table

This table has ten entries and each non-empty entry represents a program which the monitor has been requested to load into an available peripheral processor. If no processor is available, monitor will save the request and cause the requested program to be loaded as soon as there is an available processor (see 3.3). On the other hand, a program may appear in an entry of the table but not be loaded and running because it is waiting for a central memory table move operation (performed by program CTM) to come to completion.

The first byte of each entry represents the program running at that entry point. The second byte indicates the status of that program, that is, whether or not the program is loaded and running and if not, whether it is waiting for central memory table move completion. The remaining bytes of each

entry are not used. The table entry number is used as the processor number of the program at that entry point. Every program clears its processor status table entry immediately before dropping itself. The processor count word is treated as part of the processor status table since it is accessed via the same access switch. The first byte of this word indicates whether or not program DSA is loaded while the second and third bytes respectively specify the number of copies of programs FEA and MSA that are running.

5.1.2 The Processor Communication Buffer Pointers

There are three processor communication buffers - one each for programs DSA, FEA and MSA. Program GSR inserts entries into these buffers and the appropriate program extracts these entries for processing. Each of these communication buffers consists of two linked lists either of which, but not both, may be empty. The free-list consists of entries into which strings of information can be inserted, whereas the main-list comprises entries already containing such strings (see 5.2). Each buffer has a set of pointers; they are a pointer to the first entry in the free-list, a pointer to the first entry in the main-list, a pointer to the last entry in the main-list, the number of main-list entries in the buffer and the start address of the buffer.

5.1.3 The INPUT Buffer Parameters

The INPUT buffer is accessible to both programs GSR and PP4. Therefore information about the status of the buffer must be available to both programs. The following parameters are stored: the end of subprogram flag, the end of file flag, the executable statement flag and the line number of the start of the current subprogram. The executable statement flag is checked by program DSA to determine whether program GSR has processed all the declarative statements and inserted them into its communication buffer. The line number of the start of the current subprogram is used by program PP4 in order to locate the beginning of all large subprograms for listing purposes.

5.1.4 The DO Table Pointers

Each entry in this table contains location information about a DO statement as well as pointers to the statement label table entry corresponding to the DO loop terminator and to the name table entry corresponding to the DO loop control variable. The table is generated by program MSA and finally processed by program PP4 in order to check the validity of the nesting of DO loops and of the type of DO terminating statement. Program PP4 also checks that nested DO loops do not use the same control variable. The two pointers stored in this area for the DO table are the absolute address of the next free entry and the last word address of the table.

5.1.5 The Error Table Pointers

Every syntax error detected during the compilation process causes an entry in this table. Each entry comprises the source line number at which the error occurred as well as a numeric error code (see appendix 3). Each word in the table holds two entries. The pointers are the current size of the table in entries, the absolute address of the next free entry and the total number of errors.

5.1.6 The Name Table Pointers

Every name, except COMMON block names (see 5.1.13), used in the subprogram are stored in this table. The pointers are the first word address of the table, the current size of the table and the relative address of the next free entry in the table.

5.1.7 The Statement Label Table Pointers

This table contains an entry for every statement number defined and/or referenced in the subprogram. The pointers are the same as for the name table.

5.1.8 The Program PP4 Parameters

These parameters are generated by program GSR and eventually accepted by program PP4. Two examples of these parameters are the INPUT buffer address of the first source statement of the current source subprogram and a flag to indicate whether or not the current subprogram is large.

5.1.9 The File Environment Tables

Each of the two files INPUT and OUTPUT require their own five word file environment table. The format and significance of the file environment table is described in 3.4.

5.1.10 The Error Message Table

The text of the error messages printed by program PP4 during its listing of the source subprogram is too large to be stored in PP4 itself. Each error message is represented by a variable-length string of display characters. These strings are packed together in table form and access is via an index table which provides a pointer to the beginning of each string and an indication of its length. This index table is also stored in central memory.

5.1.11 The Central Memory Table Full Flag Word

When a program detects that a central memory table main area is full, it sets the appropriate bit in this word and then requests monitor to load program CTM, if it has not already been requested, to perform the central memory table move operation (see also 11.0). This flag word provides a bit to indicate full status for each of the DO, error, name and statement label tables. In addition, the increment size for each table is also stored in this word. The tables are only incremented by the specified amount if they are full, otherwise they are incremented by one quarter of the specified number of words (see also note III of 11.1).

5.1.12 The Subprogram Name and Header Flags

The name of the current source subprogram is stored here as well as flags to indicate whether a header card has been detected, whether the current subprogram is BLOCK DATA and whether or not the last statement was a transfer of control. The latter is used by program GSR to determine whether to flag the "no path to this statement" error if the current statement does not have a statement number defined in its statement label field. This flag is only updated if program GSR must for any reason drop itself before detecting the end of the current subprogram.

5.1.13 The COMMON Block Name Table

This table is only accessed by program DSA and so does not require an access switch. Each valid COMMON block name is stored in the table together with a flag to indicate whether it has been used in the current source subprogram. This is to make it possible to detect duplicate definitions of a given COMMON block name in the current subprogram. At the end of processing the current subprogram, program PP4 clears all flags in this table but does not clear the rest of each entry. This is to allow the detection of the definition of more than sixty-one COMMON blocks, which causes an error to be flagged.

5.2 The Processor Communication Buffer

There are three such buffers, one for each of the programs DSA, FEA and MSA. They are used to pass information from program GSR to these other programs. Each buffer consists of two linked lists, called the main-list and the free-list, of variable-length entries called strings. Initially, the main-list is empty and the free-list consists of a single string which comprises the whole buffer. A string consists of a one central memory word header followed by the rest of the string which is variable-length. The information contained in the header includes the number of central memory words in the body of the string, a link to the next entry (if any) in the buffer, the statement type of the source statement which comprises the body of the string, the line number of the original source statement and one or more optional parameters, the format of which differs according to the statement type. For the header of a free-list entry, only the word count and link fields are relevant. The body of a main-list string will be either a complete or a partial FORTRAN statement from which all internal blanks have been removed and in which certain substrings have been replaced by six-bit codes. For example, ** is replaced by 60_8, all relational operators are replaced by 61_8, the logical operator .NOT. becomes 62_8, the logical operators .AND. and .OR. are represented as 63_8, logical constants are 64_8, all varieties of Hollerith text constants are 65_8 and the end of statement is represented by 00_8. Since 00_8 represents the end of the statement in

the body of the string, all following codes (if any) are ignored.

Each of the programs DSA, FEA and MSA attempt to extract an entry from their respective processor communication buffers at the beginning of their major loop. If the buffer is non-empty, the extracted string is stored in a special area at the high end of their peripheral memory. The area occupied in the buffer by the string just extracted is then converted into a free-list entry and this is then linked into the free-list chain. A check is next made to see if the last two entries in this free-list chain occupy adjacent areas in the buffer. If so, the two entries are merged into one in order to reduce fragmentation of the buffer. If program GSR is unable to find a sufficiently large free-list entry into which to write a string, it performs a garbage collection operation on that buffer during which the entries in the buffer are shuffled around so that all main-list entries will occupy the low buffer addresses and all free-list entries are merged together into one entry which then occupies the high buffer addresses.

5.3 The Name Table

This table is used to store information about all names, except COMMON block names, appearing in the current source subprogram. Each entry in this table represents a name and is linked into one of twenty-six possible lists according to the first letter of the name. Except for some statement function name entries, all entries comprise two

words, of which the first word is common to all entry types. This word contains the seven characters which comprise the name (unspecified characters being set to zero), a character count and a link to the next entry (if any) in the same list. The structure of the second word depends on the entry type, which is expressed as the first six bits of this word. The following sections give the details for each entry type.

5.3.1 Subroutine Name

This entry is defined by program FEA when the name is referenced for the first time in a CALL statement. The information stored is the type (that is, INTEGER, REAL etc. in case the name is subsequently also referenced as a function), whether the type was explicitly defined, the number of arguments specified in the first reference and a flag to indicate whether the name appeared in an EXTERNAL statement.

5.3.2 Function Name

This entry contains the same information as 5.3.1 above.

5.3.3 Subroutine Subprogram (Header) Name

No additional information is stored. The purpose of this entry is to allow program FEA to ensure that no recursive calls are attempted.

5.3.4 Function Subprogram (Header) Name

This entry is similar to that in 5.3.3 above and exists as a separate entry because this name may be referenced as a variable name whereas the subroutine subprogram name may not.

5.3.5 Variable Name

This entry specifies the type of the variable and whether it was explicitly defined. It also indicates whether the variable was equivalenced and if so, provides a pointer to another variable name in the same equivalence group. There is a flag to indicate whether the name was ever actually used as a variable, whether it is subscripted and if so, the number of subscripts, whether the array was defined with variable subscripts and the number of elements in the array. There is also a flag to indicate whether the name was defined in labelled COMMON or blank COMMON and a pointer to the COMMON block name table entry to which the variable belongs. The fact that the name appears in a formal parameter list or in the variable list of a DATA statement is also indicated by flags.

5.3.6 Statement Function Name

If the statement function has more than eight arguments, this entry is more than two words long; one extra word is added for each twenty (or part thereof) arguments in excess of eight. The entry stores the type of the function, the number of words in excess of two in this entry and a link to the next group of words (if any). There is also a bit to indicate whether the function type was explicitly defined. The entry also specifies the number of arguments defined as well as a list giving the type of each argument starting with the first.

On detecting a statement function reference, program FEA uses the information in this entry to check the validity of each actual argument (see also 11.0).

5.3.7 Namelist Name

This entry does not define any further information. It is merely used to specify that the name has been used as a NAMELIST name since such names must be unique.

5.4 The Statement Label Table

This table is used to store information about all statement numbers defined and/or referenced. The table consists of sixty-four linked lists and an entry, which consists of one central memory word, is attached to a particular list according to a six bit hash code (see note XXXVI of 6.1) generated from the non-space characters which make up the statement number.

Each entry stores not only the five digits (unused digit positions are set to binary zero) which make up the statement number, but also a flag to indicate whether the statement number was defined. For undefined labels, a bit indicates whether the label reference occurred in a DO statement, whereas for defined labels, the same bit indicates whether or not the definition statement is an allowable DO terminator. Other bits indicate whether the statement number was defined as a FORMAT statement number or referenced as such. A further bit indicates whether the statement number was referenced by a transfer of control statement. The entry also holds the source line number of

the defining statement and a pointer to the next entry in the same list.

Program GSR inserts an entry into this table whenever the statement label field of the current statement contains a valid statement number which has not been previously referenced. Program MSA inserts an entry into this table, if it does not already exist, whenever a reference to a statement label is detected.

6. PROGRAM GSR - GENERAL STATEMENT RECOGNIZER

6.0 Introduction

This program is loaded by the system for the first source subprogram and by program PP4 for each subsequent subprogram. Its primary task is to extract source statements from the INPUT buffer, recognize the type of each statement and pass it on via the appropriate processor communication buffer to whichever program is to further process it. It does, however, syntax check CONTINUE and RETURN statements itself.

As a by-product of this primary task, it creates entries in the statement label table for statement numbers defined in the statement label field. It breaks up multiple statement lines (see note IV of 6.1) into separate statements and splits each IF statement into an expression part, which is to be processed further by program FEA and either a statement number list part which is passed on to program MSA or a statement part which is thereafter treated as a free-standing statement. The program also recognizes the end of the current subprogram and the occurrence of a large subprogram (see 4.3).

During the lexical analysis stage, program GSR checks for the presence of logical and exponential operators within expressions and if found, flags the statements concerned for special action by program FEA (see 9.0).

The program also carries out maintenance on the three processor communication buffers whenever the free-list

entries become unusably small due to fragmentation. It also checks that there is a logic path to every statement by ensuring that each statement following a transfer of control statement has a statement number defined in its statement label field.

Program GSR is responsible both directly and indirectly for the initiation of the other programs in the system. Thus, when the program detects a recognizable header statement, it requests monitor to load program DSA to further process this header as well as any subsequent declaratives. Further, program GSR periodically requests monitor to load program PCR so that the latter may initiate the loading of programs FEA and MSA, as necessary. On detecting the end of the current source subprogram, program GSR drops itself after requesting monitor to load program PCR for the last time for the current subprogram (see 7.0).

6.1 Explanatory Notes to the Flow-Chart

Although not explicitly stated in the accompanying flow-chart, references to all tables are made only after access to the required table has been successfully requested, using the procedure described in 5.1.0.

The following notes refer to the flow-chart:

I. This is done for the case that this is not the first call to this program for the current source subprogram (see 5.1.12).

II. Only the first call to this program for the first source subprogram calls program CIO on entry.

III. This is the initiation of the mechanism referred to in 4.3.

IV. A multiple statement line consists of any number of statements, separated by the character $, on the one line.

V. If this is true then the current subprogram is large. In this case, program CIO is requested to refill the buffer with more source and meanwhile program GSR drops itself.

VI. This is the beginning of the lexical analysis of the current source statement. During this analysis, a count is kept of the number of occurrences of equals as well as the number of times that the character comma occurs outside parentheses. These counts are used to distinguish between assignment statements, DO statements and those statements which start with a keyword.

VII. These types of special codes are described in 5.2.

VIII. These pointers will be required when the lexical analysis of the current multiple statement is resumed. This resumption will occur when the current statement has been recognized and passed on to the appropriate program for further processing.

IX. This is the beginning of the statement recognition phase. If the equals count (see note VI above) is zero, recognition is attempted by keyword. If the comma count is zero, the statement is recognized

as an assignment statement (or possibly a statement function definition statement, see 9.0). Otherwise if the comma count is one or two, the equals count is one and the first two characters are DO, the statement is recognized as a DO statement. See also note X below.

X. If the statement starts with IF, it may be an IF statement of the type

IF (ℓ) S

where ℓ is a logical expression

S is an assignment statement.

XI. If a header card has not already been detected, an error is flagged for the current statement, which is thereafter ignored.

XII. If this is true, the current statement, which is a declarative, is positioned after at least one executable statement.

XIII. The "logical operator flag" is used by program FEA to determine which method is to be used to further process the current statement (see also note XI of 9.1). The "statement number flag" indicates whether a statement number was defined for the current statement. This is used by program FEA to check that a statement function definition statement does not have a statement label defined.

XIV. See 5.1.12 for the significance of the "last statement type" flag.

XV. All statements in a BLOCK DATA subprogram must be declaratives of the type processed by program DSA. Therefore if the current statement is not END then it is an error.

XVI. If a header statement has already been detected, then the current statement is invalid.

XVII. As soon as a header is recognized and loaded into the processor 1 communication buffer, program DSA may be requested to process it.

XVIII. A statement label must be defined for every CONTINUE statement.

XIX. These special IF statements are treated differently from other IF statements and when recognized are passed on to program MSA. See also note X of 10.1.

XX. The last open parenthesis of an IF statement signals the end of the expression part. The next character indicates whether the statement is not instead an assignment statement (or even a statement function definition statement). If the character is equals then the statement is of this type otherwise it is a true IF statement.

XXI. If this is true, then the statement is a one-branch IF and the expression part must be logical (see also note X above).

XXII. If this is true, then the statement is a two-branch IF and the expression part must be logical.

XXIII. If this is true, then the statement is a three-branch IF and the expression part must be arithmetic.

XXIV. The current statement must not be the second part of a one-branch IF.

XXV. The expression part has already been inserted into the processor 2 communication buffer for further processing by program FEA. Here, the statement number list part is being passed on to program MSA.

XXVI. For the significance of this flag, see note XXIV above.

XXVII. The statement part of this one-branch IF must now be recognized.

XXVIII. The END statement may optionally be followed by the name of the current source subprogram (see 5.1.12).

XXIX. A maximum of nineteen continuation cards are allowed.

XXX. Statement label fields of continuation cards must be blank.

XXXI. Just in case it was not previously set, as for example with a BLOCK DATA subprogram.

XXXII. Since BLOCK DATA subprograms may not validly contain executable statements, a call to program PCR is not required since both processor 2 and processor 3 communication buffers must be empty.

XXXIII. This flag is checked (see note V above) at the beginning of the major loop and, if set, causes program CIO to be requested to fill the buffer. The flag is cleared when program CIO has completed the operation.

XXXIV. If the beginning of the major loop is not reached between the setting of the flag for the first time and the total exhaustion of the INPUT buffer (see note XXXIII above), a fatal system error is flagged and the further compilation of the remainder of the current subprogram is abandoned. This is an extremely improbable event as it can only occur if the statement after the last complete statement in the buffer has more than twenty continuation cards (see note XXIX above), all of which are punched in column 72 (since trailing blanks are removed), and/or has a large number of comment cards imbedded within its continuation cards.

XXXV. A card with any of the characters C, asterisk, dollar or point punched in column 1 is regarded as a comment card.

XXXVI. The hash code generated is the middle-most six bits of the thirteen bit result of summing the display code character representations of the statement number taking two digits at a time. For this purpose, the statement number is taken left-justified with zero bit fill for unspecified digit positions.

XXXVII. Error 23 flagged if referenced by a DO statement and not defined as a valid DO terminator.

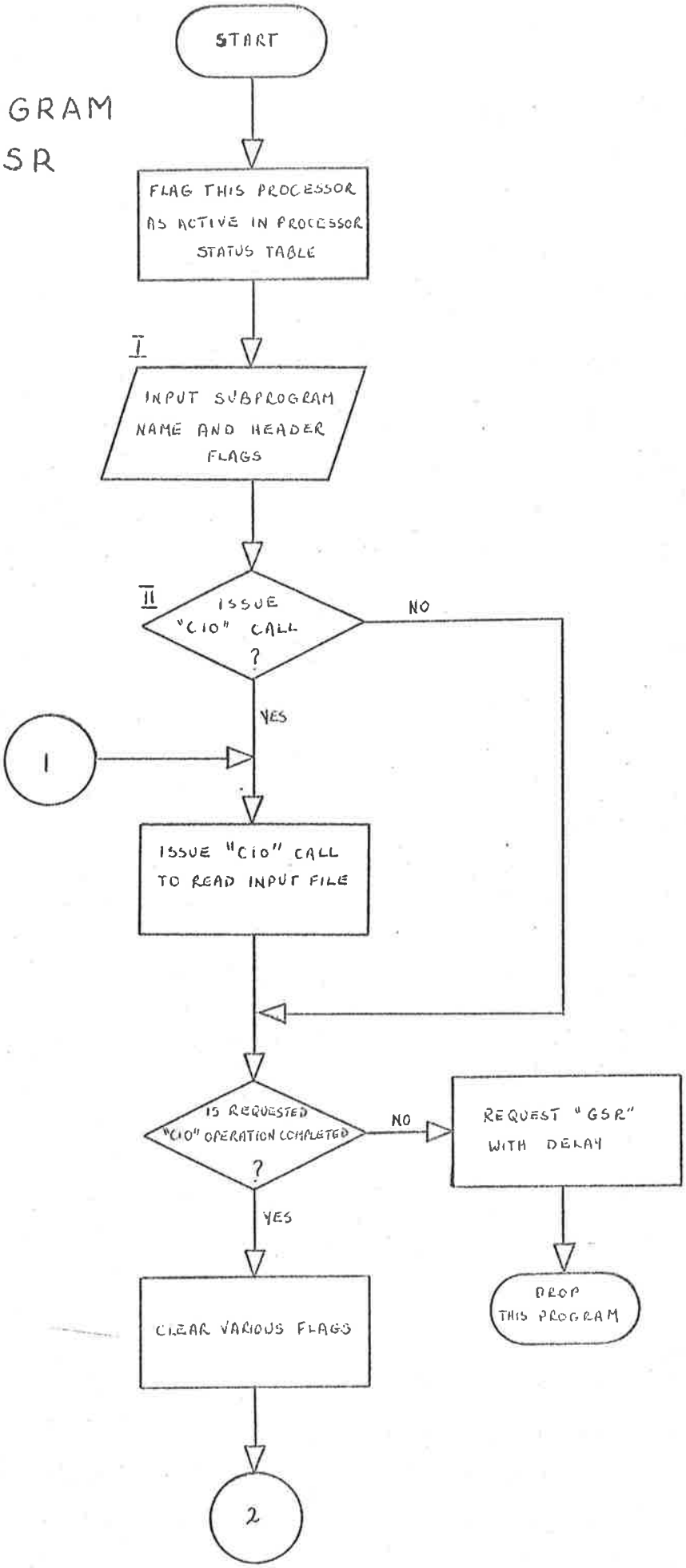
Error 14 flagged if referenced as a FORMAT statement number but not defined as such.

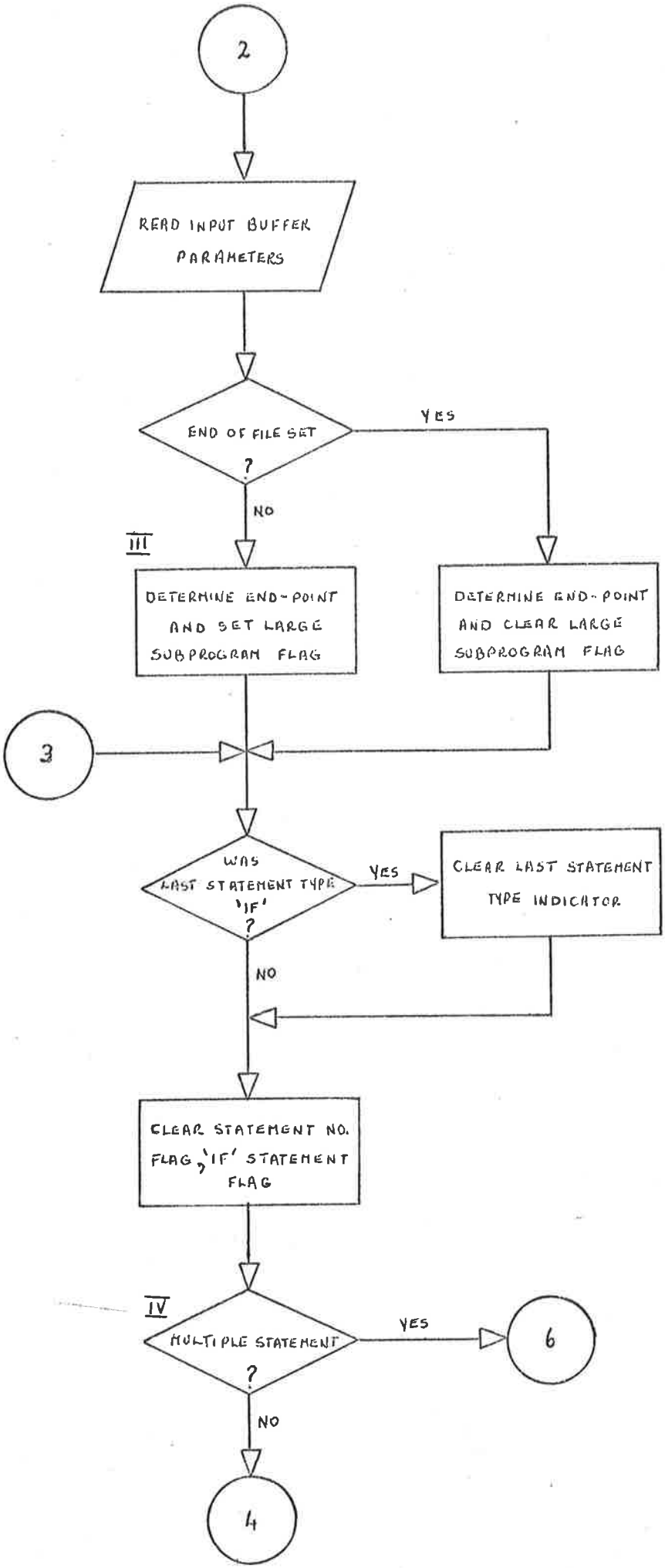
Error 41 flagged if referenced by a transfer-of-control statement but defined as a FORMAT statement number.

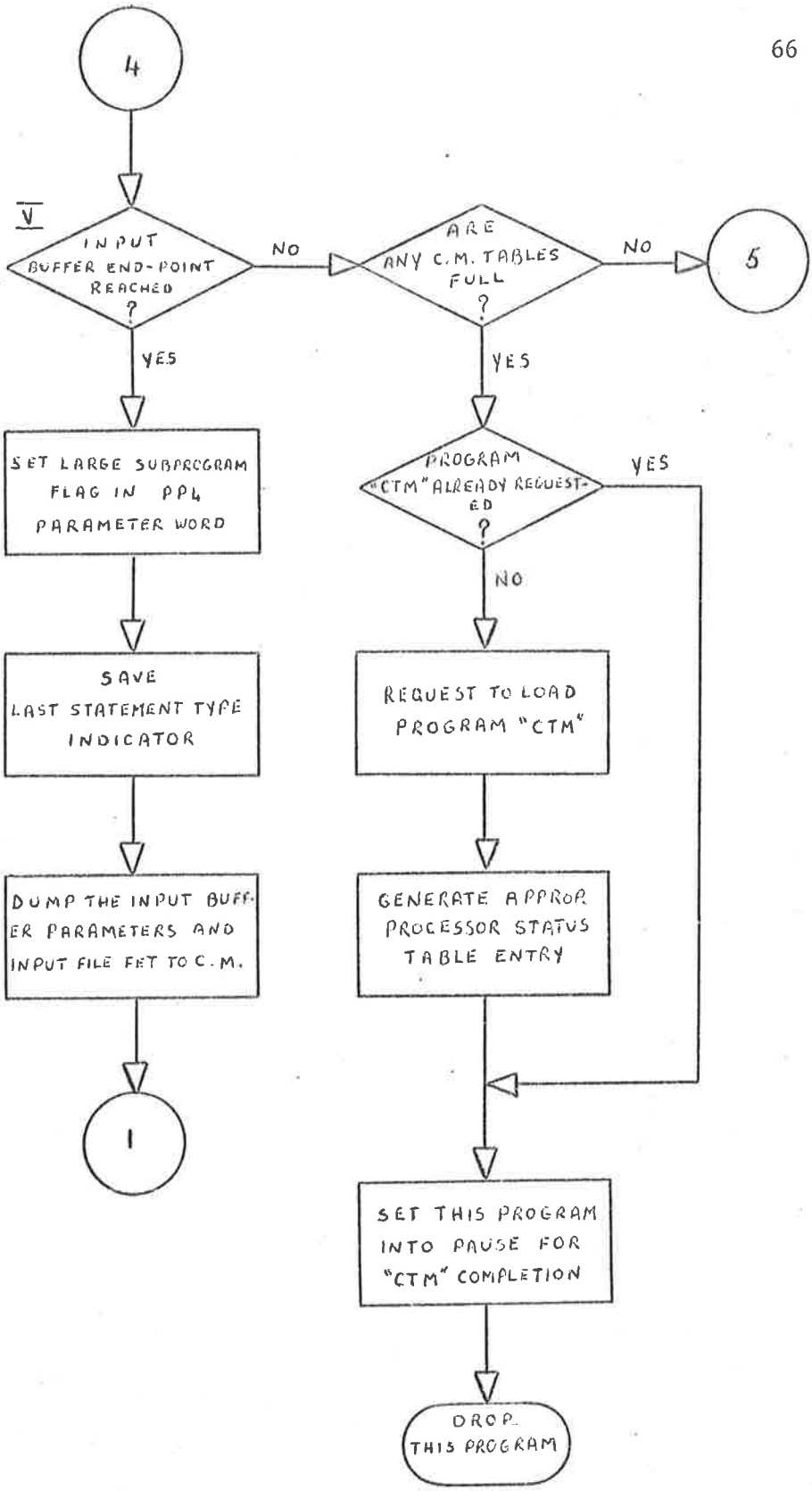
XXXVIII. Declarative statements must have blank statement label fields.

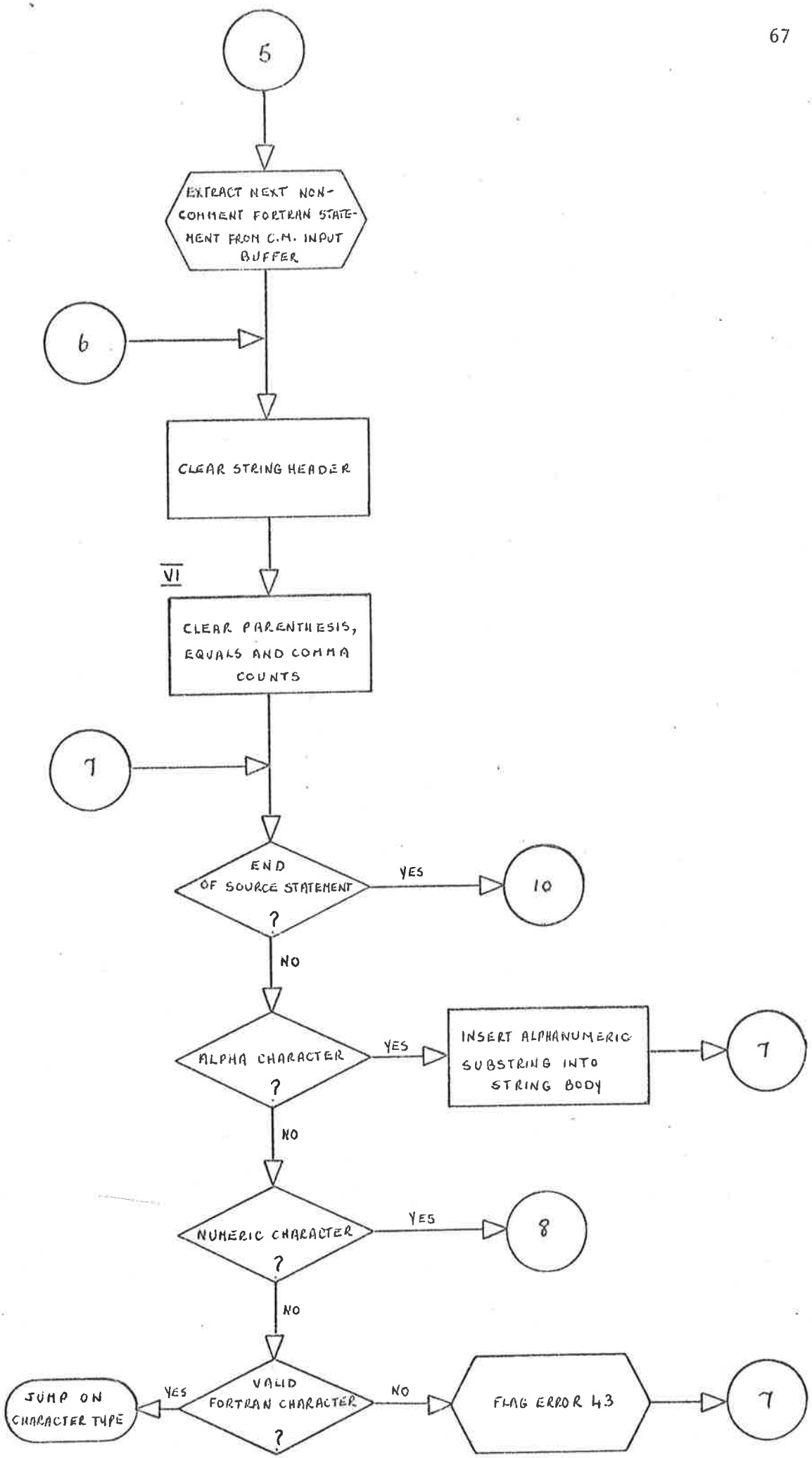
XXXIX. If, due to fragmentation (see note V of 8.1), none of the entries in the free-list is large enough to accommodate the current string, garbage collection is performed on the communcation buffer. When completed, the free-list will consist of a single entry representing the whole of the buffer free-space.

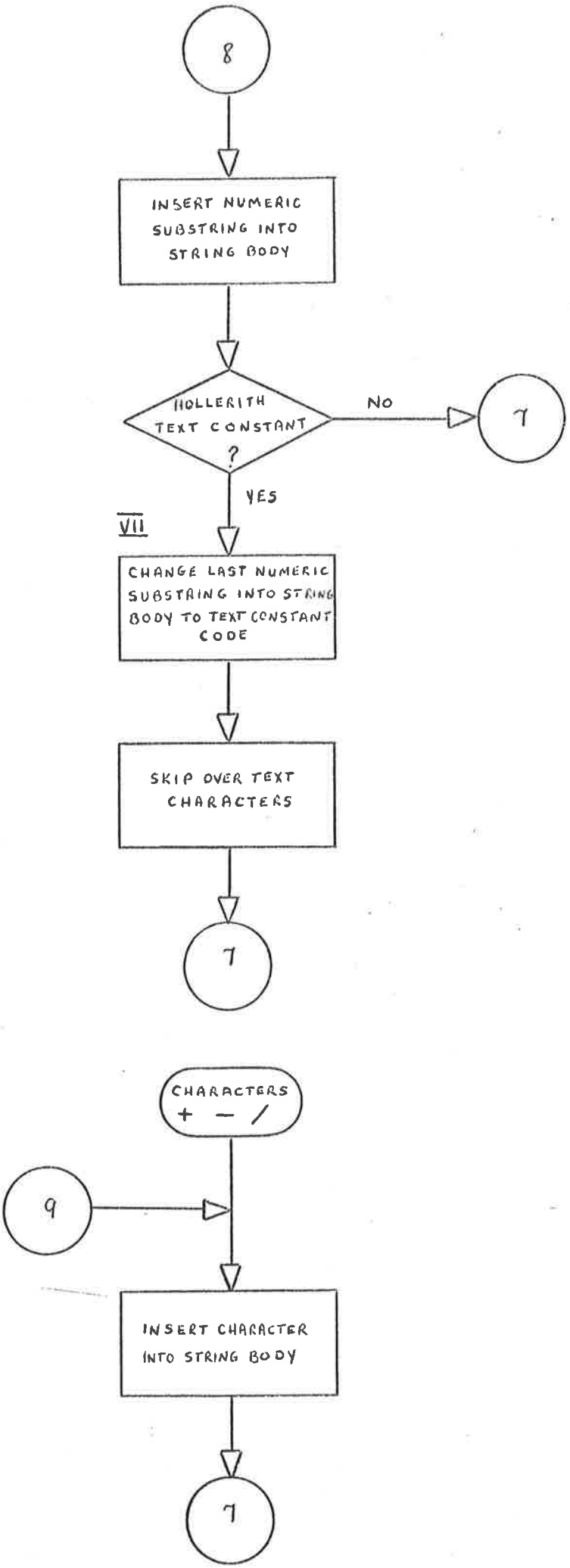
PROGRAM
GSR

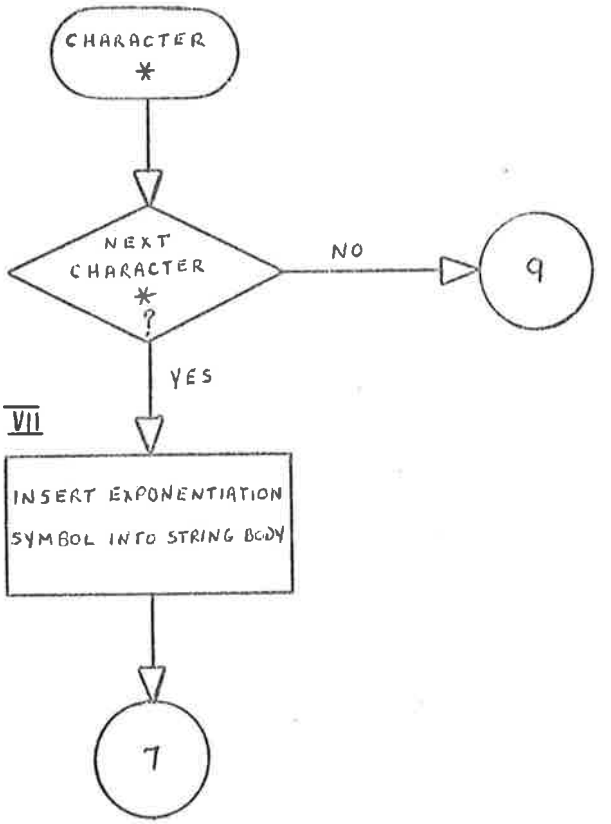

START
FLAG THIS PROCESSOR AS ACTIVE IN PROCESSOR STATUS TABLE
I
INPUT SUBPROGRAM NAME AND HEADER FLAGS
II
ISSUE "CIO" CALL ?
NO
YES
1
ISSUE "CIO" CALL TO READ INPUT FILE
IS REQUESTED "CIO" OPERATION COMPLETED ?
NO
REQUEST "GSR" WITH DELAY
YES
DROP THIS PROGRAM
CLEAR VARIOUS FLAGS
2

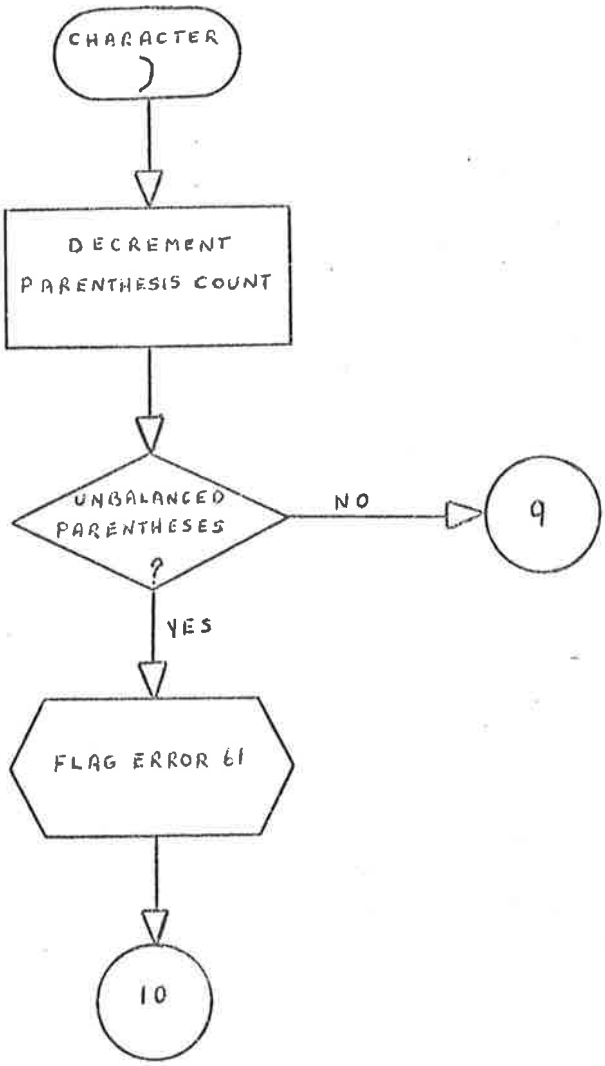

2
READ INPUT BUFFER PARAMETERS
END OF FILE SET ?
YES
NO
III
DETERMINE END-POINT AND SET LARGE SUBPROGRAM FLAG
DETERMINE END-POINT AND CLEAR LARGE SUBPROGRAM FLAG
3
WAS LAST STATEMENT TYPE 'IF' ?
YES
CLEAR LAST STATEMENT TYPE INDICATOR
NO
CLEAR STATEMENT NO. FLAG, 'IF' STATEMENT FLAG
IV
MULTIPLE STATEMENT ?
YES
6
NO
4

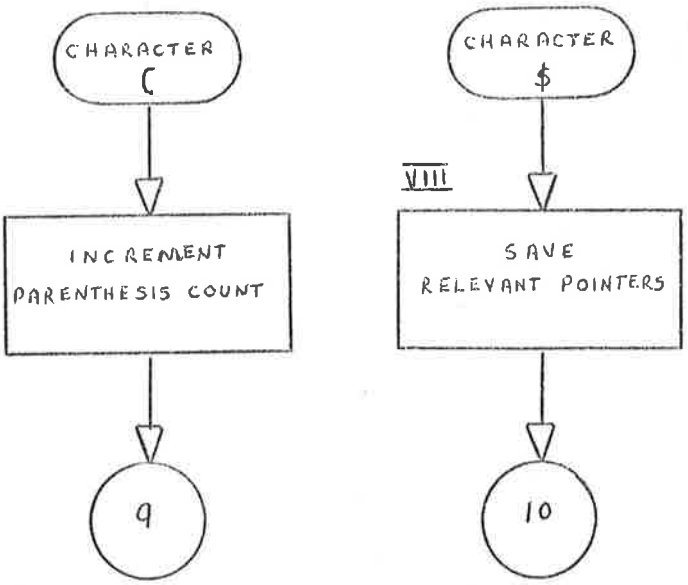

4
V
INPUT BUFFER END-POINT REACHED ?
NO
ARE ANY C.M. TABLES FULL ?
NO
5
YES
YES
SET LARGE SUBPROGRAM FLAG IN PP4 PARAMETER WORD
PROGRAM "CTM" ALREADY REQUESTED ?
YES
NO
SAVE LAST STATEMENT TYPE INDICATOR
REQUEST TO LOAD PROGRAM "CTM"
DUMP THE INPUT BUFFER PARAMETERS AND INPUT FILE FET TO C.M.
GENERATE APPROR PROCESSOR STATUS TABLE ENTRY
1
SET THIS PROGRAM INTO PAUSE FOR "CTM" COMPLETION
DROP THIS PROGRAM

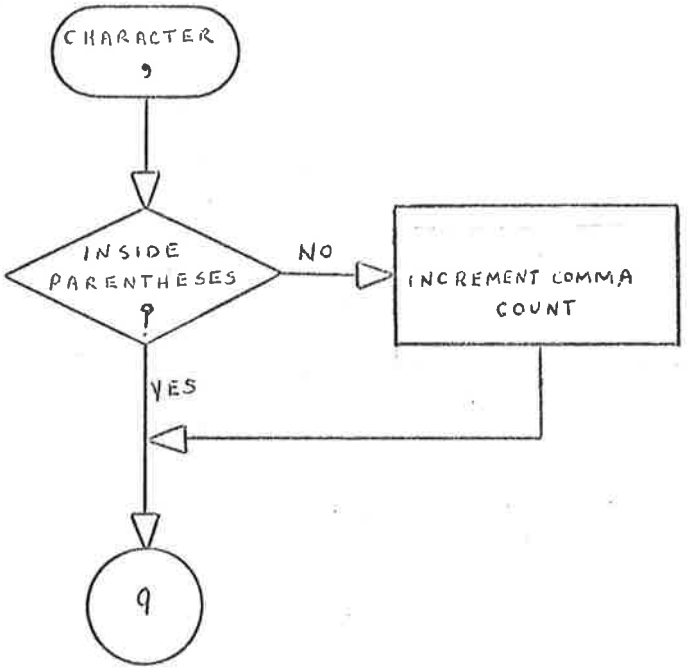

5
EXTRACT NEXT NON-COMMENT FORTRAN STATEMENT FROM C.M. INPUT BUFFER
6
CLEAR STRING HEADER
VI
CLEAR PARENTHESIS, EQUALS AND COMMA COUNTS
7
END OF SOURCE STATEMENT ?
YES
10
NO
ALPHA CHARACTER ?
YES
INSERT ALPHANUMERIC SUBSTRING INTO STRING BODY
7
NO
NUMERIC CHARACTER ?
YES
8
NO
VALID FORTRAN CHARACTER ?
YES
JUMP ON CHARACTER TYPE
NO
FLAG ERROR 43
7

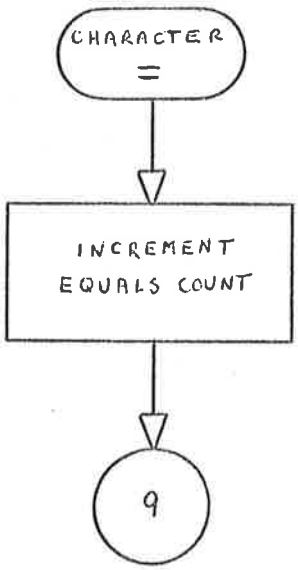

8
INSERT NUMERIC SUBSTRING INTO STRING BODY
HOLLERITH TEXT CONSTANT ?
NO
7
YES
VII
CHANGE LAST NUMERIC SUBSTRING INTO STRING BODY TO TEXT CONSTANT CODE
SKIP OVER TEXT CHARACTERS
7
CHARACTERS + − /
9
INSERT CHARACTER INTO STRING BODY
7

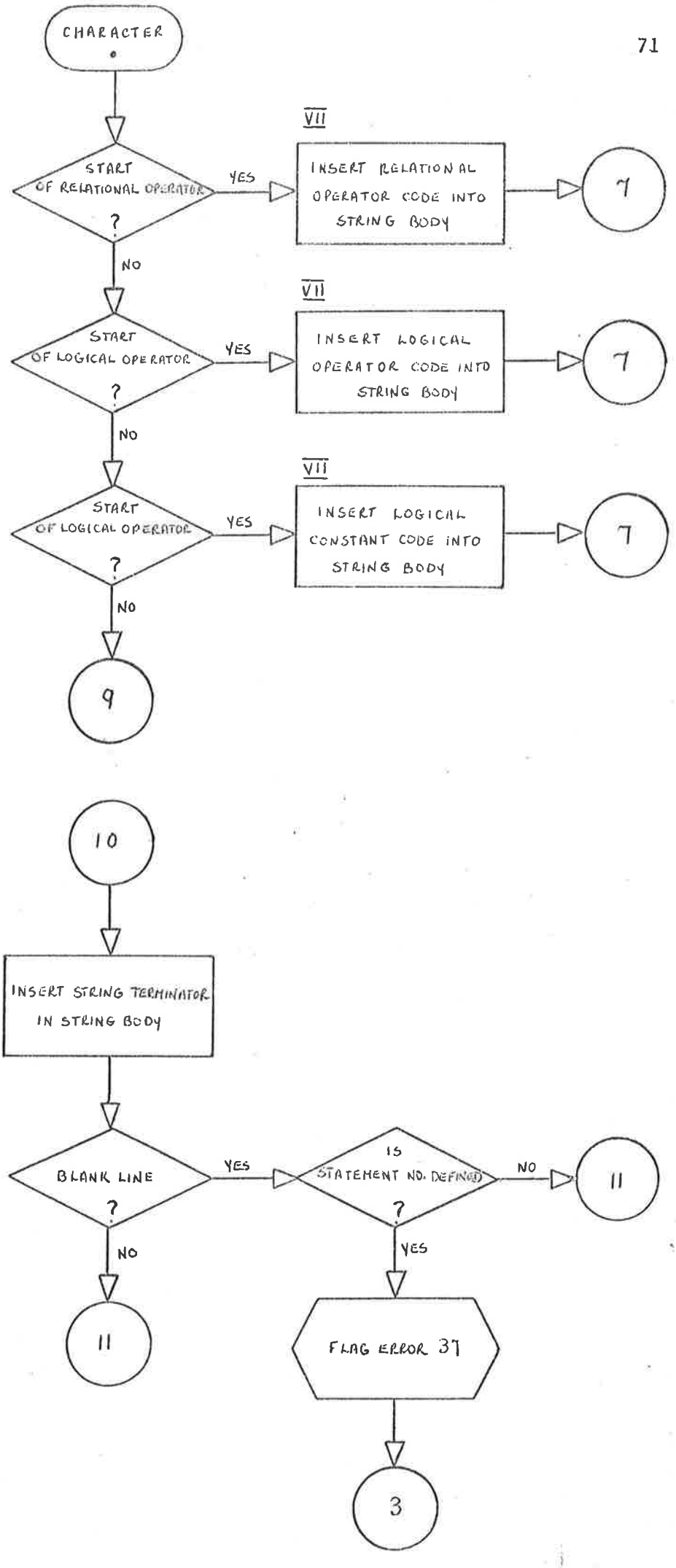

CHARACTER
*
NEXT CHARACTER
*
?
NO
9
YES
VII
INSERT EXPONENTIATION SYMBOL INTO STRING BODY
7

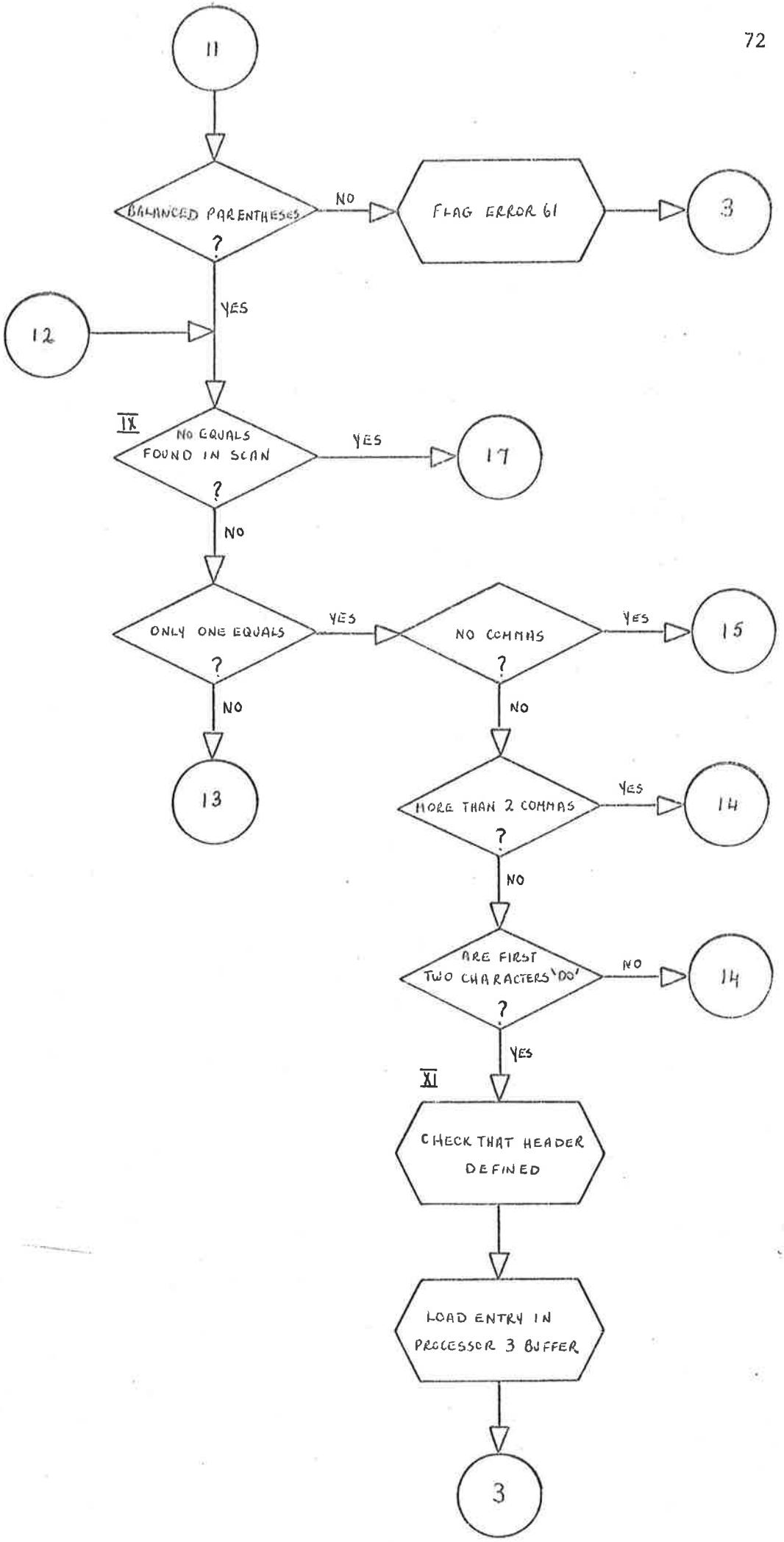

CHARACTER
)
DECREMENT PARENTHESIS COUNT
UNBALANCED PARENTHESES
?
NO
9
YES
FLAG ERROR 61
10

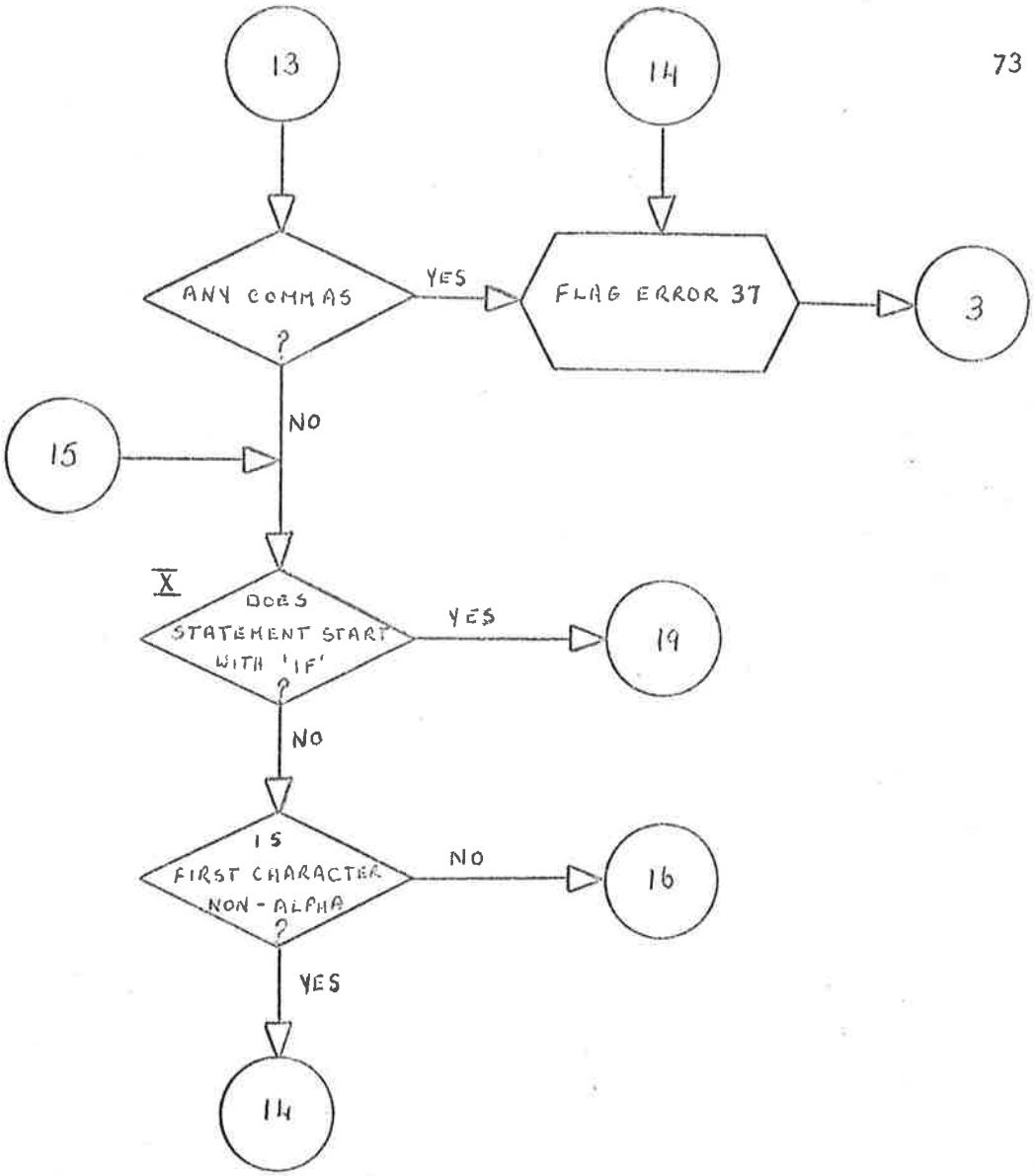

CHARACTER
(
INCREMENT
PARENTHESIS COUNT
9
CHARACTER
$
VIII
SAVE
RELEVANT POINTERS
10

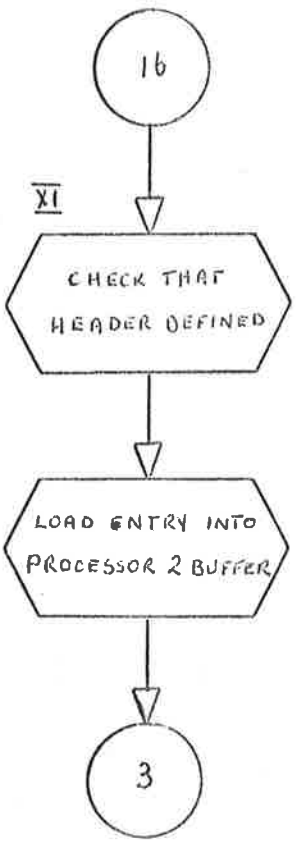

CHARACTER
,
INSIDE
PARENTHESES
?
NO
INCREMENT COMMA
COUNT
YES
9

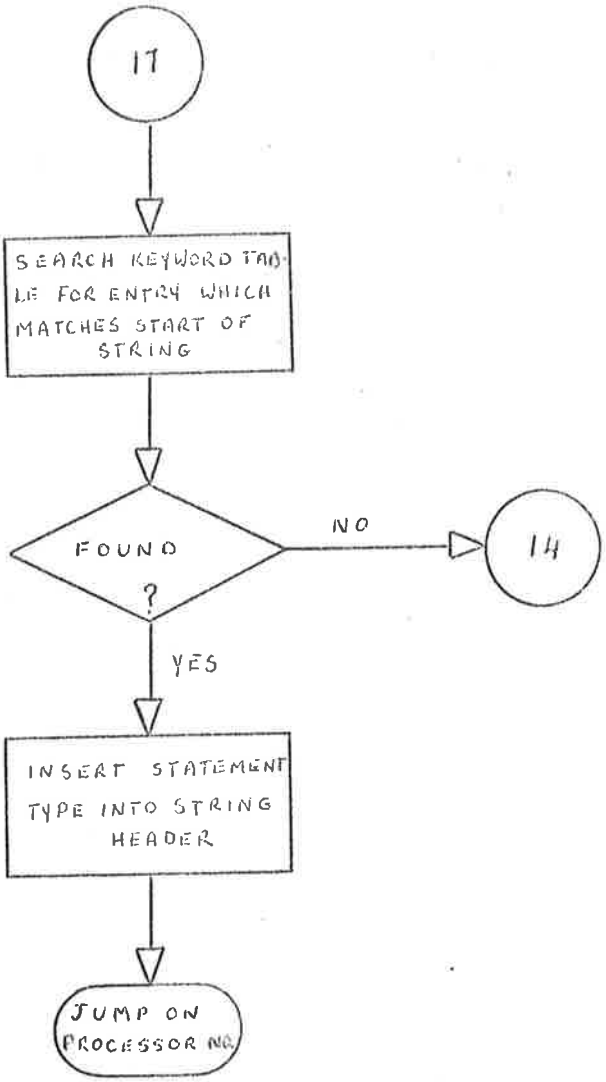

CHARACTER
=
INCREMENT
EQUALS COUNT
9

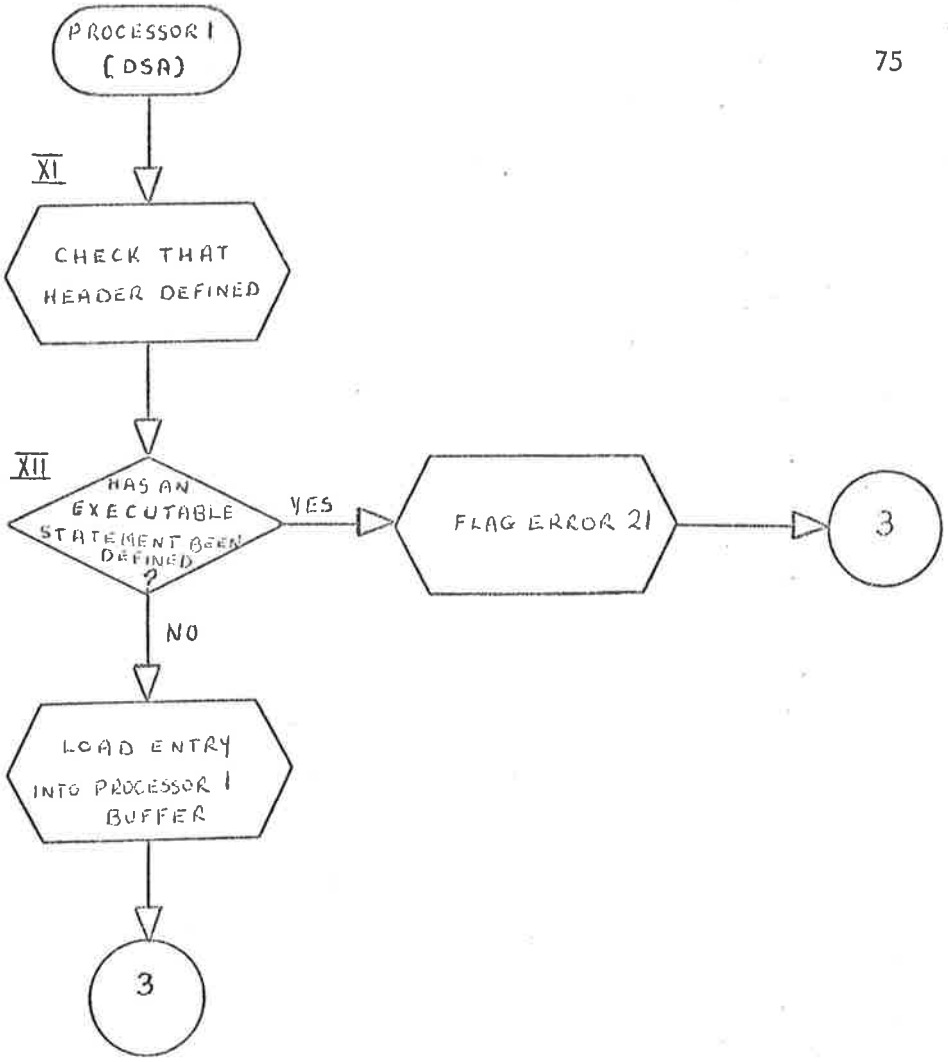

CHARACTER
START OF RELATIONAL OPERATOR ?
YES
VII
INSERT RELATIONAL OPERATOR CODE INTO STRING BODY
7
NO
START OF LOGICAL OPERATOR ?
YES
VII
INSERT LOGICAL OPERATOR CODE INTO STRING BODY
7
NO
START OF LOGICAL OPERATOR ?
YES
VII
INSERT LOGICAL CONSTANT CODE INTO STRING BODY
7
NO
9
10
INSERT STRING TERMINATOR IN STRING BODY
BLANK LINE ?
YES
IS STATEMENT NO. DEFINED ?
NO
11
NO
11
YES
FLAG ERROR 37
3

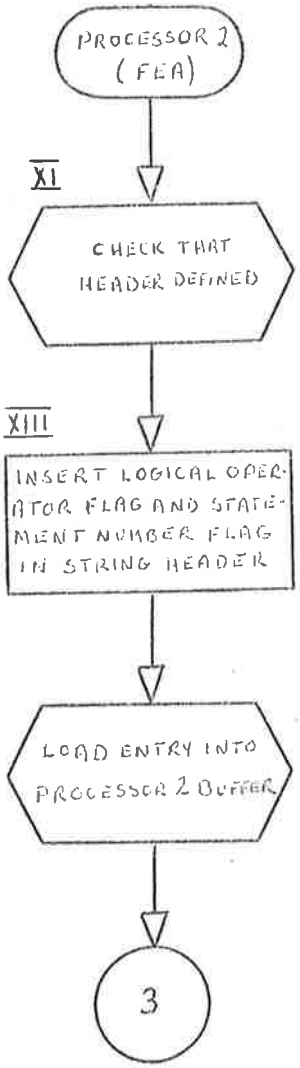

11
BALANCED PARENTHESES ?
NO
FLAG ERROR 61
3
YES
12
IX
NO EQUALS FOUND IN SCAN ?
YES
17
NO
ONLY ONE EQUALS ?
YES
NO COMMAS ?
YES
15
NO
13
NO
MORE THAN 2 COMMAS ?
YES
14
NO
ARE FIRST TWO CHARACTERS 'DO' ?
NO
14
YES
XI
CHECK THAT HEADER DEFINED
LOAD ENTRY IN PROCESSOR 3 BUFFER
3

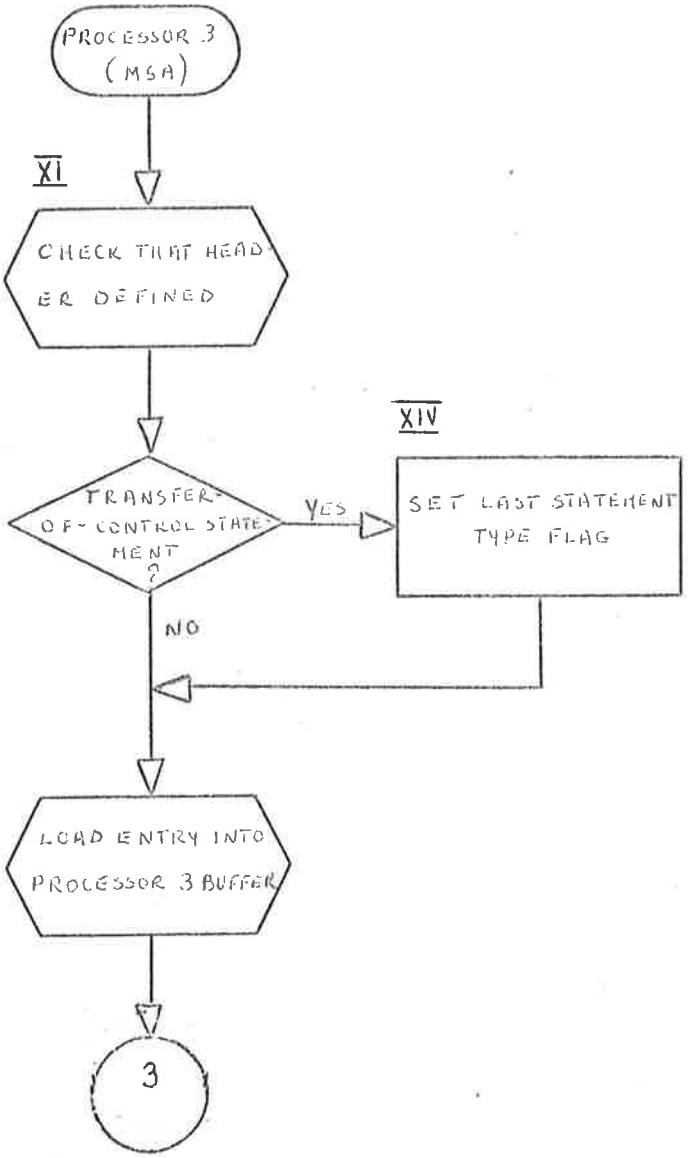

13
14
ANY COMMAS ?
YES
FLAG ERROR 37
3
NO
15
X
DOES STATEMENT START WITH 'IF' ?
YES
19
NO
IS FIRST CHARACTER NON-ALPHA ?
NO
16
YES
14

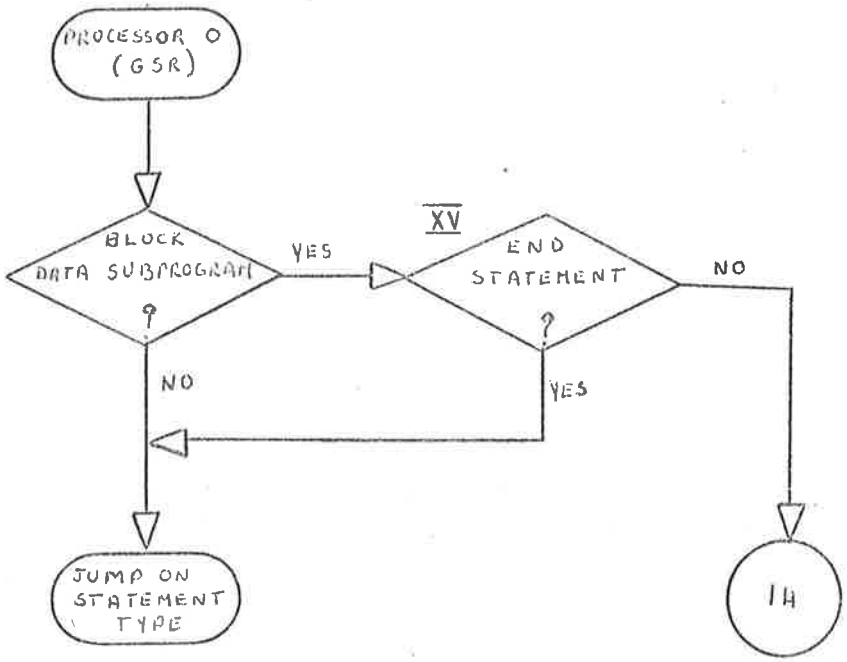

16
XI
CHECK THAT HEADER DEFINED
LOAD ENTRY INTO PROCESSOR 2 BUFFER
3

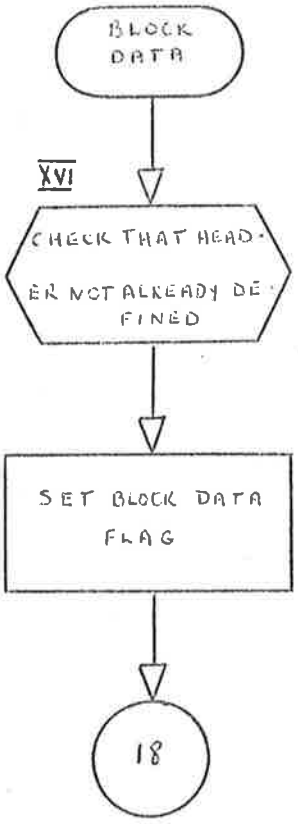

17
SEARCH KEYWORD TABLE FOR ENTRY WHICH MATCHES START OF STRING
FOUND ?
NO
14
YES
INSERT STATEMENT TYPE INTO STRING HEADER
JUMP ON PROCESSOR NO.

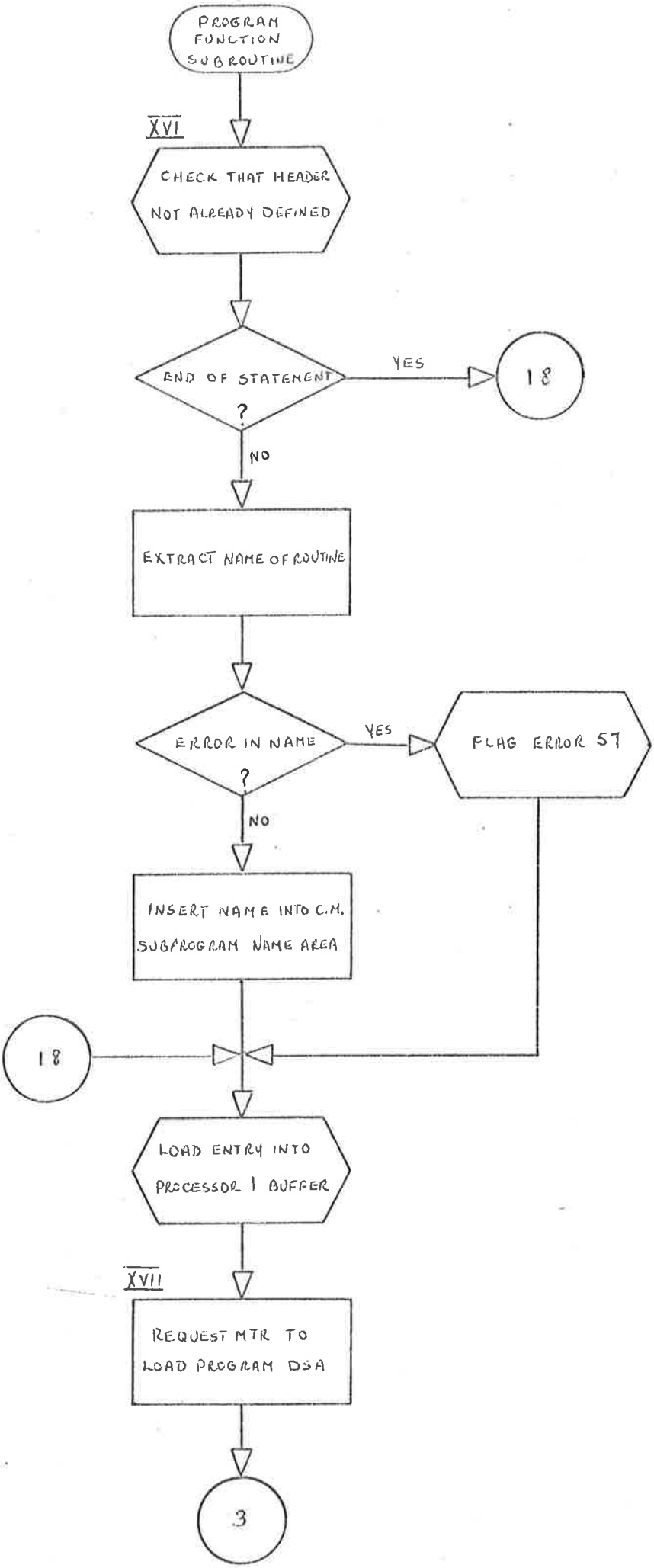

PROCESSOR 1
(DSA)
XI
CHECK THAT
HEADER DEFINED
XII
HAS AN
EXECUTABLE
STATEMENT BEEN
DEFINED
?
YES
FLAG ERROR 21
3
NO
LOAD ENTRY
INTO PROCESSOR 1
BUFFER
3

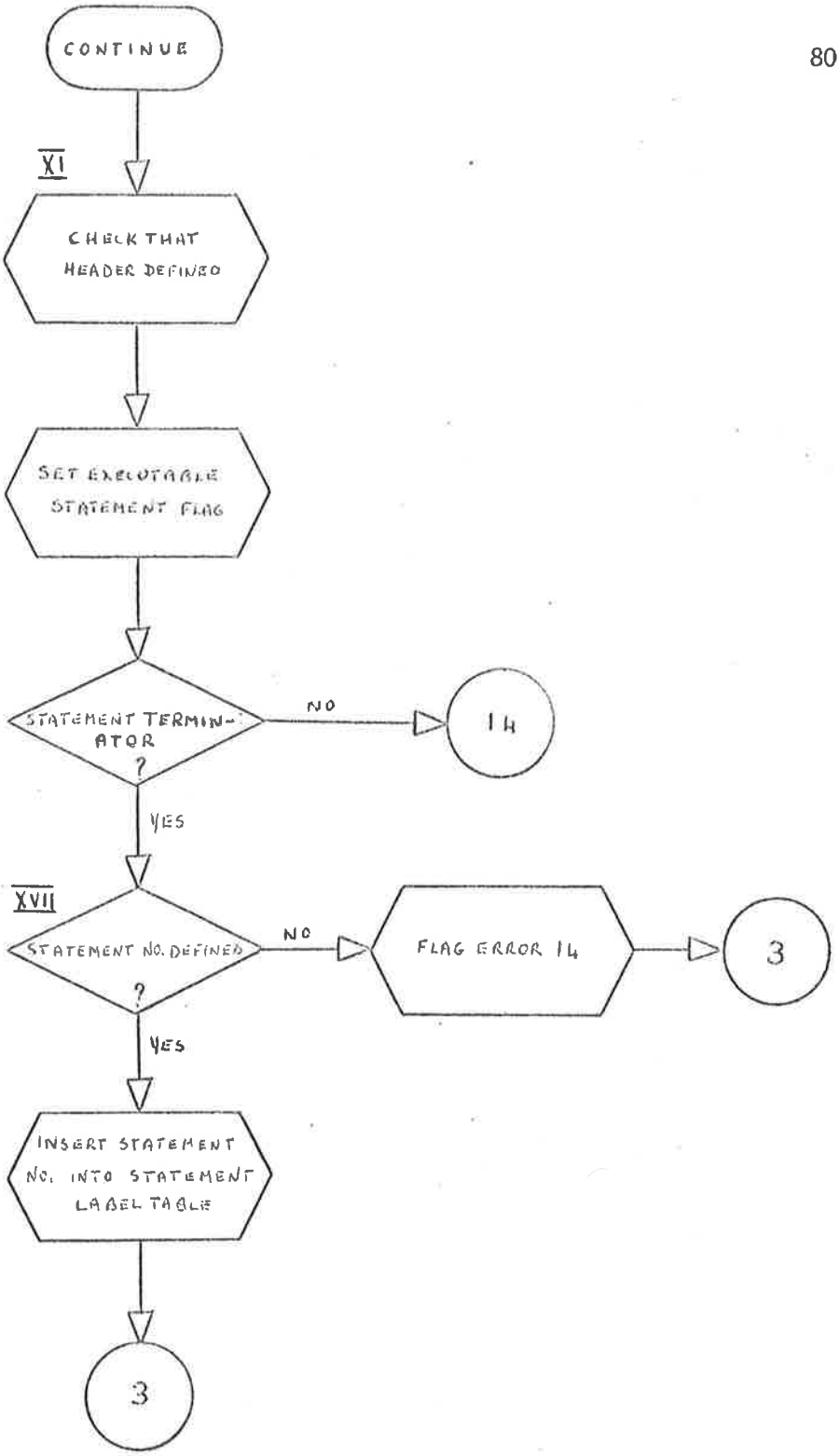

PROCESSOR 2
(FEA)
XI
CHECK THAT
HEADER DEFINED
XIII
INSERT LOGICAL OPER-
ATOR FLAG AND STATE-
MENT NUMBER FLAG
IN STRING HEADER
LOAD ENTRY INTO
PROCESSOR 2 BUFFER
3

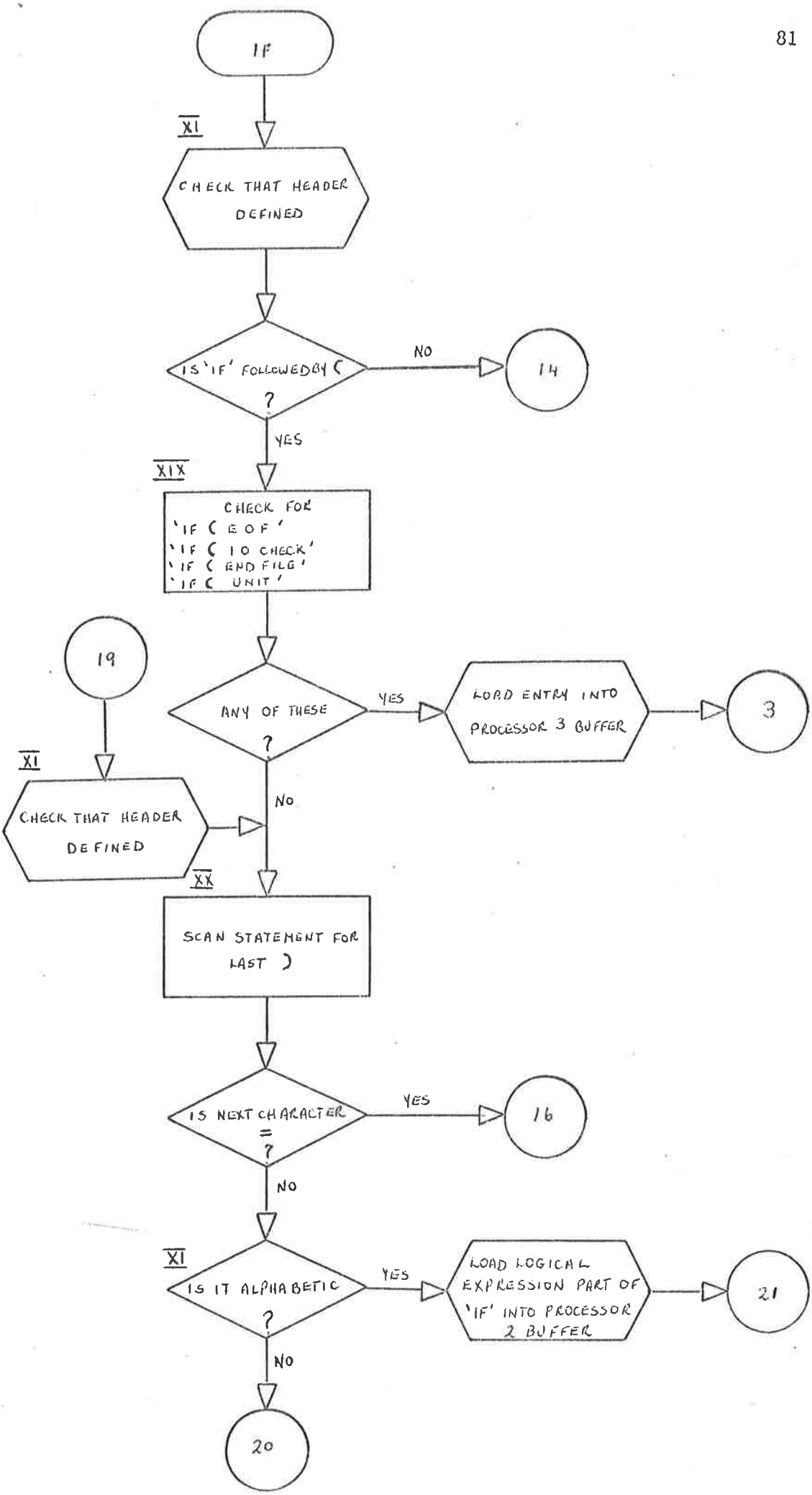

PROCESSOR 3
(MSA)
XI
CHECK THAT HEAD
ER DEFINED
XIV
TRANSFER-
OF-CONTROL STATE-
MENT
?
YES
SET LAST STATEMENT
TYPE FLAG
NO
LOAD ENTRY INTO
PROCESSOR 3 BUFFER
3

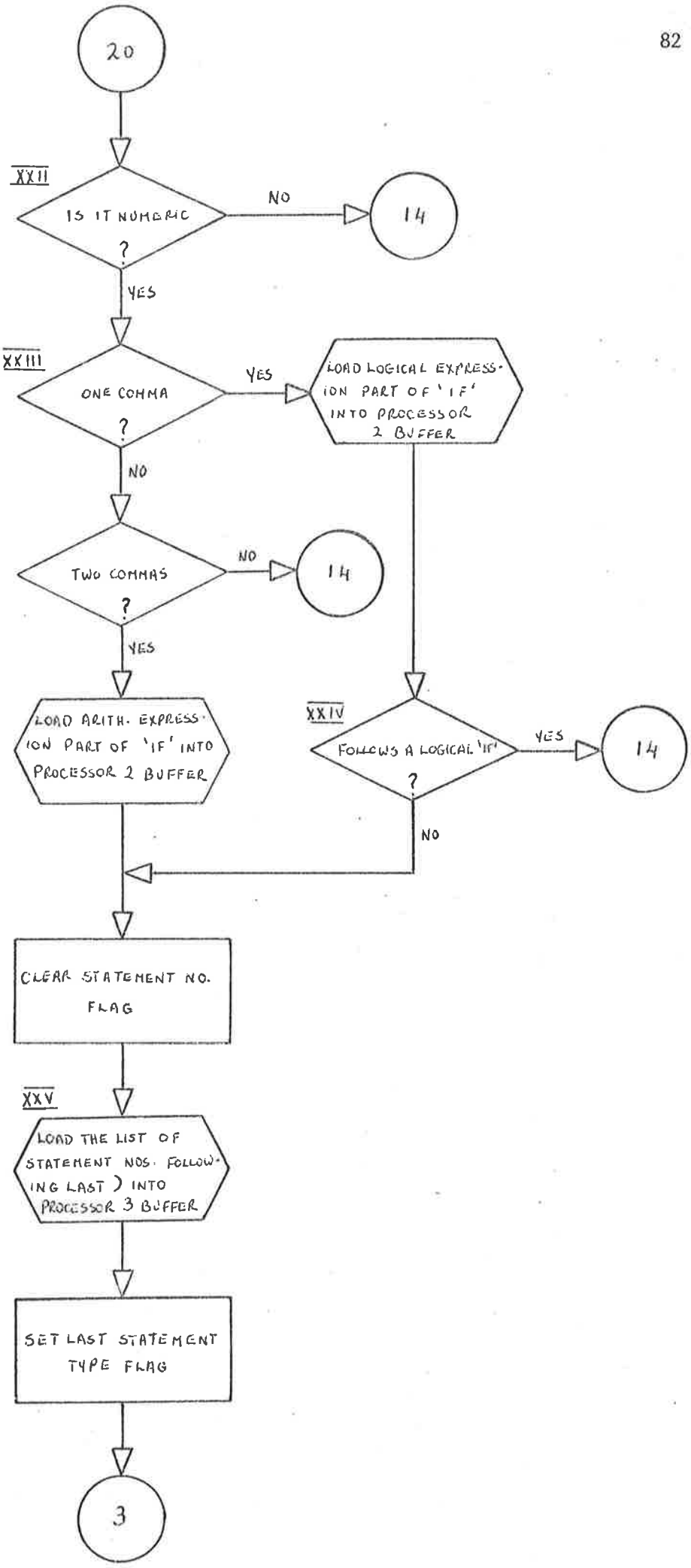

PROCESSOR 0
(GSR)
BLOCK
DATA SUBPROGRAM
?
YES
XV
END
STATEMENT
?
NO
NO
YES
JUMP ON
STATEMENT
TYPE
1A

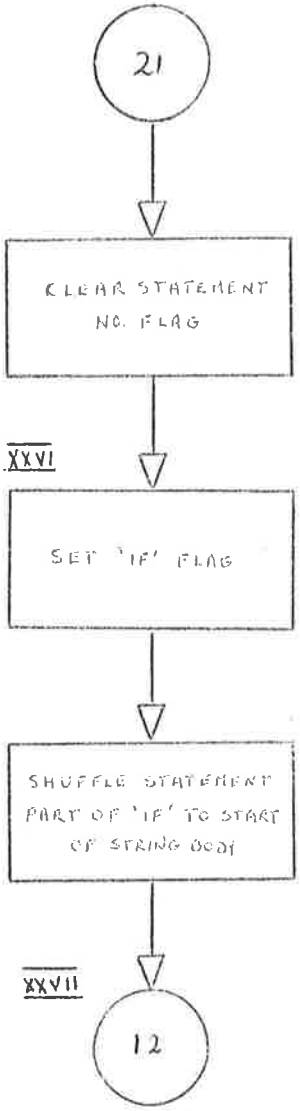

BLOCK
DATA
XVI
CHECK THAT HEAD-
ER NOT ALREADY DE-
FINED
SET BLOCK DATA
FLAG
18

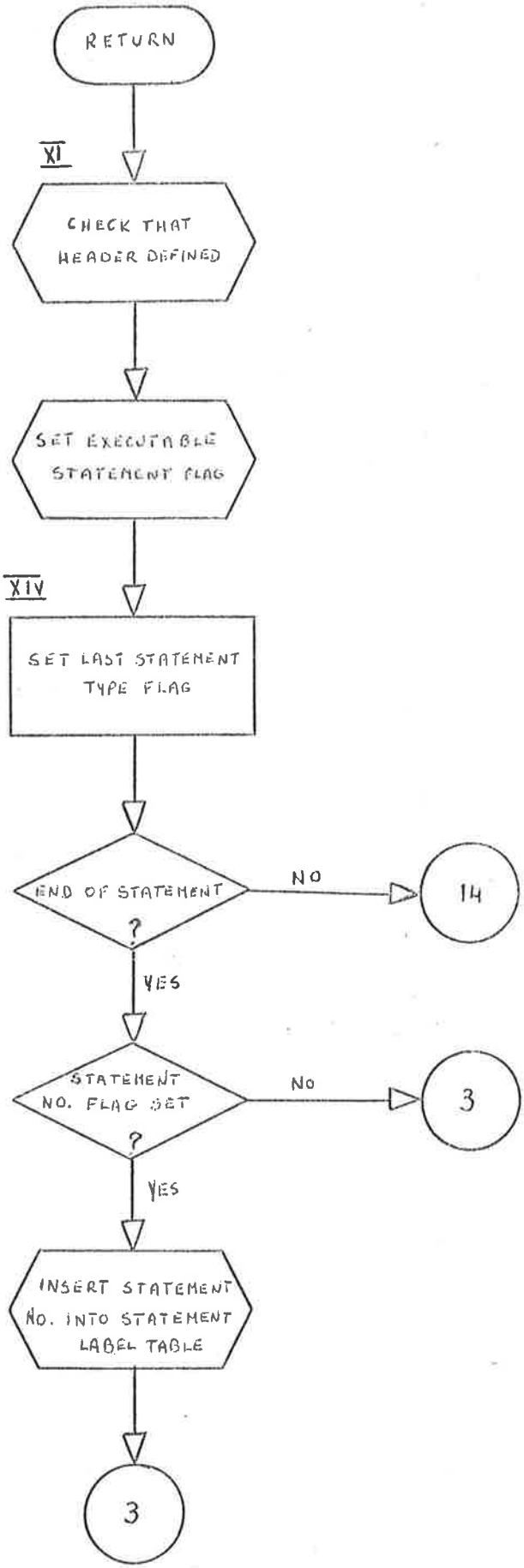

PROGRAM FUNCTION SUBROUTINE
XVI
CHECK THAT HEADER NOT ALREADY DEFINED
END OF STATEMENT ?
YES
18
NO
EXTRACT NAME OF ROUTINE
ERROR IN NAME ?
YES
FLAG ERROR 57
NO
INSERT NAME INTO C.M. SUBPROGRAM NAME AREA
18
LOAD ENTRY INTO PROCESSOR 1 BUFFER
XVII
REQUEST MTR TO LOAD PROGRAM DSA
3

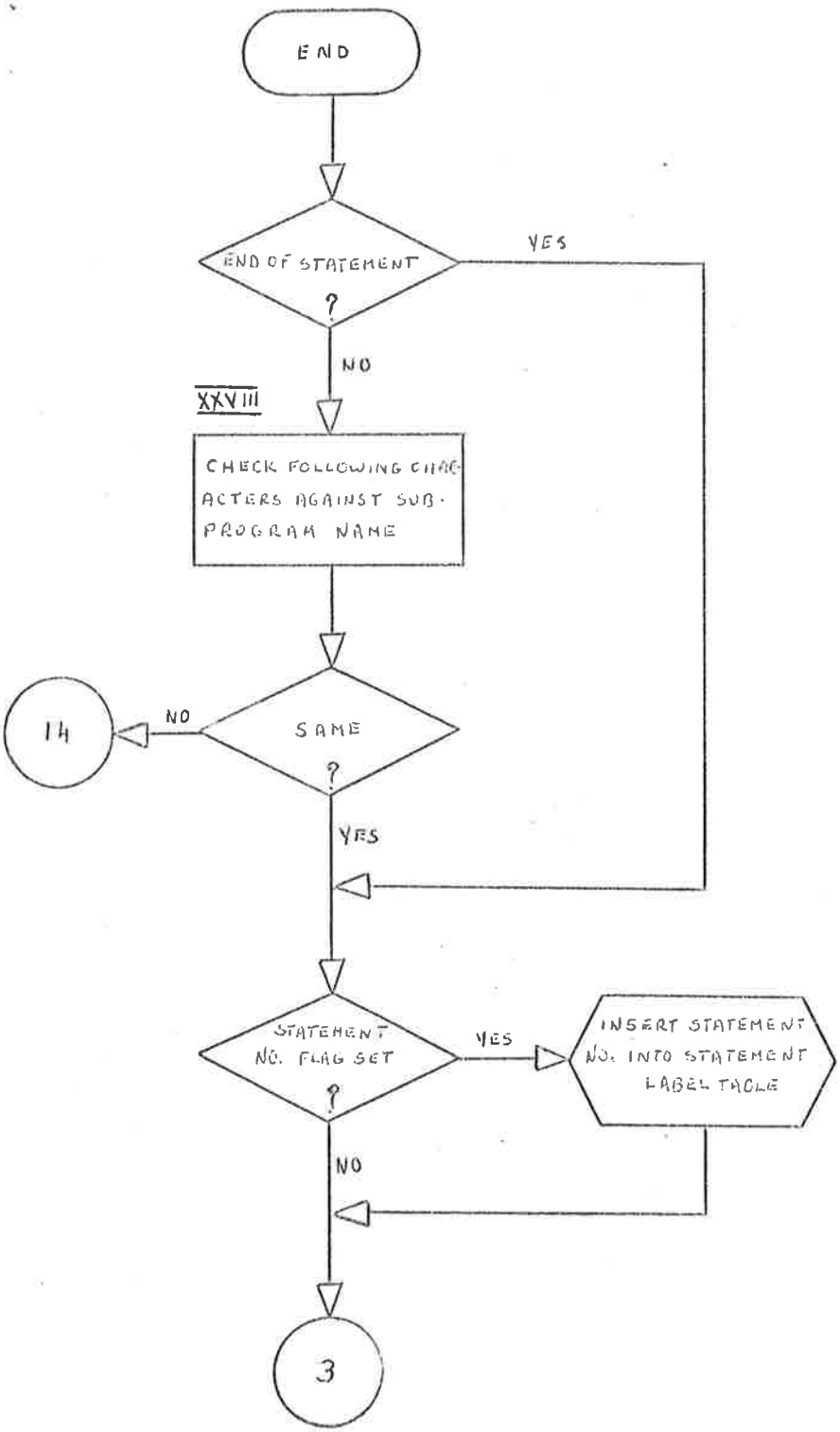

CONTINUE
XI
CHECK THAT HEADER DEFINED
SET EXECUTABLE STATEMENT FLAG
STATEMENT TERMINATOR ?
NO
14
YES
XVII
STATEMENT NO. DEFINED ?
NO
FLAG ERROR 14
3
YES
INSERT STATEMENT NO. INTO STATEMENT LABEL TABLE
3

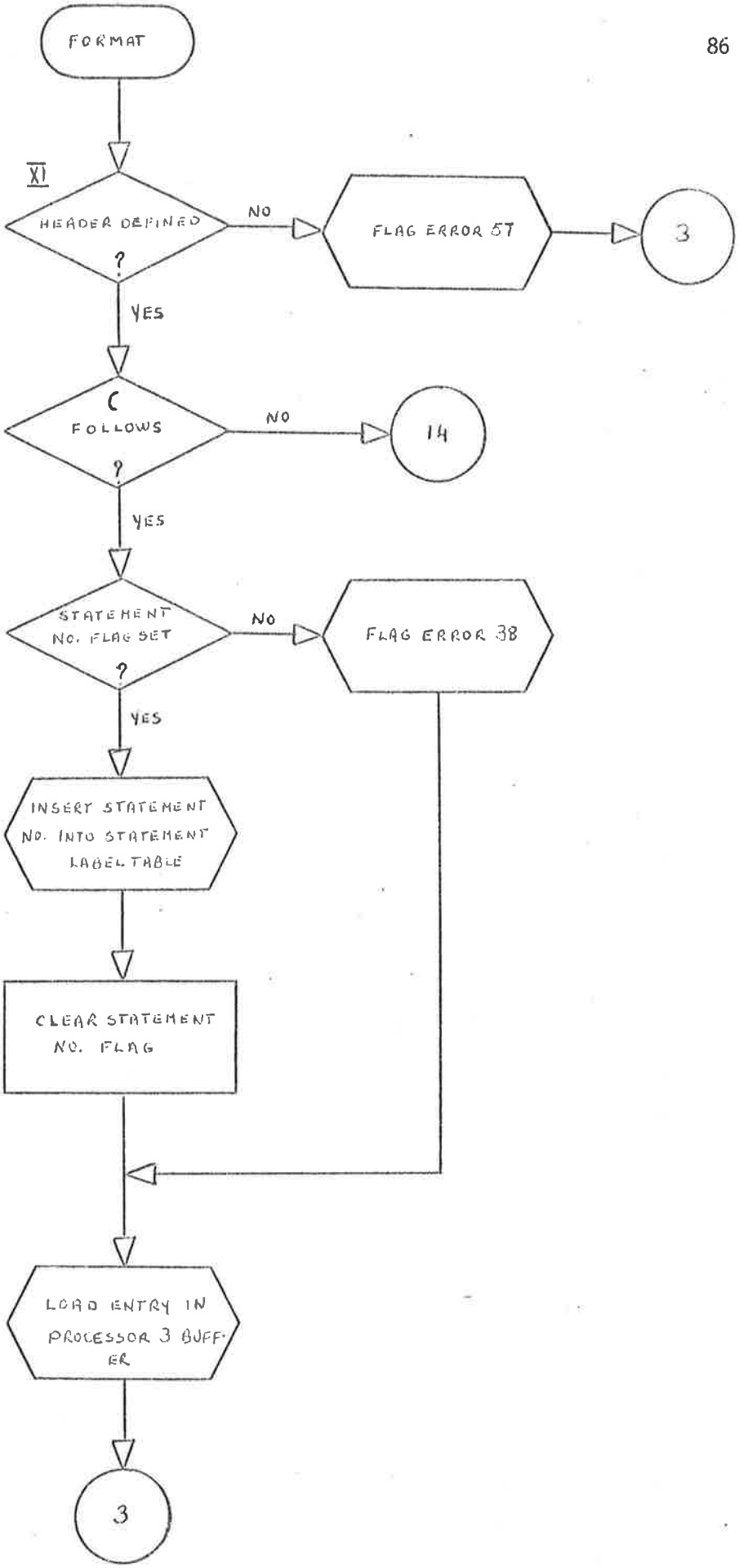

IF
XI
CHECK THAT HEADER DEFINED
IS 'IF' FOLLOWED BY (
?
NO
14
YES
XIX
CHECK FOR
'IF (E O F'
'IF (I O CHECK'
'IF (END FILE'
'IF (UNIT'
19
ANY OF THESE
?
YES
LOAD ENTRY INTO PROCESSOR 3 BUFFER
3
XI
CHECK THAT HEADER DEFINED
NO
XX
SCAN STATEMENT FOR LAST)
IS NEXT CHARACTER =
?
YES
16
NO
XI
IS IT ALPHABETIC
?
YES
LOAD LOGICAL EXPRESSION PART OF 'IF' INTO PROCESSOR 2 BUFFER
21
NO
20

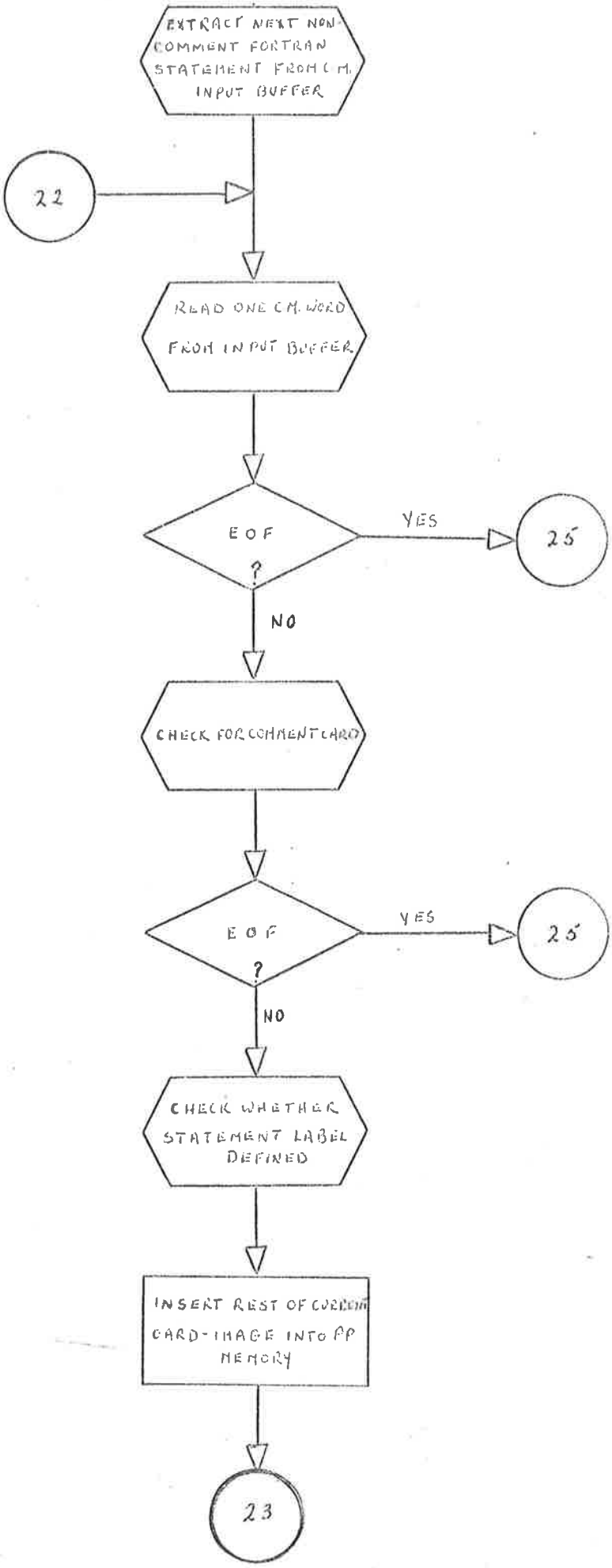

20
XXII
IS IT NUMERIC ?
NO
14
YES
XXIII
ONE COMMA ?
YES
LOAD LOGICAL EXPRESSION PART OF 'IF' INTO PROCESSOR 2 BUFFER
NO
TWO COMMAS ?
NO
14
YES
LOAD ARITH. EXPRESSION PART OF 'IF' INTO PROCESSOR 2 BUFFER
XXIV
FOLLOWS A LOGICAL 'IF' ?
YES
14
NO
CLEAR STATEMENT NO. FLAG
XXV
LOAD THE LIST OF STATEMENT NOS. FOLLOWING LAST) INTO PROCESSOR 3 BUFFER
SET LAST STATEMENT TYPE FLAG
3

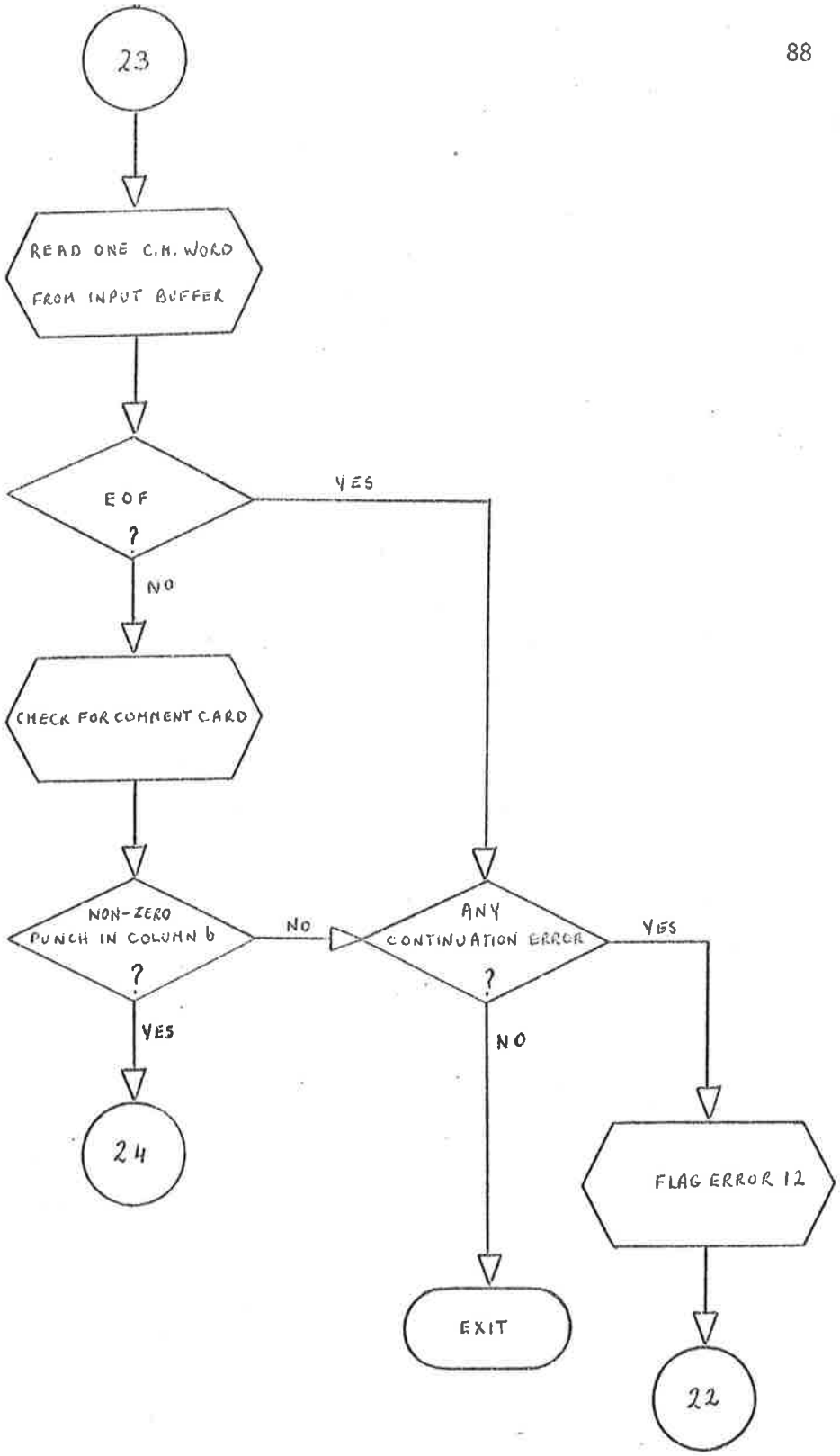

21
CLEAR STATEMENT NO. FLAG
XXVI
SET 'IF' FLAG
SHUFFLE STATEMENT PART OF 'IF' TO START OF STRING 0001
XXVII
12

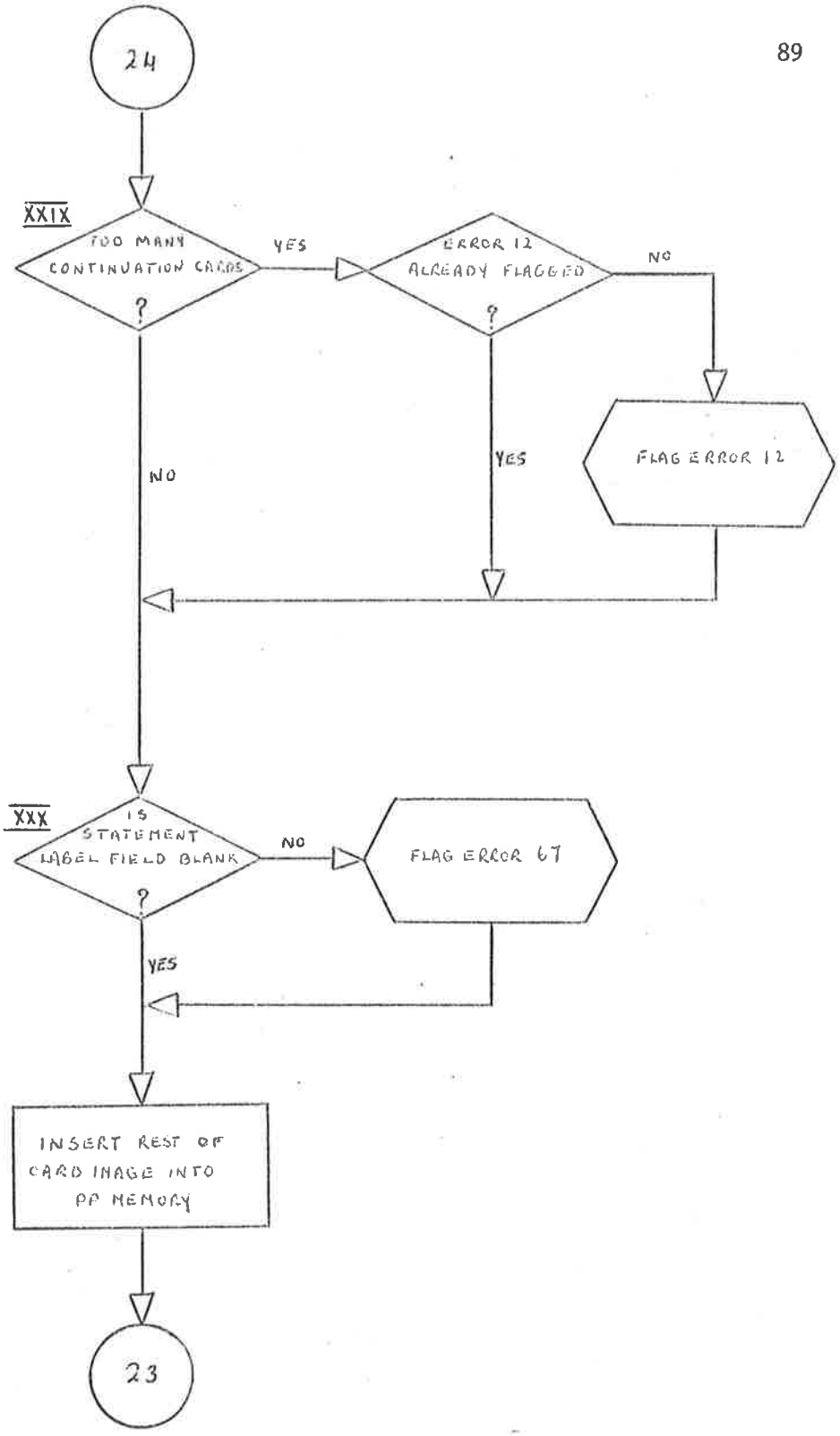

RETURN
XI
CHECK THAT HEADER DEFINED
SET EXECUTABLE STATEMENT FLAG
XIV
SET LAST STATEMENT TYPE FLAG
END OF STATEMENT ?
NO
14
YES
STATEMENT NO. FLAG SET ?
No
3
YES
INSERT STATEMENT No. INTO STATEMENT LABEL TABLE
3

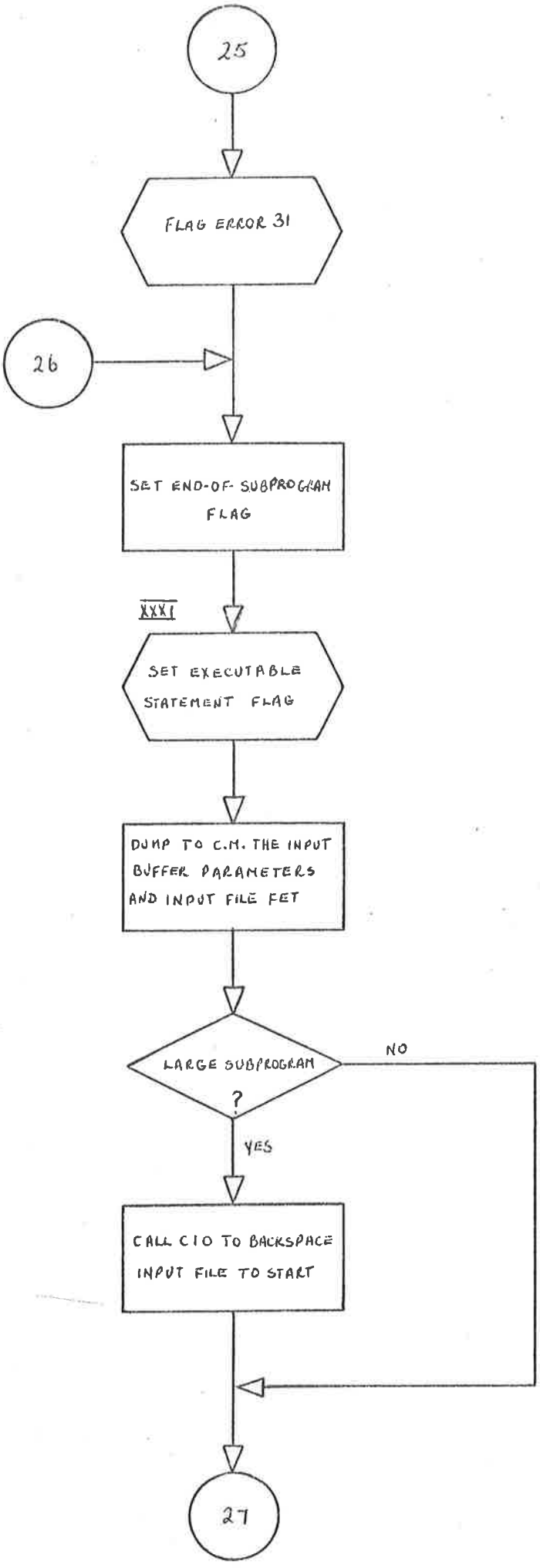

END
END OF STATEMENT ?
YES
NO
XXVIII
CHECK FOLLOWING CHARACTERS AGAINST SUB-PROGRAM NAME
SAME ?
NO
14
YES
STATEMENT NO. FLAG SET ?
YES
INSERT STATEMENT NO. INTO STATEMENT LABEL TABLE
NO
3

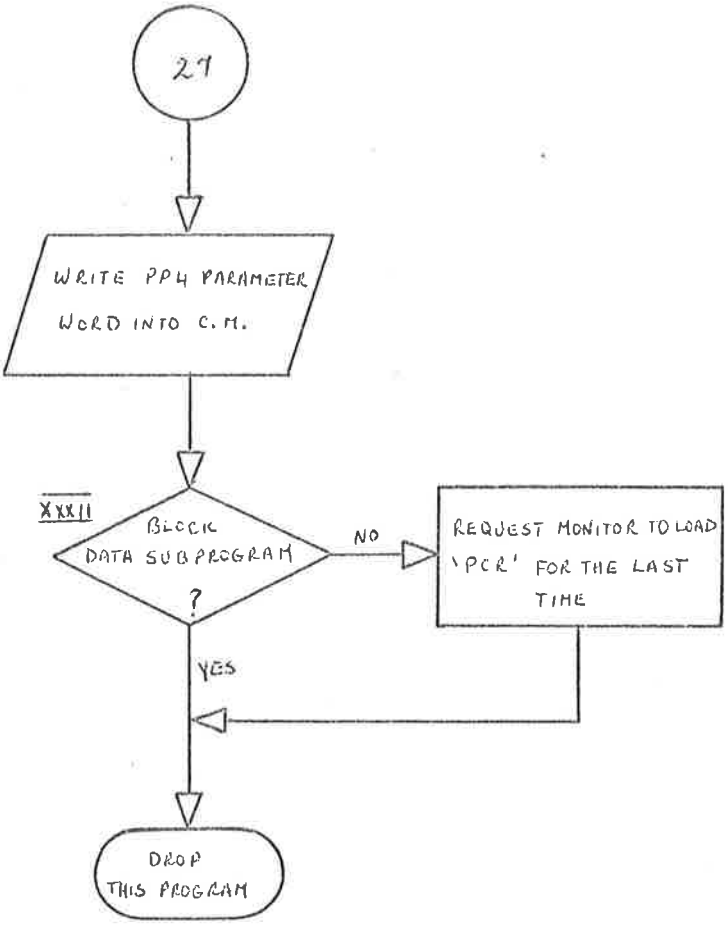

FORMAT
XI
HEADER DEFINED ?
NO
FLAG ERROR 57
3
YES
C FOLLOWS ?
NO
14
YES
STATEMENT NO. FLAG SET ?
NO
FLAG ERROR 38
YES
INSERT STATEMENT NO. INTO STATEMENT LABEL TABLE
CLEAR STATEMENT NO. FLAG
LOAD ENTRY IN PROCESSOR 3 BUFFER
3

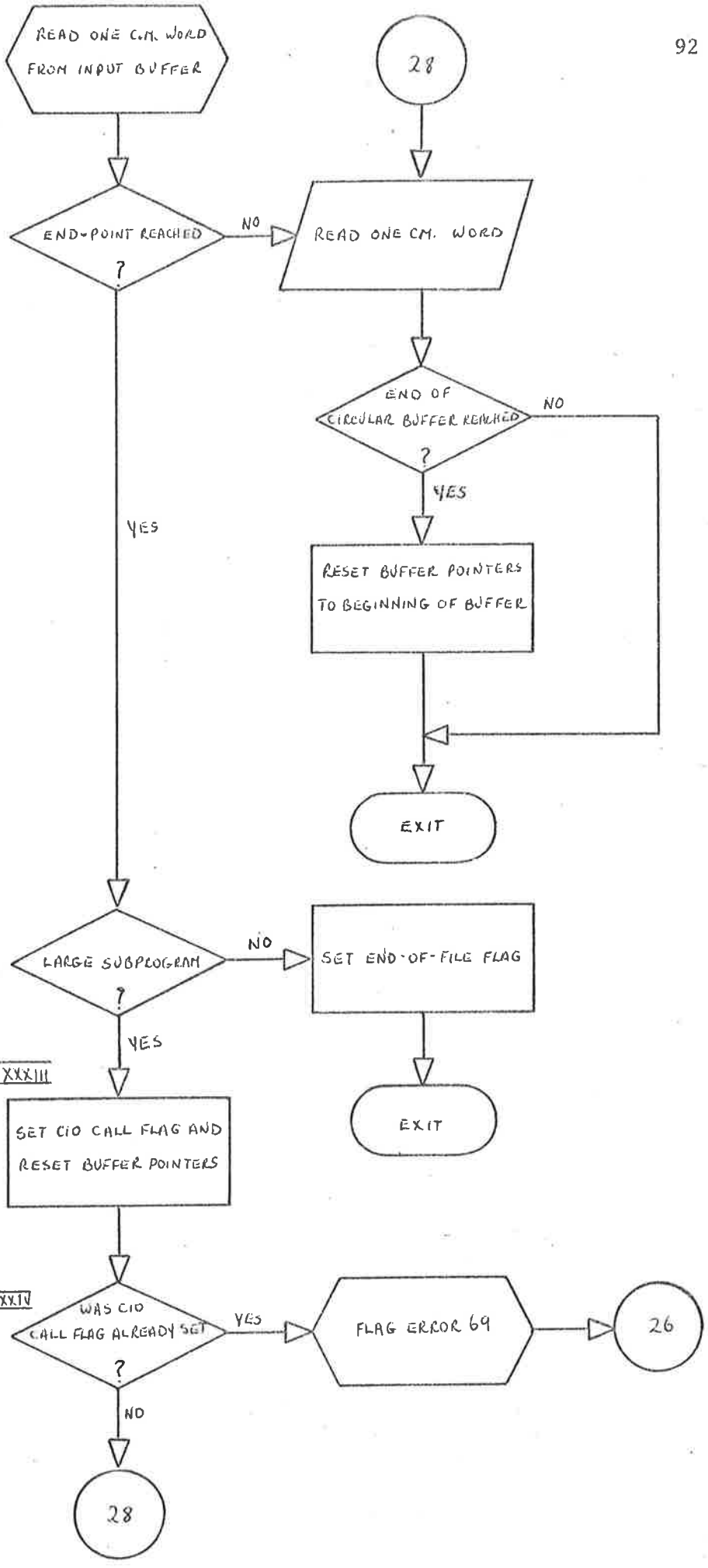

EXTRACT NEXT NON-COMMENT FORTRAN STATEMENT FROM C.M. INPUT BUFFER
22
READ ONE CM WORD FROM INPUT BUFFER
EOF ?
YES
25
NO
CHECK FOR COMMENT CARD
EOF ?
YES
25
NO
CHECK WHETHER STATEMENT LABEL DEFINED
INSERT REST OF CURRENT CARD-IMAGE INTO PP MEMORY
23

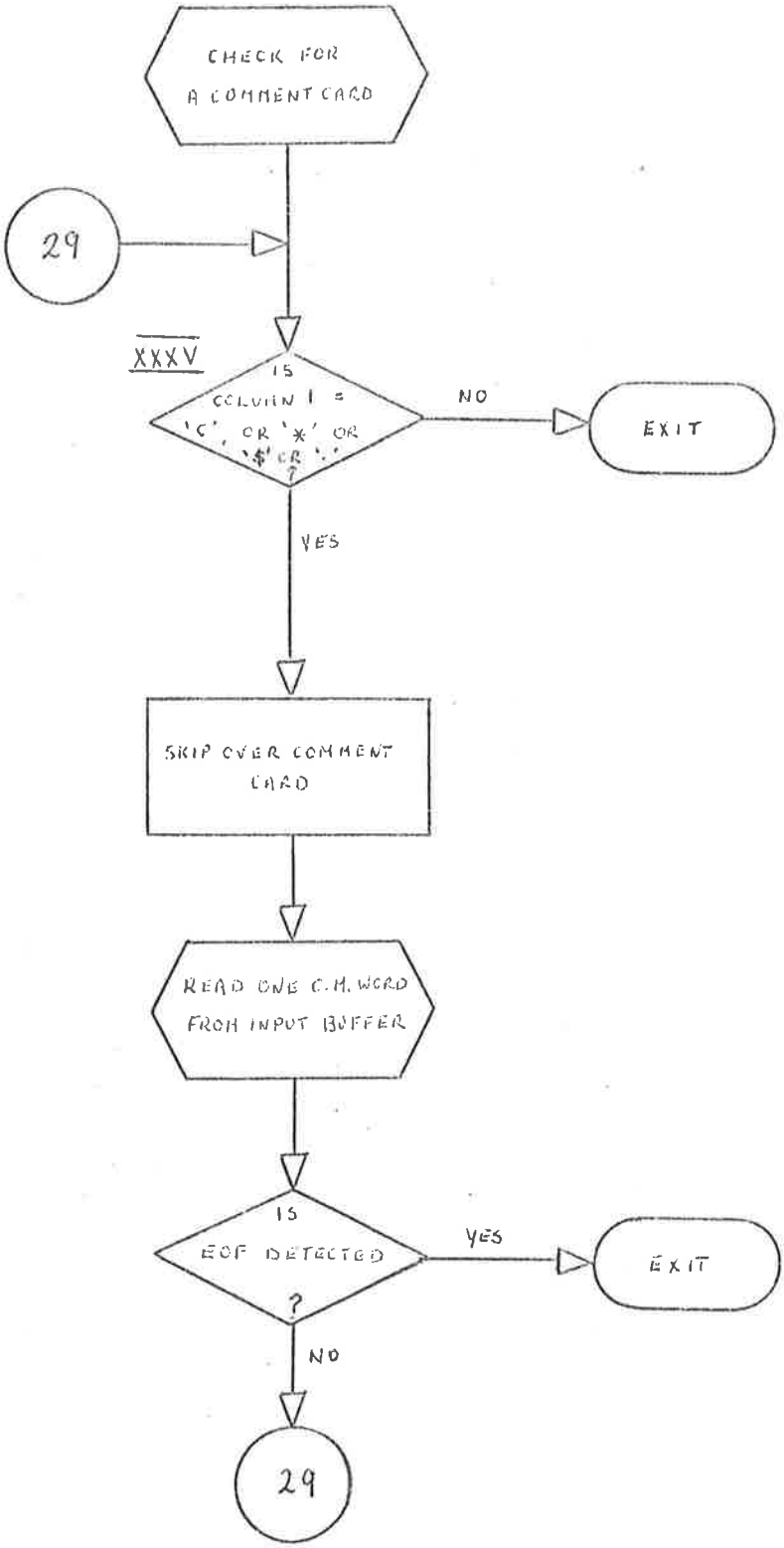

23
READ ONE C.M. WORD FROM INPUT BUFFER
EOF ?
YES
NO
CHECK FOR COMMENT CARD
NON-ZERO PUNCH IN COLUMN 6 ?
NO
YES
24
ANY CONTINUATION ERROR ?
YES
NO
FLAG ERROR 12
EXIT
22

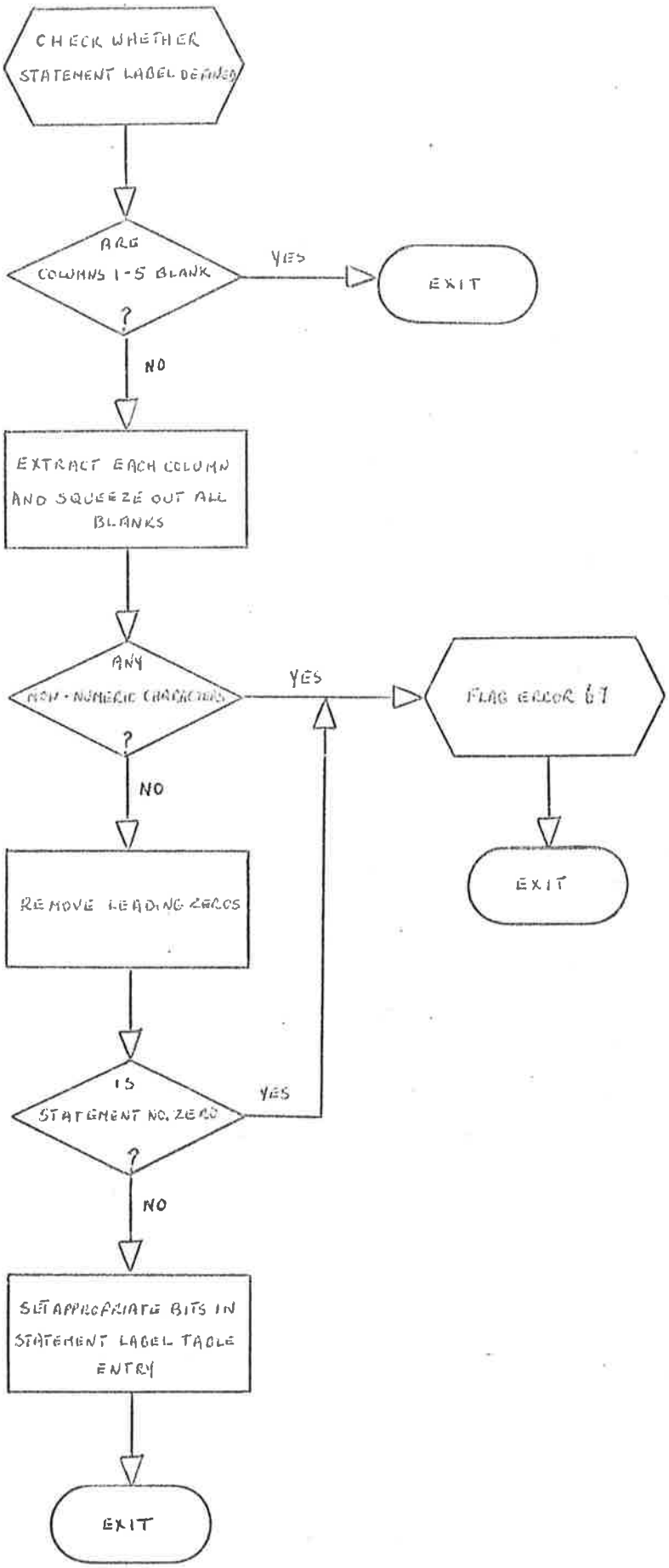

24
XXIX
TOO MANY CONTINUATION CARDS ?
YES
ERROR 12 ALREADY FLAGGED ?
NO
FLAG ERROR 12
YES
NO
XXX
IS STATEMENT LABEL FIELD BLANK ?
NO
FLAG ERROR 67
YES
INSERT REST OF CARD IMAGE INTO PP MEMORY
23

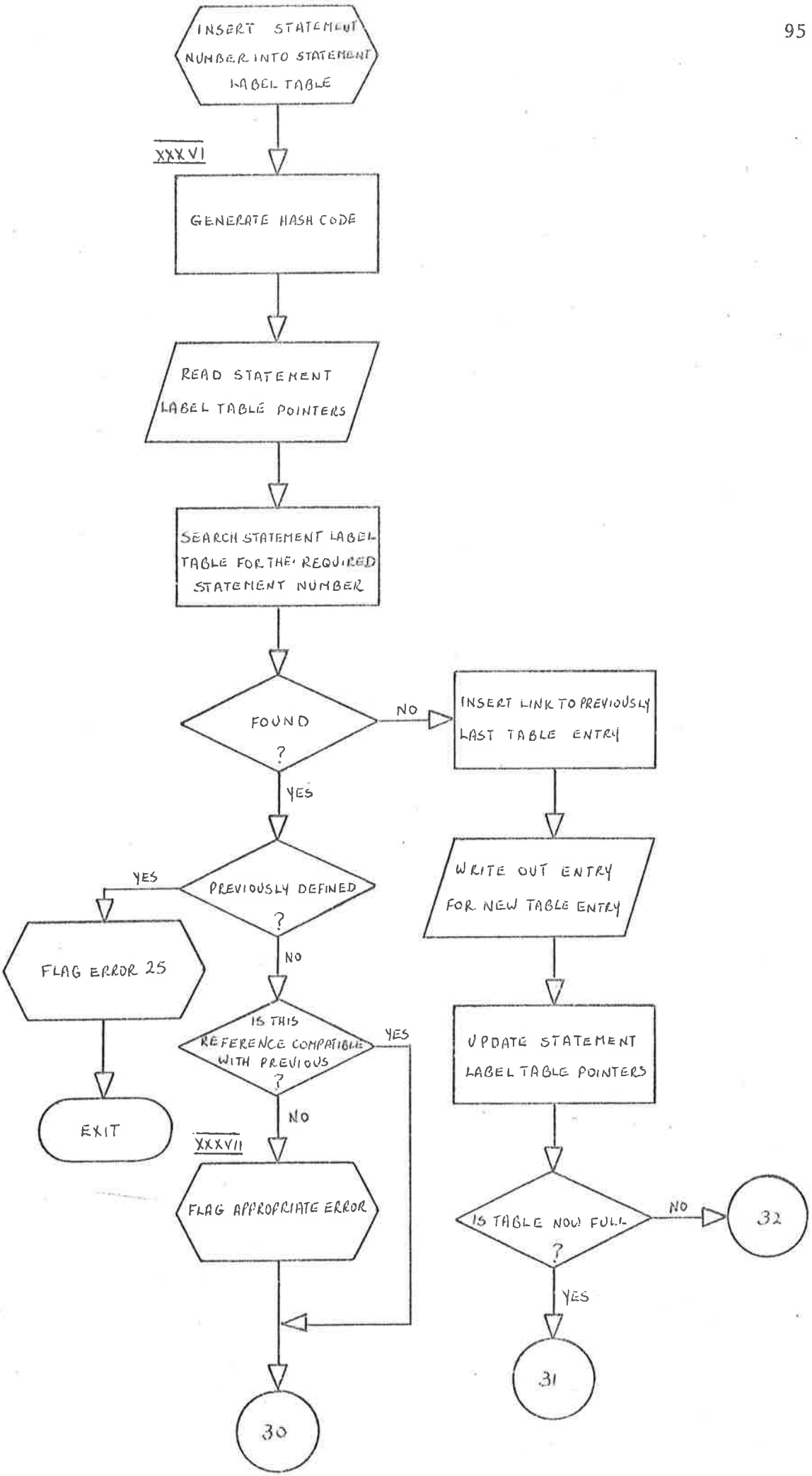

25
FLAG ERROR 31
26
SET END-OF-SUBPROGRAM FLAG
XXXI
SET EXECUTABLE STATEMENT FLAG
DUMP TO C.M. THE INPUT BUFFER PARAMETERS AND INPUT FILE FET
LARGE SUBPROGRAM ?
NO
YES
CALL CIO TO BACKSPACE INPUT FILE TO START
27

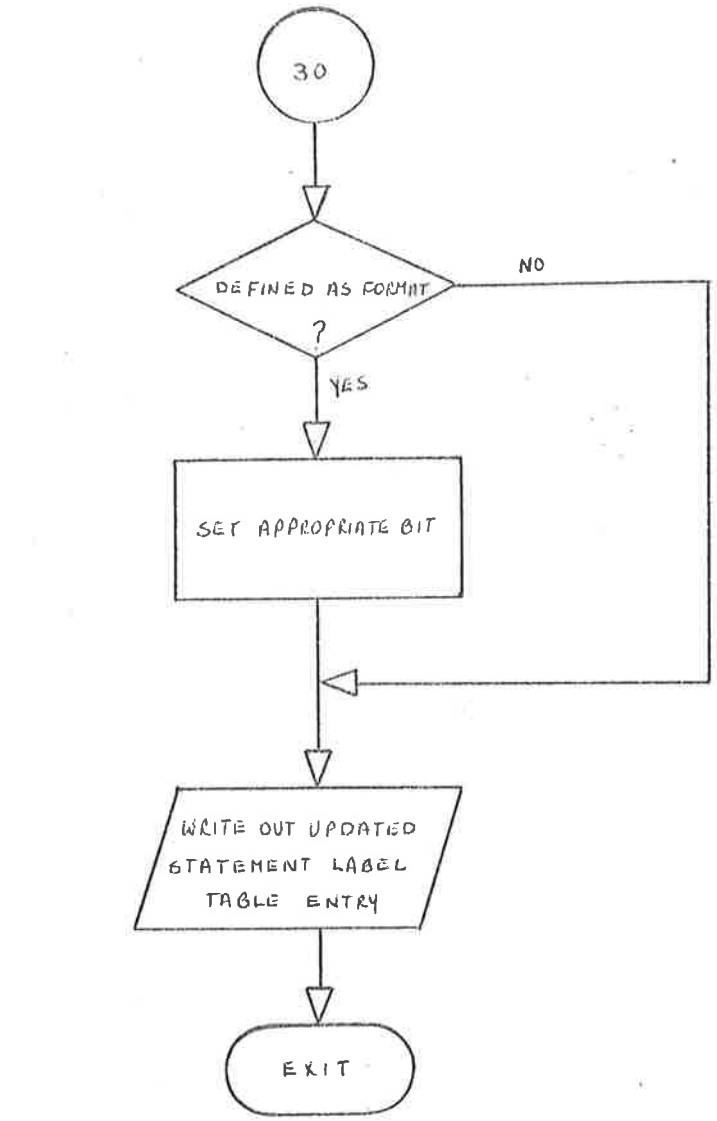

27
WRITE PP4 PARAMETER WORD INTO C.M.
XXXII
BLOCK DATA SUBPROGRAM ?
NO
REQUEST MONITOR TO LOAD 'PCR' FOR THE LAST TIME
YES
DROP THIS PROGRAM

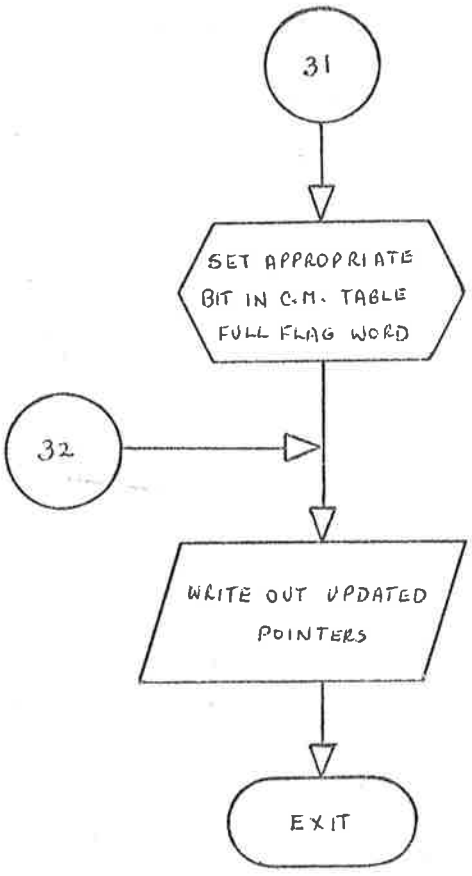

READ ONE C.M. WORD FROM INPUT BUFFER
28
END-POINT REACHED ?
NO
READ ONE CM. WORD
END OF CIRCULAR BUFFER REACHED ?
NO
YES
YES
RESET BUFFER POINTERS TO BEGINNING OF BUFFER
EXIT
LARGE SUBPROGRAM ?
NO
SET END-OF-FILE FLAG
YES
XXXIII
EXIT
SET CIO CALL FLAG AND RESET BUFFER POINTERS
XXXIV
WAS CIO CALL FLAG ALREADY SET ?
YES
FLAG ERROR 69
26
NO
28

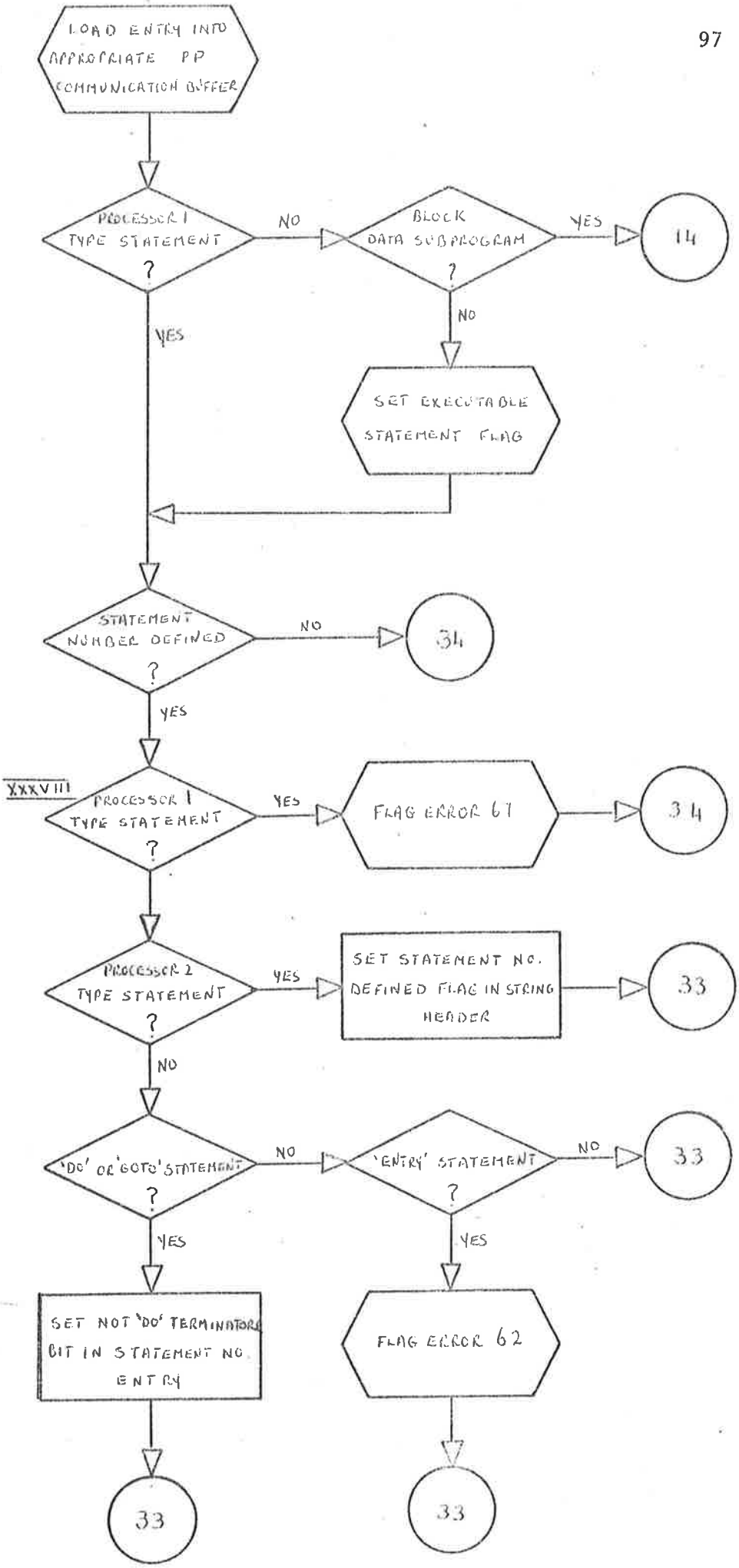

CHECK FOR
A COMMENT CARD
29
XXXV
IS
COLUMN 1 =
'C' OR '*' OR
'$' OR '.'
?
NO
EXIT
YES
SKIP OVER COMMENT
CARD
READ ONE C.M. WORD
FROM INPUT BUFFER
IS
EOF DETECTED
?
YES
EXIT
NO
29

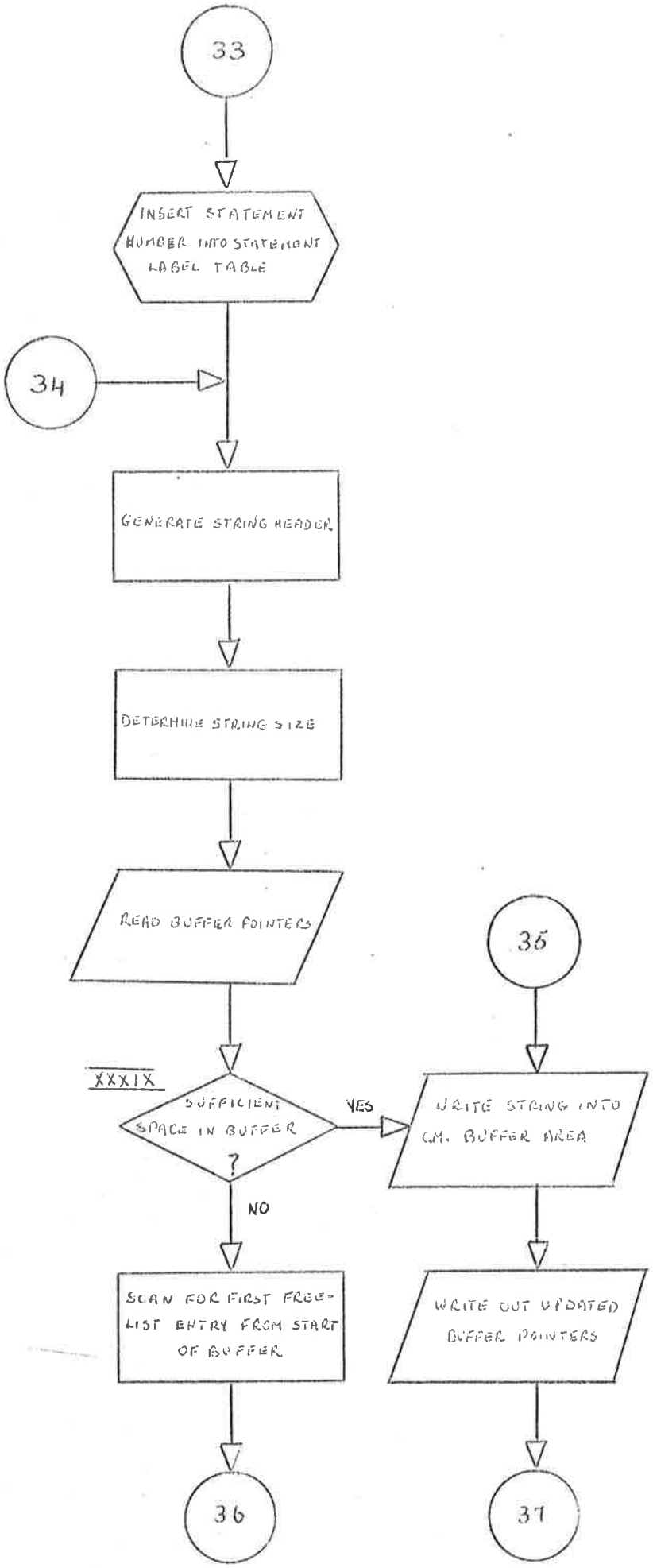

CHECK WHETHER STATEMENT LABEL DEFINED
ARE COLUMNS 1-5 BLANK ?
YES
EXIT
NO
EXTRACT EACH COLUMN AND SQUEEZE OUT ALL BLANKS
ANY NON-NUMERIC CHARACTERS ?
YES
FLAG ERROR 67
EXIT
NO
REMOVE LEADING ZEROS
IS STATEMENT NO. ZERO ?
YES
NO
SET APPROPRIATE BITS IN STATEMENT LABEL TABLE ENTRY
EXIT

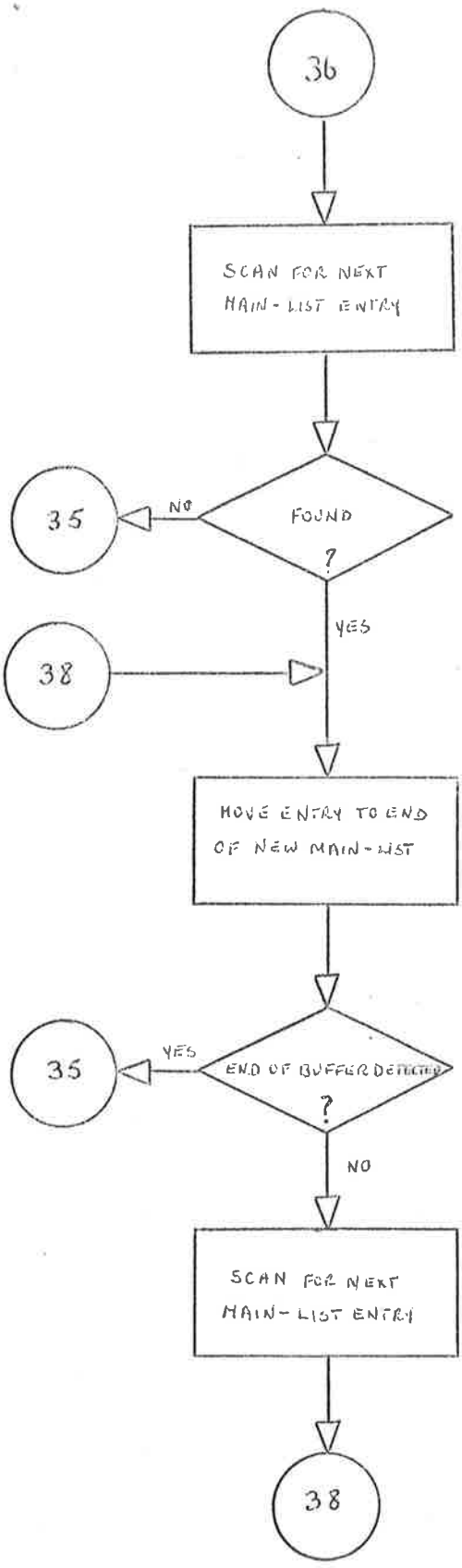

INSERT STATEMENT NUMBER INTO STATEMENT LABEL TABLE
XXXVI
GENERATE HASH CODE
READ STATEMENT LABEL TABLE POINTERS
SEARCH STATEMENT LABEL TABLE FOR THE REQUIRED STATEMENT NUMBER
FOUND ?
NO
INSERT LINK TO PREVIOUSLY LAST TABLE ENTRY
YES
PREVIOUSLY DEFINED ?
YES
FLAG ERROR 25
EXIT
NO
IS THIS REFERENCE COMPATIBLE WITH PREVIOUS ?
YES
NO
XXXVII
FLAG APPROPRIATE ERROR
30
WRITE OUT ENTRY FOR NEW TABLE ENTRY
UPDATE STATEMENT LABEL TABLE POINTERS
IS TABLE NOW FULL ?
NO
32
YES
31

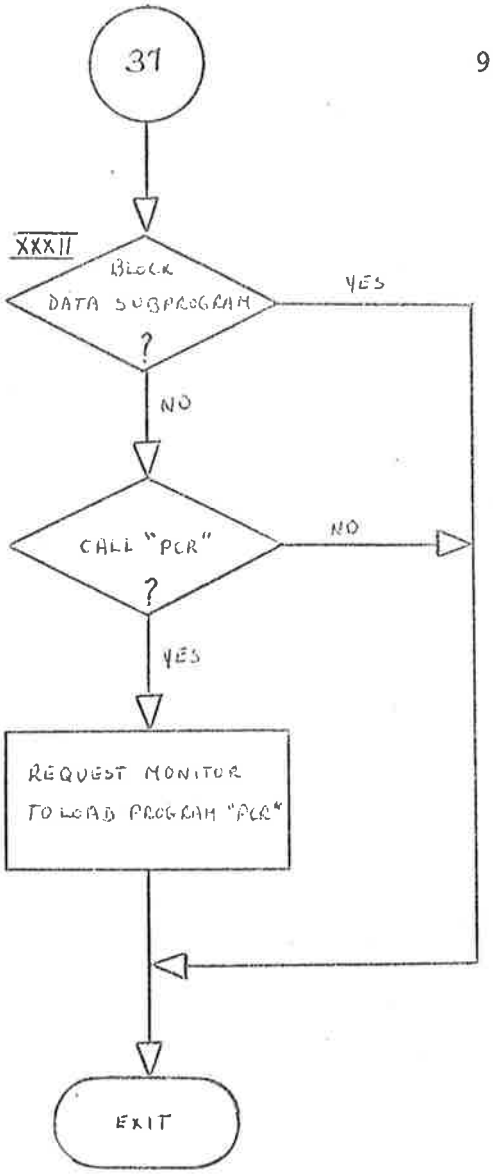

30
DEFINED AS FORMAT ?
NO
YES
SET APPROPRIATE BIT
WRITE OUT UPDATED STATEMENT LABEL TABLE ENTRY
EXIT


31
SET APPROPRIATE BIT IN C.M. TABLE FULL FLAG WORD
32
WRITE OUT UPDATED POINTERS
EXIT


LOAD ENTRY INTO APPROPRIATE PP COMMUNICATION BUFFER
PROCESSOR 1 TYPE STATEMENT ?
NO
BLOCK DATA SUBPROGRAM ?
YES
14
NO
YES
SET EXECUTABLE STATEMENT FLAG
STATEMENT NUMBER DEFINED ?
NO
34
YES
XXXVIII
PROCESSOR 1 TYPE STATEMENT ?
YES
FLAG ERROR 61
34
PROCESSOR 2 TYPE STATEMENT ?
YES
SET STATEMENT NO. DEFINED FLAG IN STRING HEADER
33
NO
'DO' OR 'GOTO' STATEMENT ?
NO
'ENTRY' STATEMENT ?
NO
33
YES
YES
SET NOT 'DO' TERMINATOR BIT IN STATEMENT NO. ENTRY
FLAG ERROR 62
33
33


33
INSERT STATEMENT NUMBER INTO STATEMENT LABEL TABLE
34
GENERATE STRING HEADER
DETERMINE STRING SIZE
READ BUFFER POINTERS
XXXIX
SUFFICIENT SPACE IN BUFFER ?
YES
NO
WRITE STRING INTO CM. BUFFER AREA
SCAN FOR FIRST FREE-LIST ENTRY FROM START OF BUFFER
35
WRITE OUT UPDATED BUFFER POINTERS
36
37


36
SCAN FOR NEXT MAIN-LIST ENTRY
35
NO
FOUND ?
YES
38
MOVE ENTRY TO END OF NEW MAIN-LIST
35
YES
END OF BUFFER DETECTED ?
NO
SCAN FOR NEXT MAIN-LIST ENTRY
38


37
XXXII
BLOCK DATA SUBPROGRAM ?
YES
NO
CALL "PCR" ?
NO
YES
REQUEST MONITOR TO LOAD PROGRAM "PCR"
EXIT

7. PROGRAM PCR - PROCESSOR CALL ROUTINE

7.0 Introduction

This program is loaded on request by program GSR after every eighth entry has been loaded into a processor communication buffer and also for the last time, during the processing of a given source subprogram, immediately before program GSR drops itself on detecting the end of the current subprogram. Its task is to determine the number required, and request the loading of as many additional copies of programs FEA and MSA as is necessary for the rapid completion of the processing of the current subprogram.

From the processor communication buffer pointers for these programs, it calculates the number of copies of the corresponding program that should be running simultaneously on the basis of one copy of the program for every eight entries in the buffer provided that if the buffer is non-empty, at least one copy is required. The last request for this program, for a given subprogram, differs from preceding ones in that requests for programs FEA and MSA are made on the basis of one copy for every two entries in their corresponding communication buffers. This ensure that in spite of any imbalance in the distribution of entries to processors 2 (that is, FEA) and 3 (MSA), both processors' communication buffers will become cleared at approximately the same time and that this will occur as quickly as possible after the completion of program GSR.

The last call to this program requests the windup routine, program PP4, to be loaded and executed, if the processor status table shows that no other programs are active. The latter event is highly unlikely to occur so soon after the completion of program GSR. But the check is inserted here because otherwise the compilation system would hang, if the event did occur.

Since program PCR does not interact with any other program in any way, it is not necessary for any other program to be aware of its status. Therefore no entry is created for it in the processor status table. Instead, it uses a special processor number in all its central memory table access requests. This number is passed to it by program GSR in the usual way via the lower eighteen bits of the processor's input register. It starts at twenty for the first call to the program for the current subprogram and increases by one on every subsequent call to the program.

7.1 Explanatory Notes to the Flow-Chart

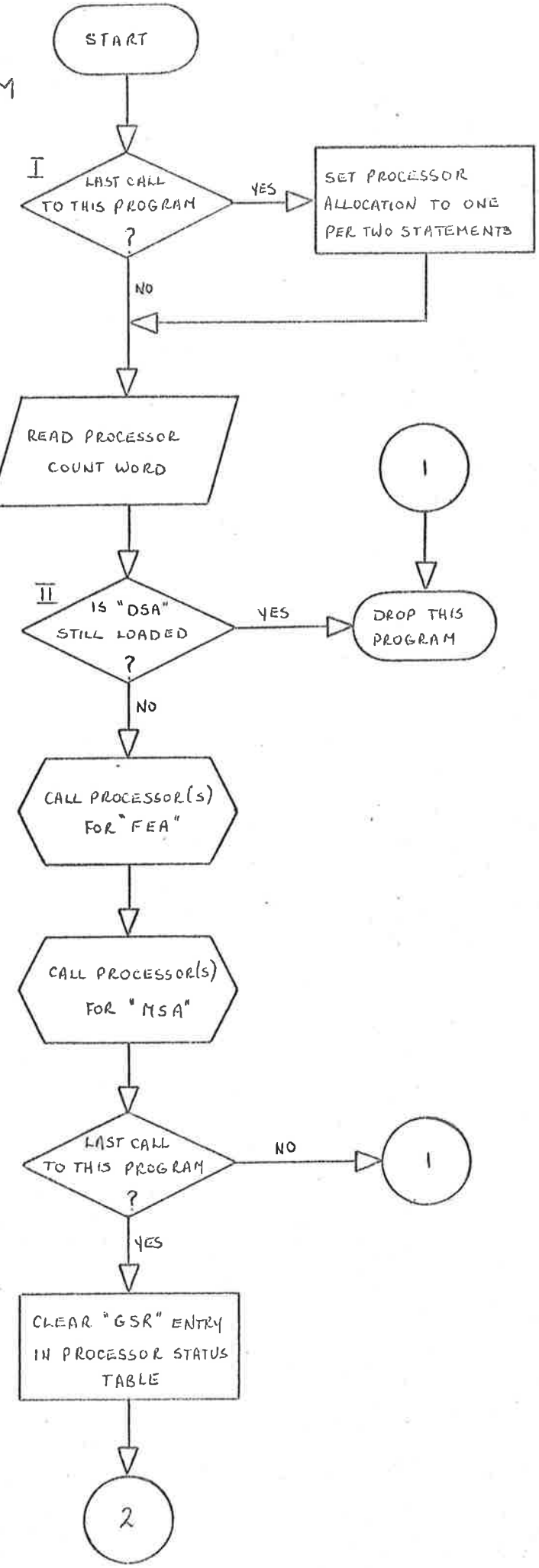
Although not explicitly stated in the accompanying flow-chart, references to all central memory tables are made only after access to the required table has been successfully requested, using the procedure described in 5.1.0.

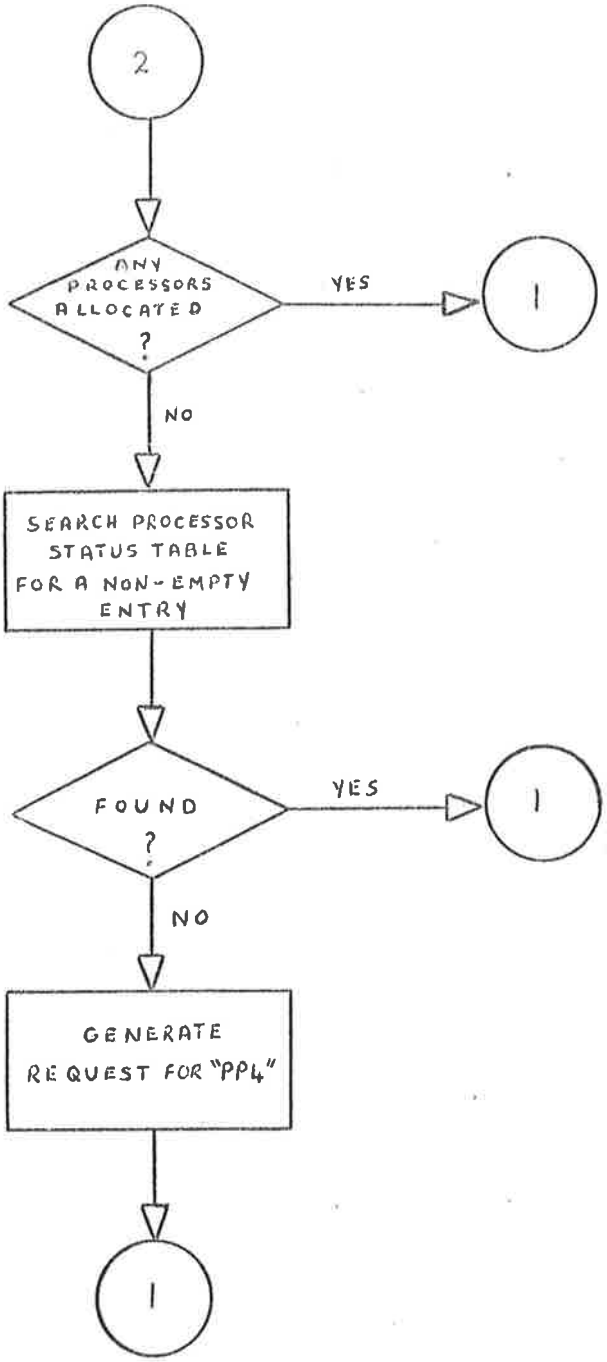
The following notes refer to the flow-chart:

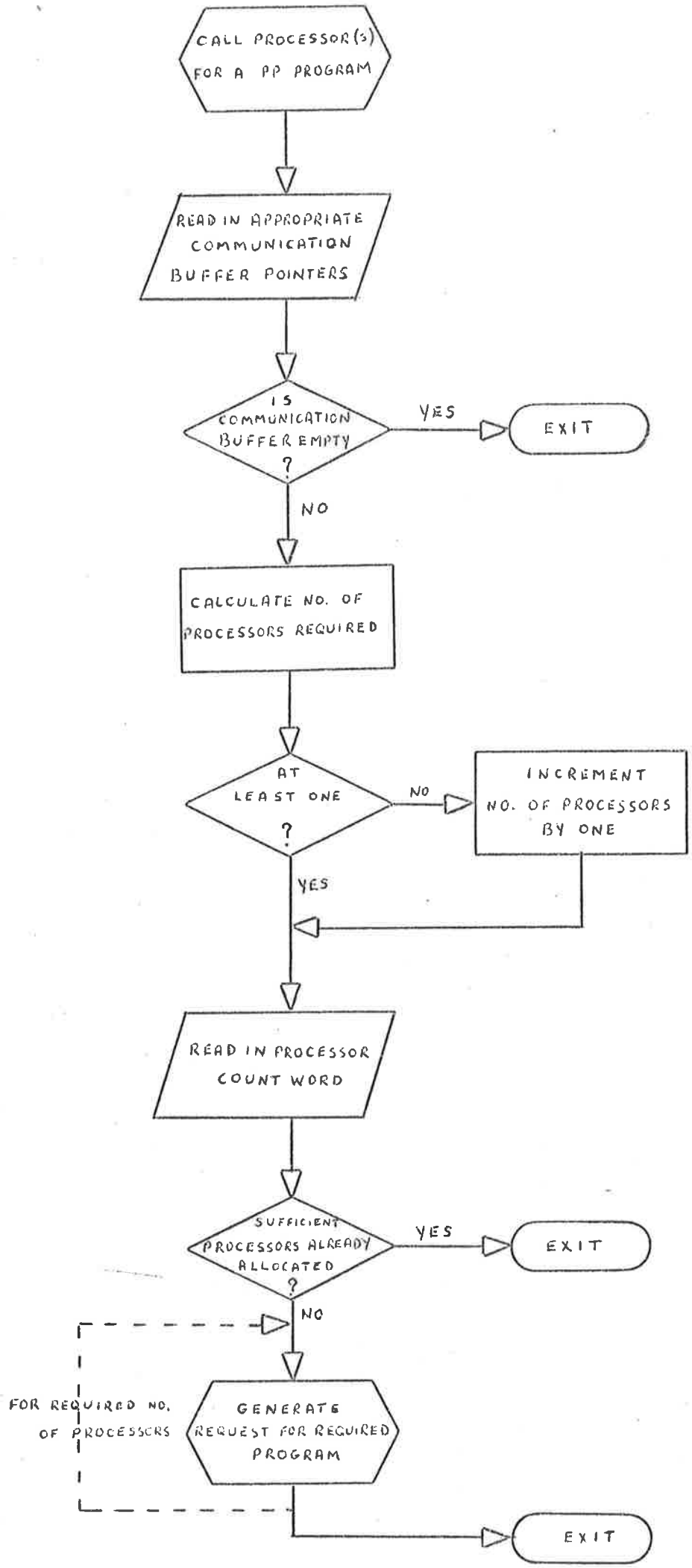
I. If not the last call to this program, processor allocation is made on the basis of one copy of the required program for every eight statements in the corresponding communication buffer.

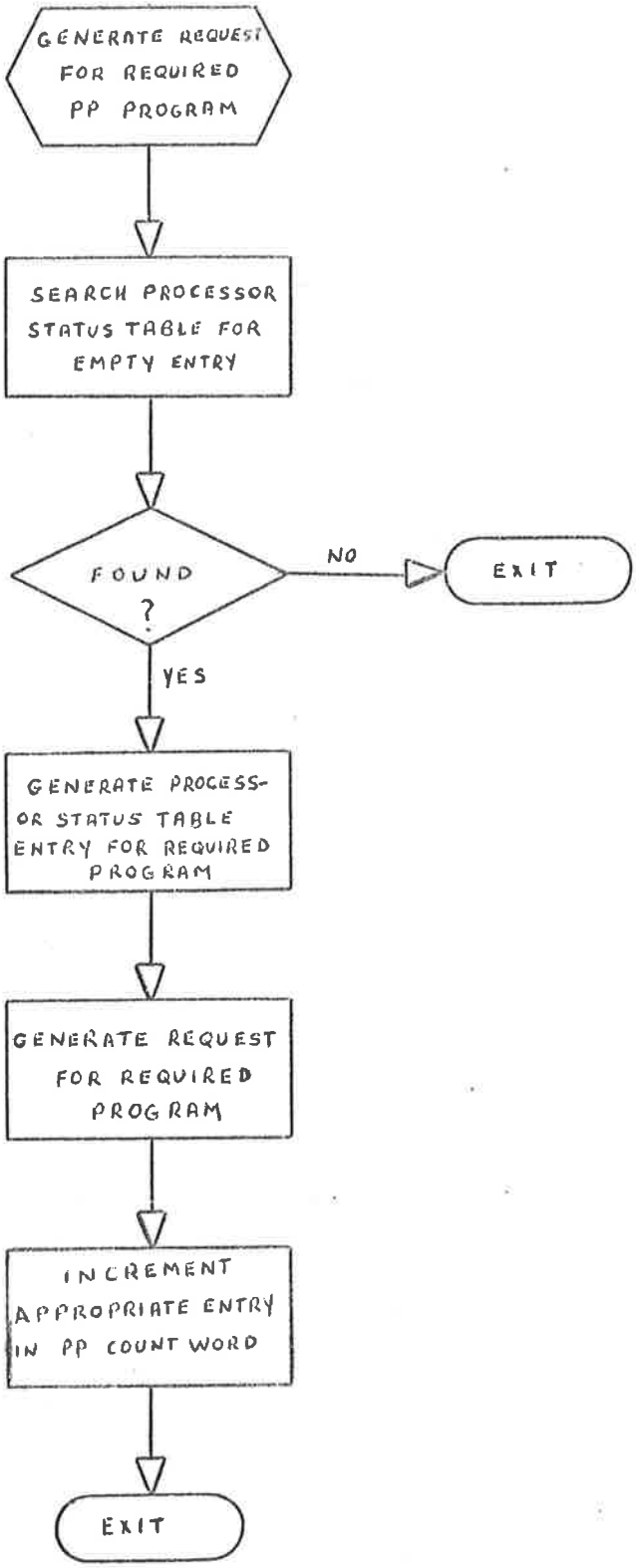
II. If this is true, the program drops itself since no statements may be processed by programs FEA and MSA until program DSA has completed the processing of all declaratives.

PROGRAM PCR


START
I
LAST CALL TO THIS PROGRAM ?
YES
SET PROCESSOR ALLOCATION TO ONE PER TWO STATEMENTS
NO
READ PROCESSOR COUNT WORD
1
II
IS "DSA" STILL LOADED ?
YES
DROP THIS PROGRAM
NO
CALL PROCESSOR(S) FOR "FEA"
CALL PROCESSOR(S) FOR "MSA"
LAST CALL TO THIS PROGRAM ?
NO
1
YES
CLEAR "GSR" ENTRY IN PROCESSOR STATUS TABLE
2


2
ANY PROCESSORS ALLOCATED ?
YES
1
NO
SEARCH PROCESSOR STATUS TABLE FOR A NON-EMPTY ENTRY
FOUND ?
YES
1
NO
GENERATE REQUEST FOR "PP4"
1


CALL PROCESSOR(S) FOR A PP PROGRAM
READ IN APPROPRIATE COMMUNICATION BUFFER POINTERS
IS COMMUNICATION BUFFER EMPTY ?
YES
EXIT
NO
CALCULATE NO. OF PROCESSORS REQUIRED
AT LEAST ONE ?
NO
INCREMENT NO. OF PROCESSORS BY ONE
YES
READ IN PROCESSOR COUNT WORD
SUFFICIENT PROCESSORS ALREADY ALLOCATED ?
YES
EXIT
NO
FOR REQUIRED NO. OF PROCESSORS
GENERATE REQUEST FOR REQUIRED PROGRAM
EXIT


GENERATE REQUEST FOR REQUIRED PP PROGRAM
SEARCH PROCESSOR STATUS TABLE FOR EMPTY ENTRY
FOUND ?
NO
EXIT
YES
GENERATE PROCESSOR STATUS TABLE ENTRY FOR REQUIRED PROGRAM
GENERATE REQUEST FOR REQUIRED PROGRAM
INCREMENT APPROPRIATE ENTRY IN PP COUNT WORD
EXIT

8. PROGRAM DSA - DECLARATIVE STATEMENT ANALYSER

8.0 Introduction

Program DSA is loaded only once for a given source subprogram by program GSR when it has successfully recognised the first header card for the next subprogram, that is, a PROGRAM, FUNCTION, SUBROUTINE or BLOCK DATA statement. The program completes the compilation of all declarative statements, except statement function definition statements (detected and processed by program FEA) and ENTRY statements (processed by program MSA), passed to it via the processor 1 communication buffer by program GSR.

Once loaded, the program continually executes its major loop, processing one processor communication buffer entry each time, until the buffer is empty. It then checks whether all declaratives, which must precede the first executable statement, have been loaded into the buffer by program GSR. This condition is signalled by program GSR via the appropriate bit in the INPUT buffer parameters (see 5.1.3). If this bit is not set when checked by program DSA, it enters an idle loop after which it checks the processor 1 communication buffer once again. This is to allow for the unusual event that program GSR cannot fill the buffer faster than program DSA can extract and process statements from it. However, if the first executable statement has been detected and the corresponding bit set by program GSR, program DSA will check the state of the communication buffers for processors 2 and 3, which correspond to programs FEA and MSA respectively. If either

or both these buffers are non-empty, this program will generate the necessary requests to monitor for the appropriate programs to be loaded, before dropping itself. It should be noted at this point that program PCR also performs the latter function of requesting programs FEA and MSA as necessary, provided that program DSA is not still loaded and running at the time. Thus, it is ensured that no executable statements will be processed before the processing of all declaratives is completed. This is, of course, most important if any of the first few statements following the declaratives reference names which appear in declaratives.

The case of a BLOCK DATA subprogram is different since it cannot validly contain any executable statements. Therefore when the executable statement flag is set (to indicate the end of the declaratives), program DSA cycles until the end of subprogram flag (see 5.1.3) is set by program GSR to indicate that it has detected a valid END statement. Program DSA then calls the windup routine, program PP4, before dropping itself.

8.1 Explanatory Notes to the Flow-Chart

Although not explicitly stated in the accompanying flow-chart, all references to the processor status and error tables, as well as the central memory table full flag word, are made only after access to the required table has been successfully requested, using the procedure described in 5.1.0. However, references to the name table are not made via this table accessing mechanism. This is because only program GSR, which does not reference the name table, can run concurrently with this program.

The following notes refer to the flow-chart:

I. The required entry number in the processor status table is passed to this program by program GSR in the last byte of this peripheral processor's input register (see 2.1.2). This number, which is always 2 for program DSA, is also used as the program's processor number in all central memory table access requests.

II. The last byte of the central memory table full flag word (see 5.1.11) provides this information.

III. See 5.1.11.

IV. This program will be recalled by program CTM when it has completed the expansion of the central memory tables.

V. The main-list entry from which the string was just extracted must now be converted to a free-list entry and linked into the free-list chain. However, in order to reduce fragmentation of the buffer free-space, this free-list entry is merged with the preceding one if they occupy adjacent areas in the buffer.

VI. All type declarative statements are processed in the same way, except that the variable type bits are, of course, different. These bits are extracted from a table, indexed by a function of the statement type.

VII. If a name specified in an EXTERNAL statement was previously defined, that previous definition must not have been other than a type declaration without dimensions.

VIII. The header statement PROGRAM may optionally have a list of parameters separated by commas and enclosed in parentheses. A parameter may be either a file-name or two file-names separated by =.

IX. The "appropriate error" is no. 1 for a SUBROUTINE header and no. 40 for a FUNCTION header.

X. There are sixty-one entries in the COMMON block name table (see 5.1.13), since this is the maximum number of COMMON blocks that can be defined. Therefore, if the table is full when an attempt is made to insert a new entry, too many COMMON blocks are being defined and error no. 7 is flagged accordingly.

XI. The COMMON bits indicate whether the variable is defined in blank COMMON or labelled COMMON.

XII. The syntactic and semantic validity of EQUIVALENCE statements are checked concurrently. The logic for the latter is expressed in the flow-chart and is further clarified in the next note.

XIII. All variables in the same equivalence group are linked together, by means of the equivalence pointer (see 5.3.5), into a circular linked list, for example:

EQUIVALENCE (A,B,C), (D,E)

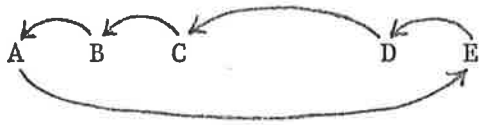
generates the following two circular linked lists:



and the additional statement:

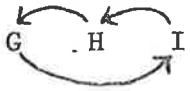
EQUIVALENCE (A,D)

merges the two lists into a single circular linked list, as follows:



As a further example, consider the effect of the following three statements:

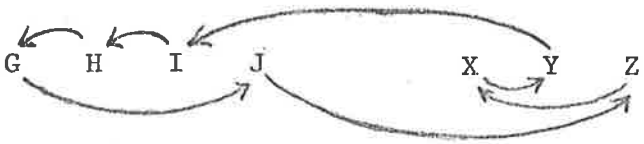
EQUIVALENCE (G,H,I)



EQUIVALENCE (G,J), (X,Y)



EQUIVALENCE (J,Y,Z)



The algorithm to generate these circular linked lists uses two pointers, as follows:

FIRST - a pointer to the name table entry corresponding to the first variable in an equivalence group.

PREVIOUS - a pointer to the name table entry corresponding to the currently last variable in the equivalence group for which a circular linked list is being progressively generated.

These two pointers are then manipulated as follows:

(a) For the first variable in an equivalence group, set FIRST to its name table entry pointer.

If the variable was previously equivalenced, set PREVIOUS equal to its equivalence pointer otherwise set PREVIOUS to FIRST.

(b) For subsequent variables,

if the variable was not previously equivalenced, its equivalence pointer is set to PREVIOUS and then PREVIOUS is set to the variable's name table pointer otherwise PREVIOUS and the variable's equivalence pointer are interchanged.

(c) When the equivalence group terminator is detected, set the equivalence pointer of the variable given by FIRST to PREVIOUS, after which the circular linked list is completely generated.

XIV. Since not more than one variable in an equivalence group can be explicitly defined in COMMON, a flag is used to indicate whether or not a COMMON variable has already been detected in the group.

XV. If a variable in an equivalence group is defined in COMMON, all variables in the same equivalence group are also regarded as being defined in COMMON.

XVI. An open parenthesis in the context of the value list of a DATA statement is treated as the start of a complex constant. A check is made that this character is followed by two real constants separated by a comma and terminated by a close parenthesis.

XVII. The syntax of a number is checked by the use of a state table having the following states:

1 Among the digits before the decimal point (if any)

2 Following the decimal point

3 Among the digits following the decimal point

4 Following the exponential symbol E or D

5 After the exponent sign (if any)

6 Among the exponent digits.

The state table below indicates the new state and the action to be taken according to the current state and the nature of the next character in the source string.

next source string character

state	digit	.	E D	+ -	Ø	B
1	1A	2B	4B	C	C	F
2	3E	*	4E	D	D	*
3	3E	*	4E	D	D	*
4	6E	*	*	5E	*	*
5	6E	*	*	*	*	*
6	6E	*	*	D	D	*

where Ø implies any other character.

The actions specified are:

A Count digits, check for occurrence of octal digits and continue.

B Set constant type to real and continue.

C Exit with a valid integer constant.

D Exit with a valid real constant (or a double precision constant if exponential symbol D occurred).

E Do nothing and continue.

F Exit with an octal constant, if only octal digits occurred and no more than twenty such digits specified.

* Exit with an error.

The same state table approach is used in the syntax checking of numbers in FORTRAN expressions by program FEA.

XVIII. The implicit variable type is generated according to the first letter of the name, that is, names beginning with I, J, K, L, M or N are integer whilst the rest are type real.

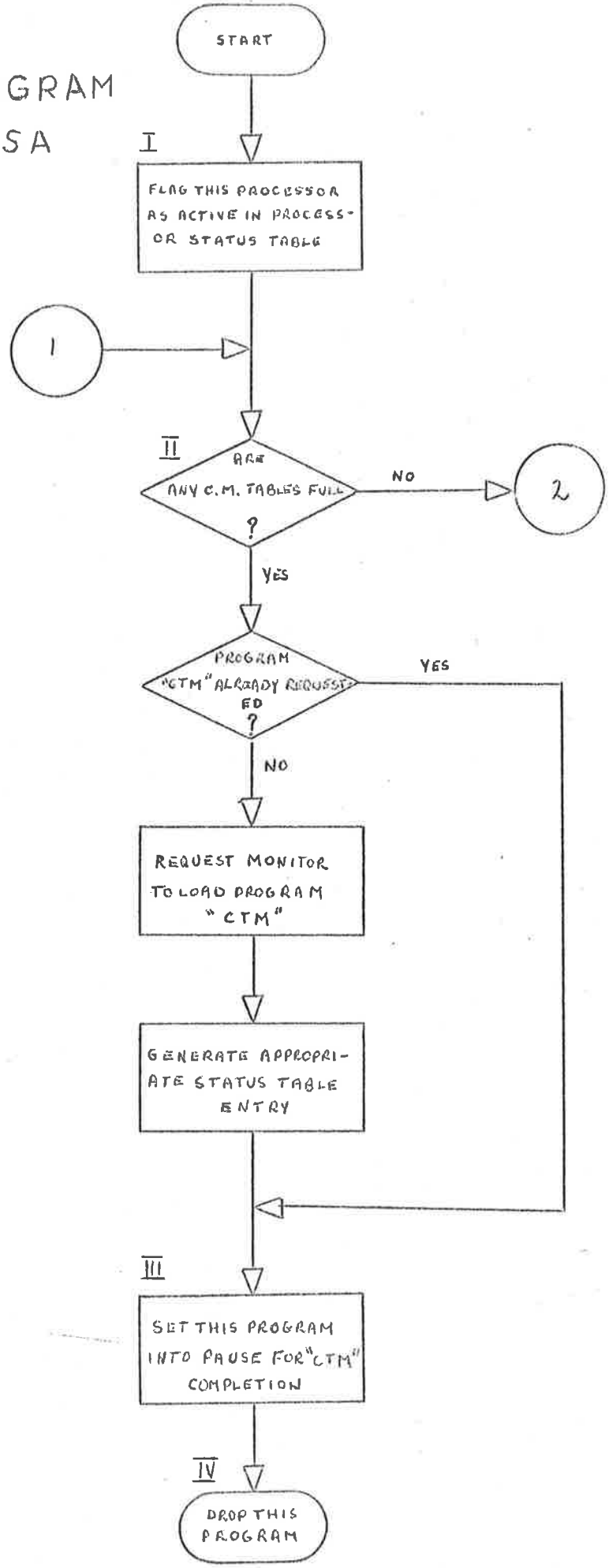
XIX. This program cannot finally drop itself until program GSR has inserted the last declarative statement into the processor 1 communication buffer. Program GSR indicates that this has occurred by setting the executable statement flag as soon as it recognises the first executable statement. Program DSA executes

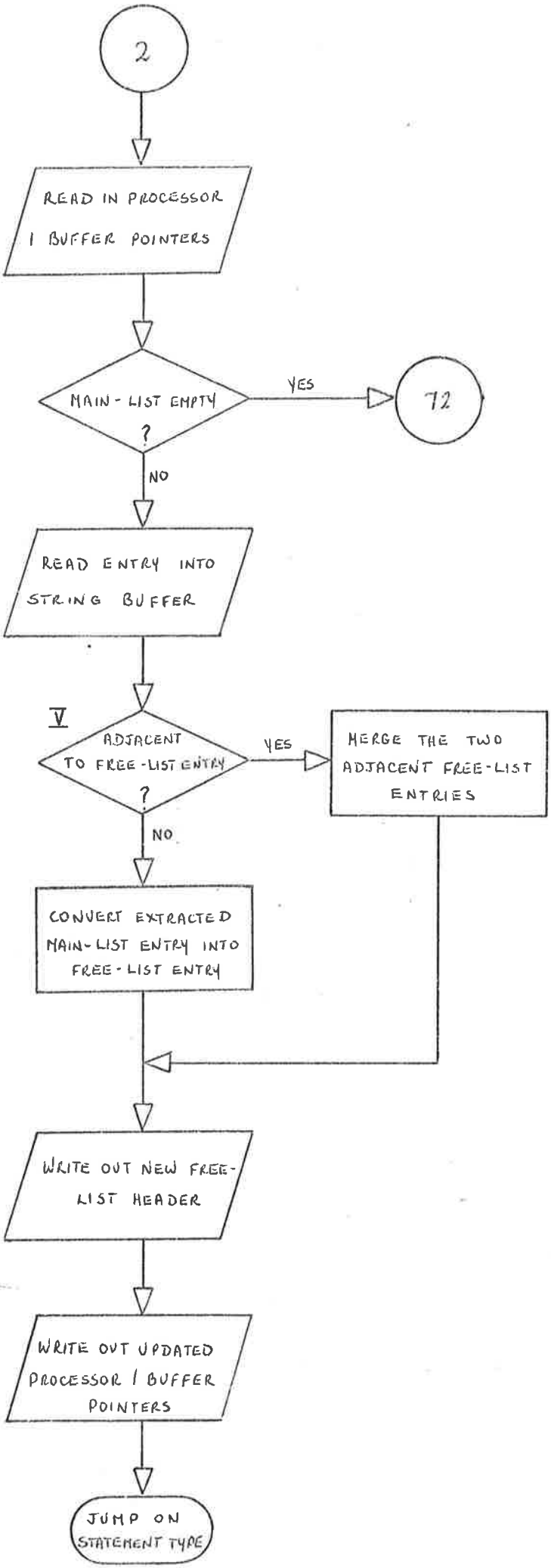
an idle loop if this flag is not yet set, to allow program GSR time to complete its processing of the next (if any) declarative statement before again attempting to extract an entry from its communication buffer. In this context, statement function definition statements are regarded as executable statements.

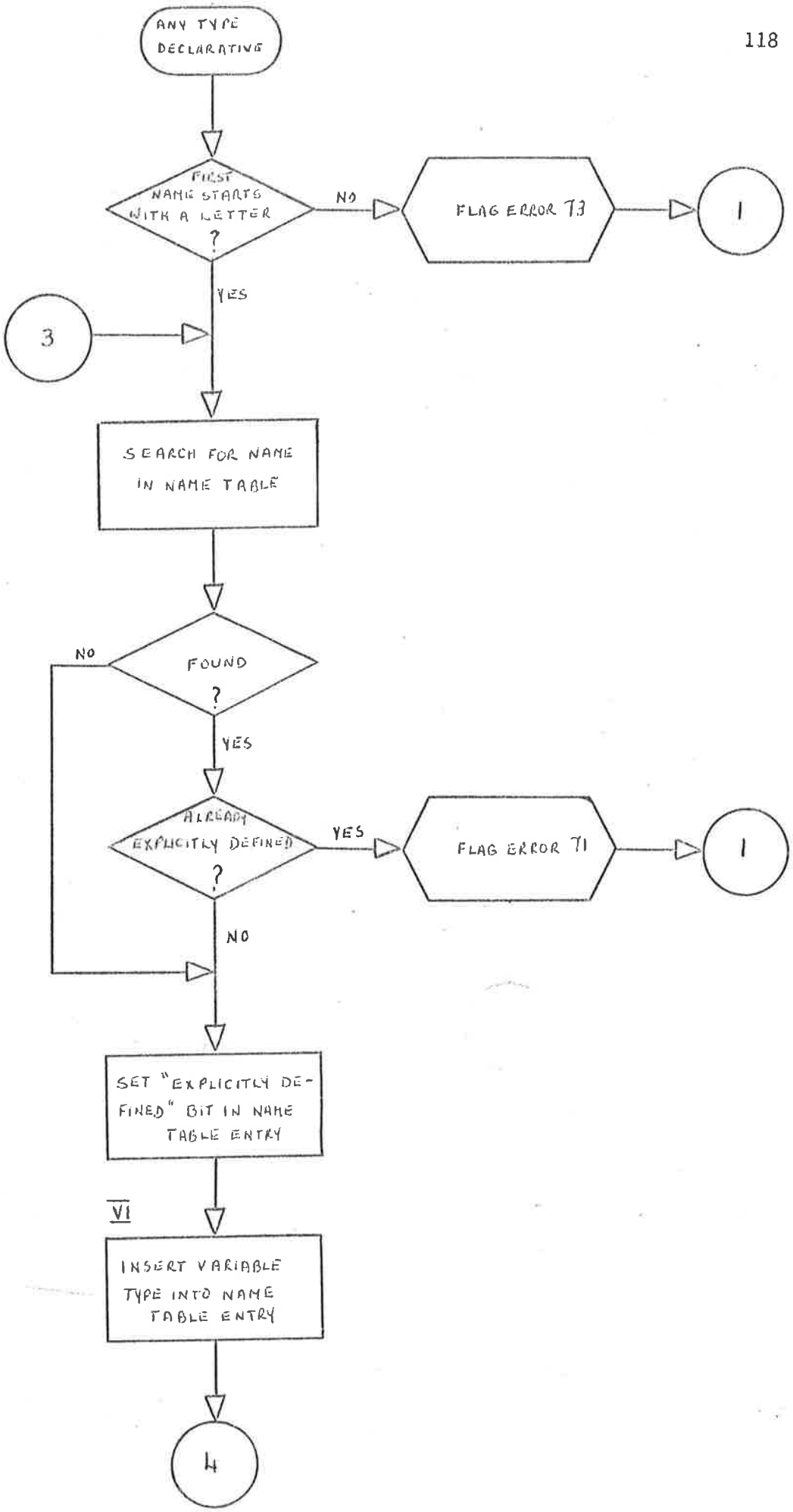
XX. Requests are made to the monitor to load programs FEA and/or MSA if their respective communication buffers are non-empty. The procedure used is the same as that in program PCR and is shown in the flow-chart to Chapter 7.

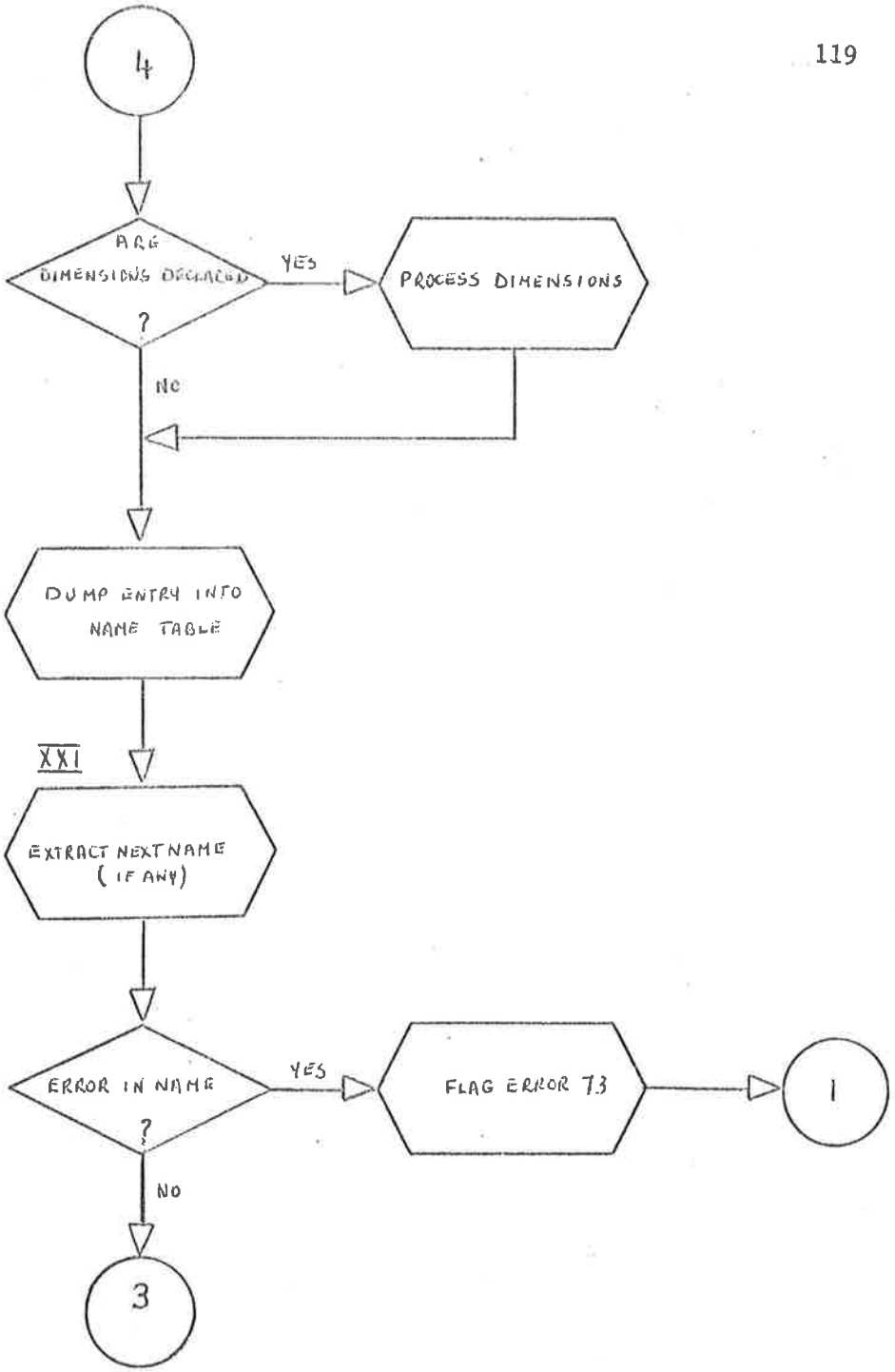
XXI. The next FORTRAN name is extracted from the source string unless the string terminator is detected first, in which case processing continues at continuation symbol 1.

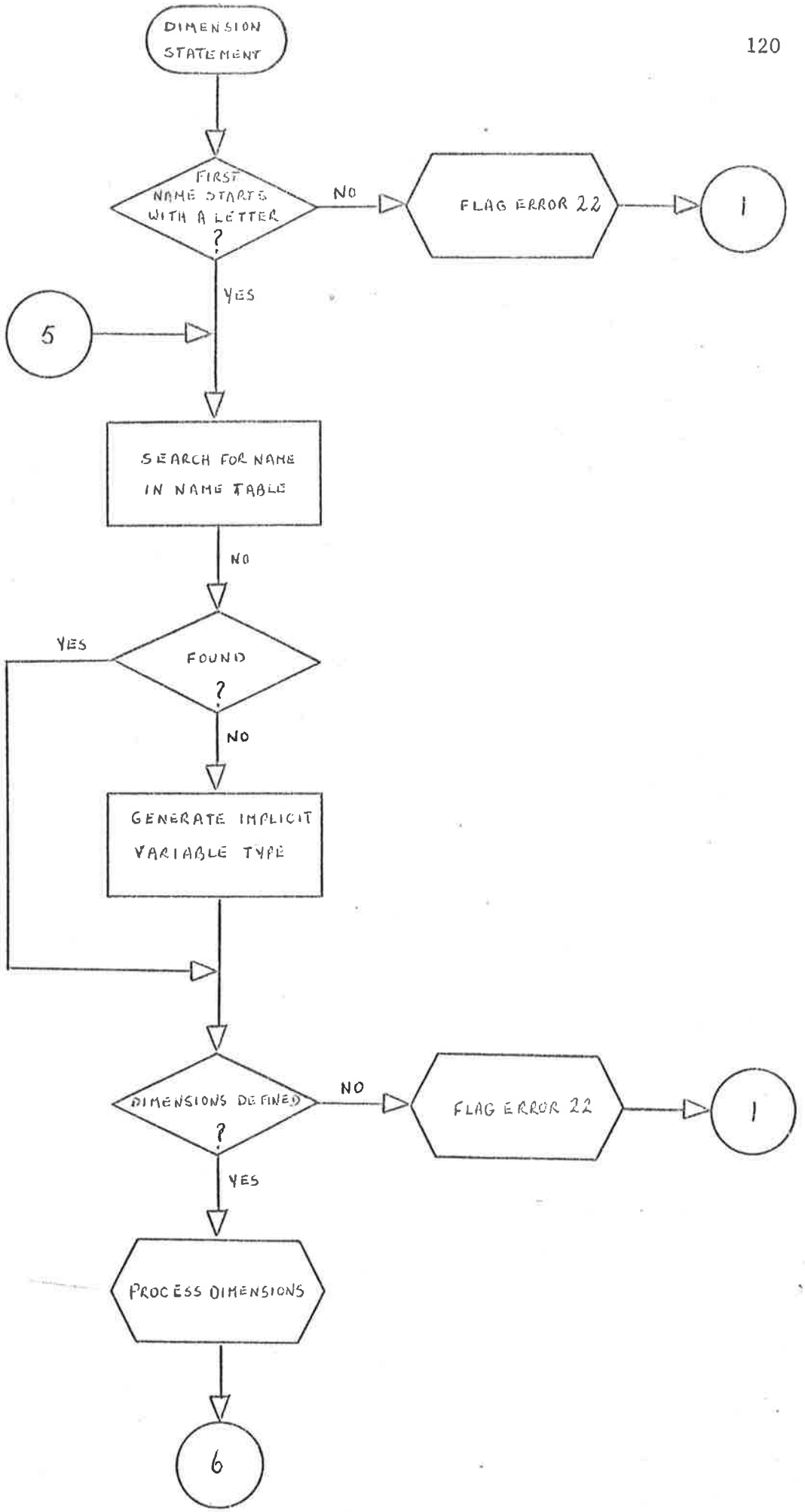
PROGRAM DSA

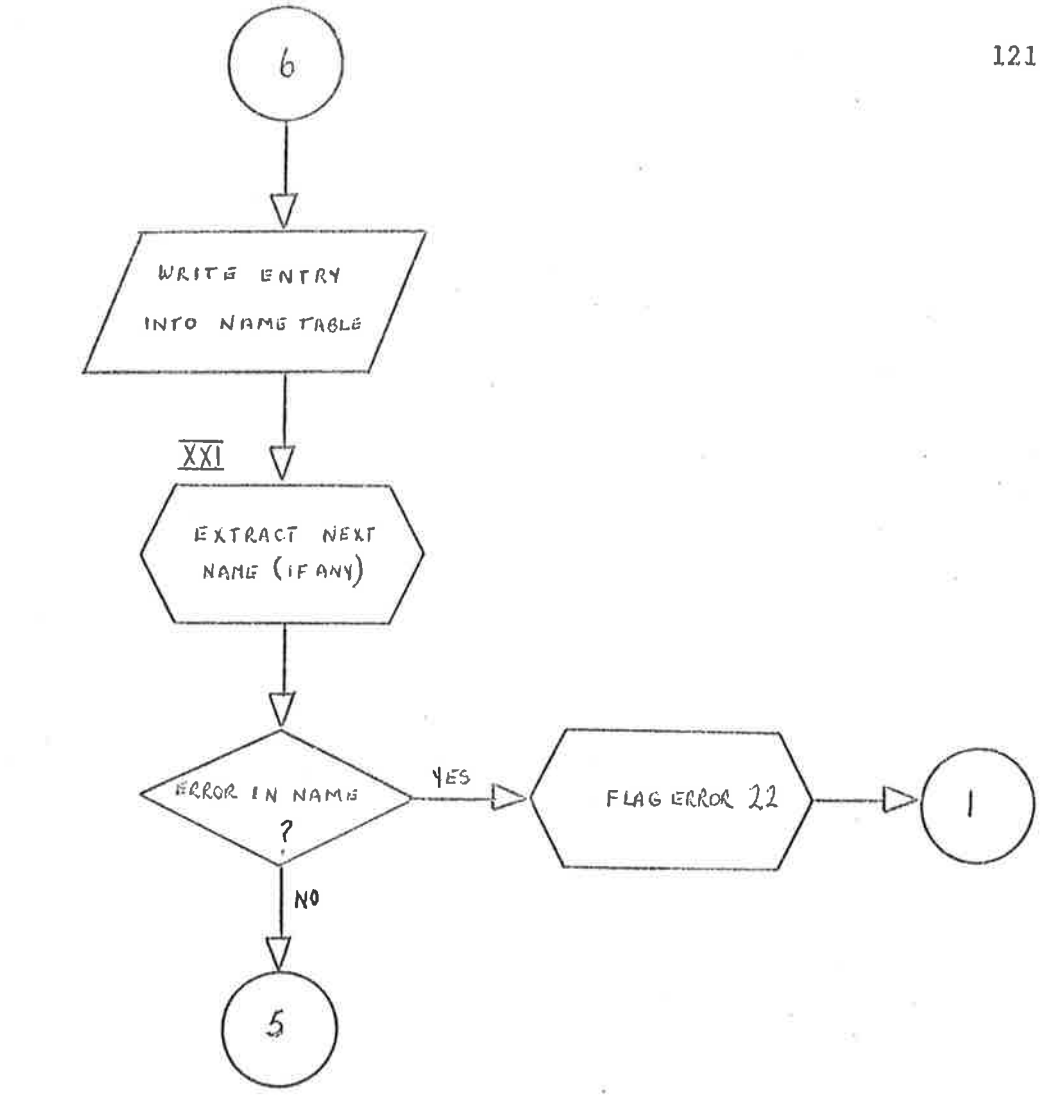

START
I
FLAG THIS PROCESSOR AS ACTIVE IN PROCESSOR STATUS TABLE
1
II
ARE ANY C.M. TABLES FULL ?
NO
2
YES
PROGRAM "CTM" ALREADY REQUESTED ?
YES
NO
REQUEST MONITOR TO LOAD PROGRAM "CTM"
GENERATE APPROPRIATE STATUS TABLE ENTRY
III
SET THIS PROGRAM INTO PAUSE FOR "CTM" COMPLETION
IV
DROP THIS PROGRAM

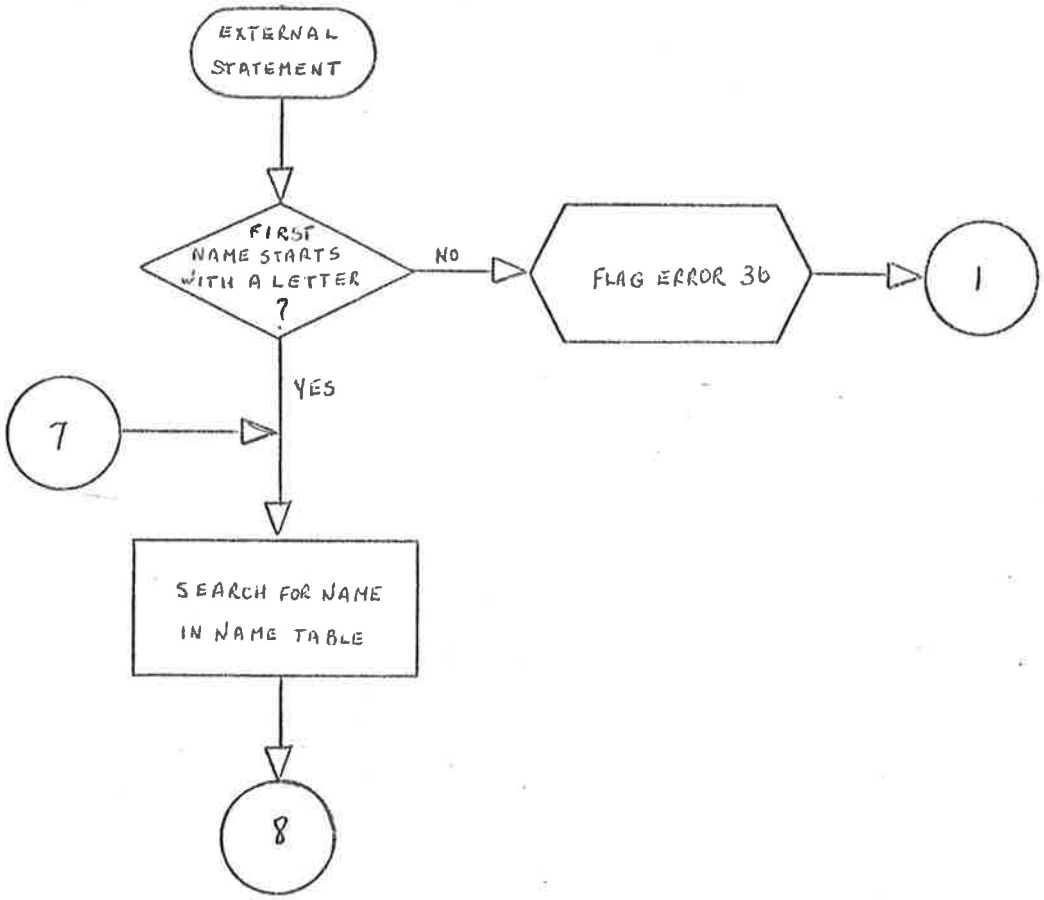

2
READ IN PROCESSOR 1 BUFFER POINTERS
MAIN-LIST EMPTY ?
YES
72
NO
READ ENTRY INTO STRING BUFFER
V
ADJACENT TO FREE-LIST ENTRY ?
YES
MERGE THE TWO ADJACENT FREE-LIST ENTRIES
NO
CONVERT EXTRACTED MAIN-LIST ENTRY INTO FREE-LIST ENTRY
WRITE OUT NEW FREE-LIST HEADER
WRITE OUT UPDATED PROCESSOR 1 BUFFER POINTERS
JUMP ON STATEMENT TYPE

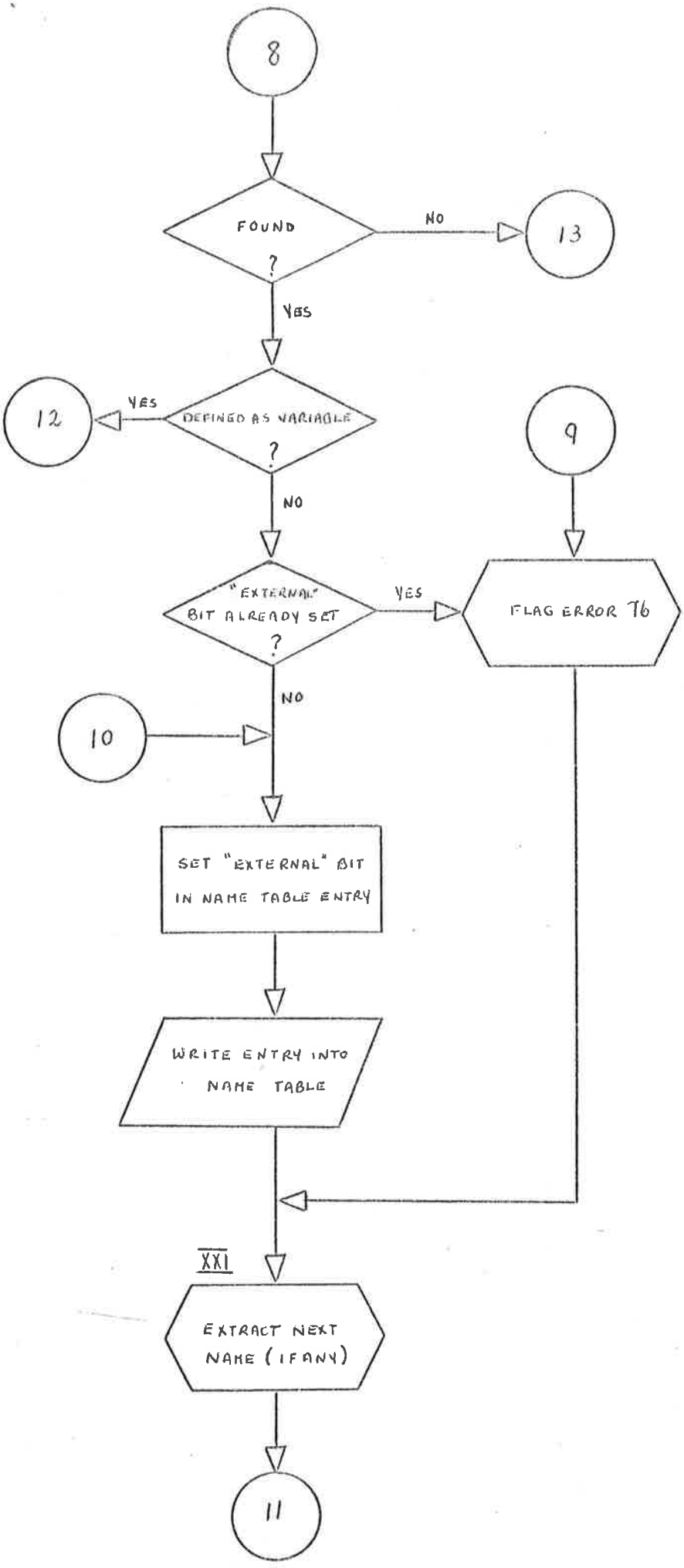

ANY TYPE DECLARATIVE
FIRST NAME STARTS WITH A LETTER ?
NO
FLAG ERROR 73
1
YES
3
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
YES
ALREADY EXPLICITLY DEFINED ?
YES
FLAG ERROR 71
1
NO
SET "EXPLICITLY DEFINED" BIT IN NAME TABLE ENTRY
VI
INSERT VARIABLE TYPE INTO NAME TABLE ENTRY
4

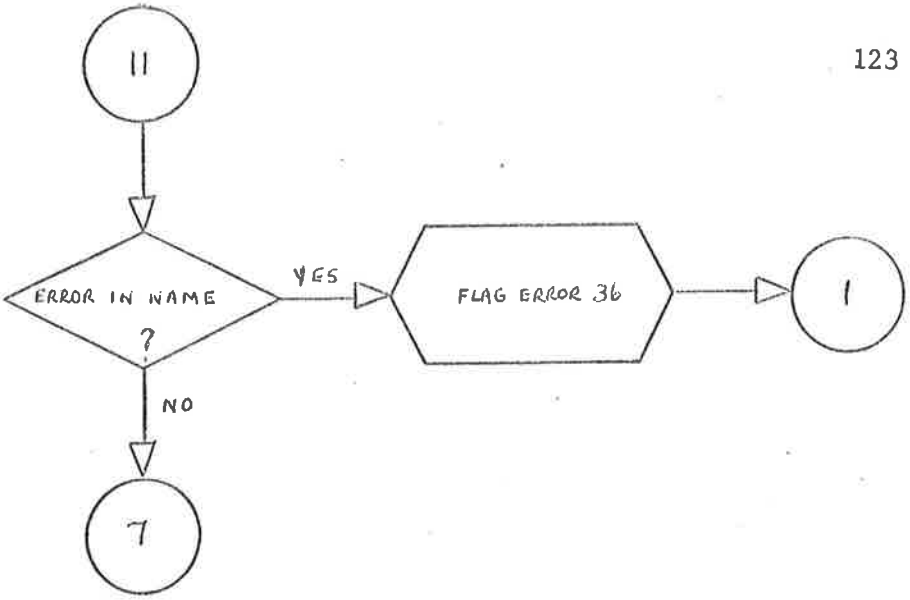

4
ARG DIMENSIONS DECLARED ?
YES
PROCESS DIMENSIONS
NO
DUMP ENTRY INTO NAME TABLE
XXI
EXTRACT NEXT NAME (IF ANY)
ERROR IN NAME ?
YES
FLAG ERROR 73
1
NO
3

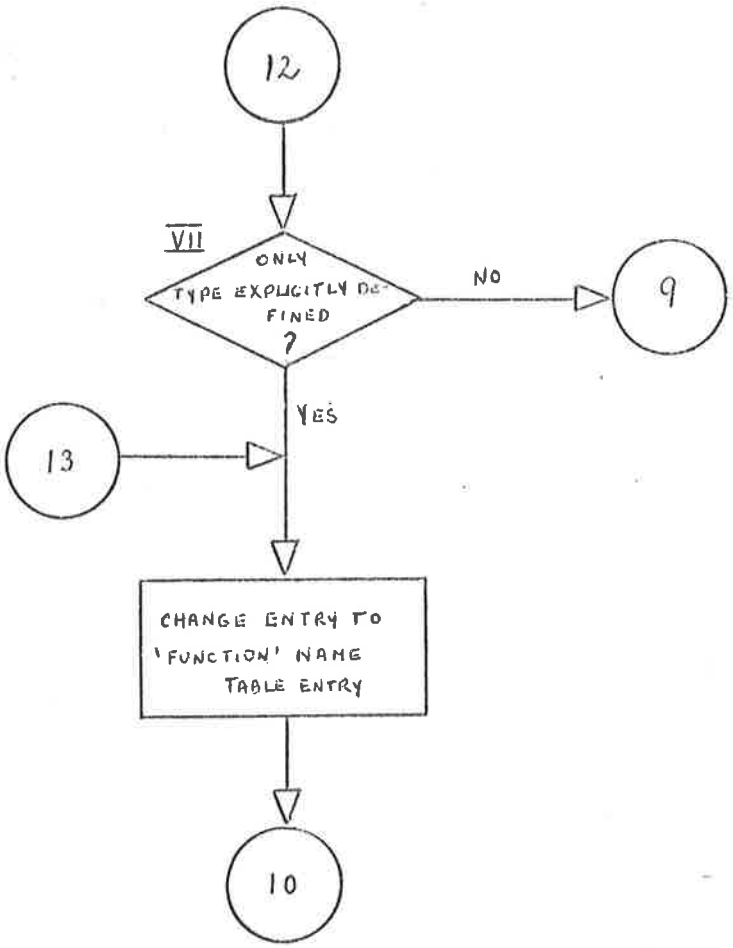

DIMENSION STATEMENT
FIRST NAME STARTS WITH A LETTER ?
NO
FLAG ERROR 22
1
YES
5
SEARCH FOR NAME IN NAME TABLE
NO
FOUND ?
YES
NO
GENERATE IMPLICIT VARIABLE TYPE
DIMENSIONS DEFINED ?
NO
FLAG ERROR 22
1
YES
PROCESS DIMENSIONS
6

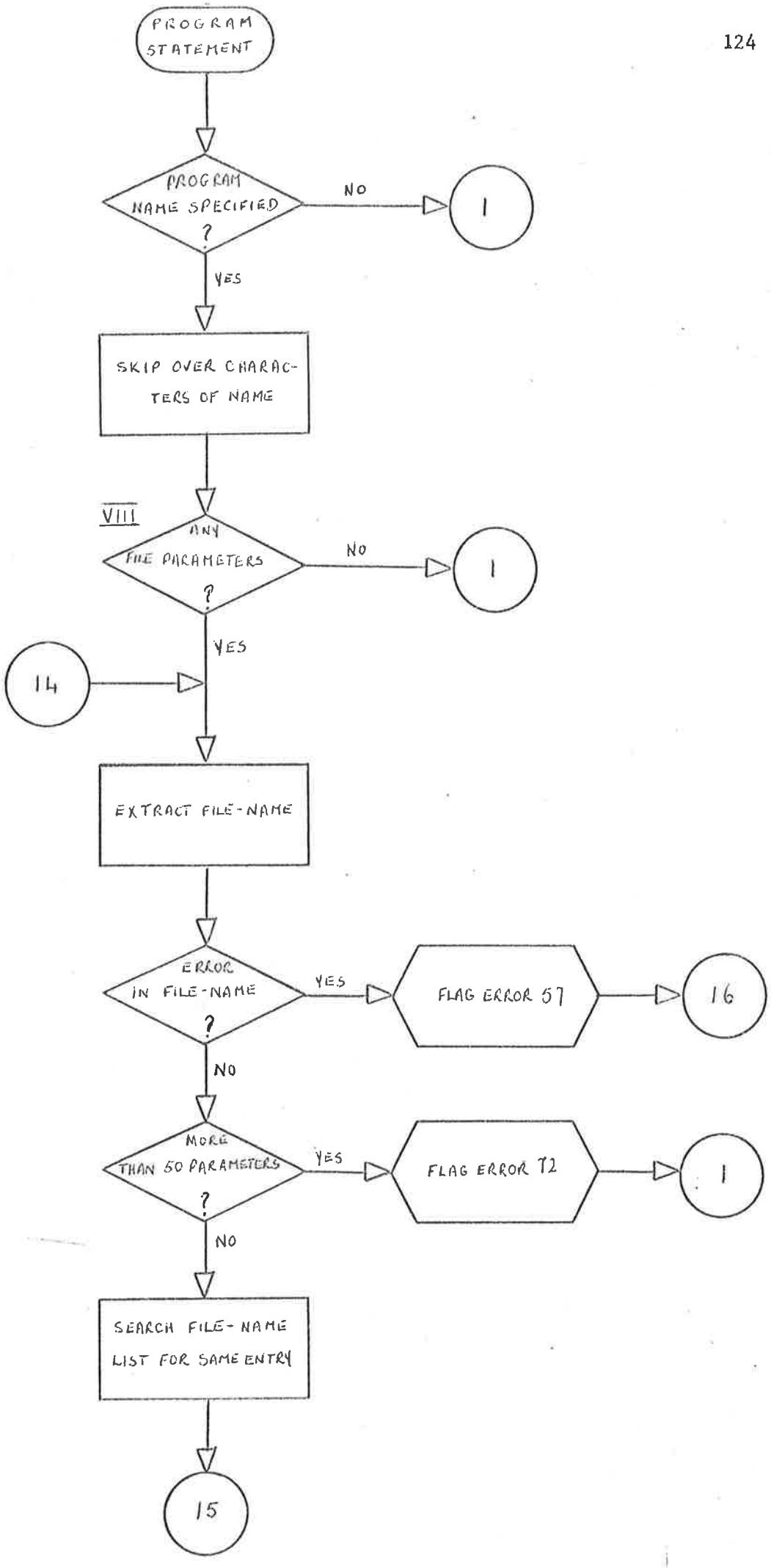

6
WRITE ENTRY INTO NAME TABLE
XXI
EXTRACT NEXT NAME (IF ANY)
ERROR IN NAME ?
YES
FLAG ERROR 22
1
NO
5

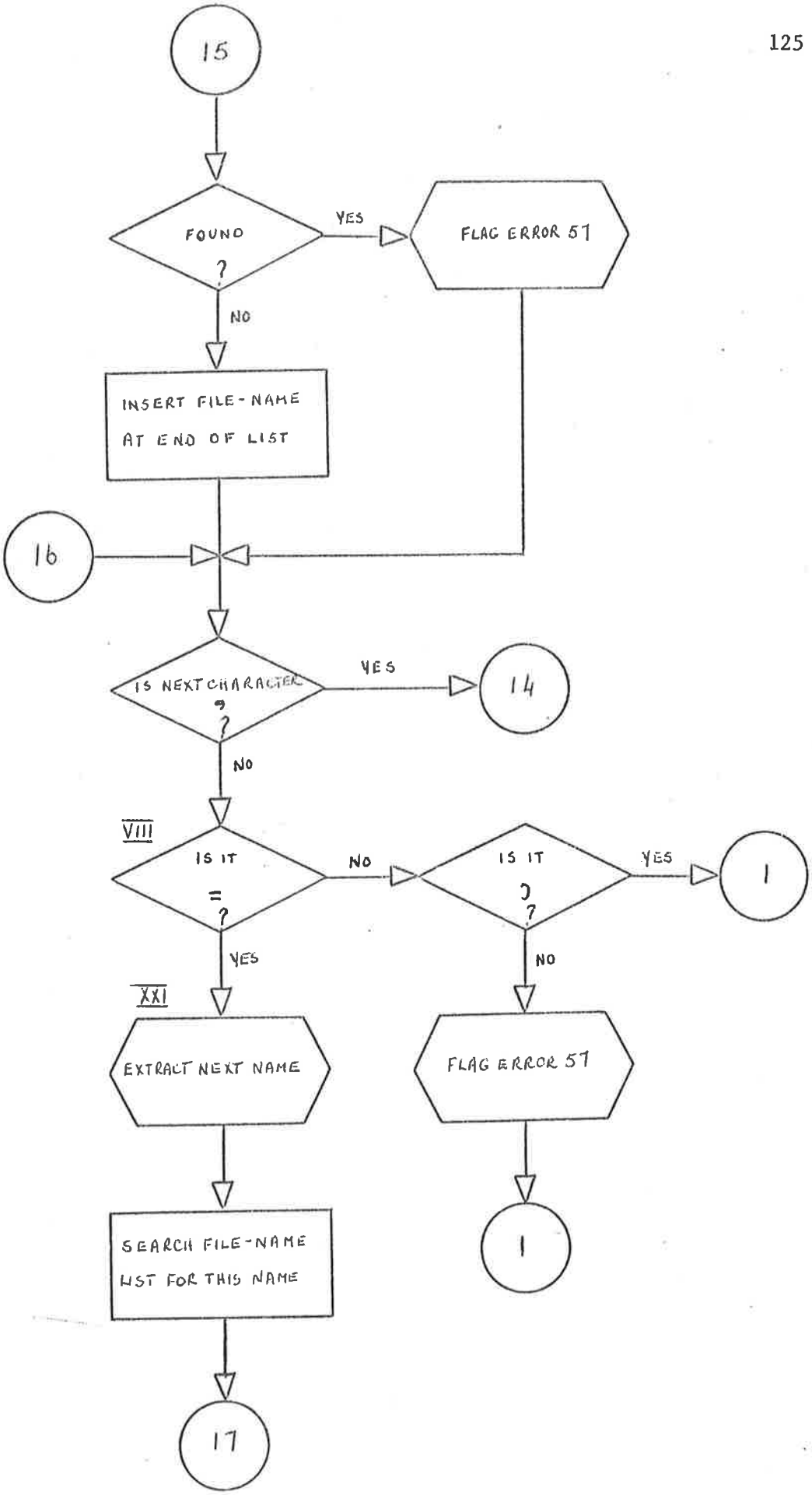

EXTERNAL STATEMENT
FIRST NAME STARTS WITH A LETTER ?
NO
FLAG ERROR 36
1
YES
7
SEARCH FOR NAME IN NAME TABLE
8

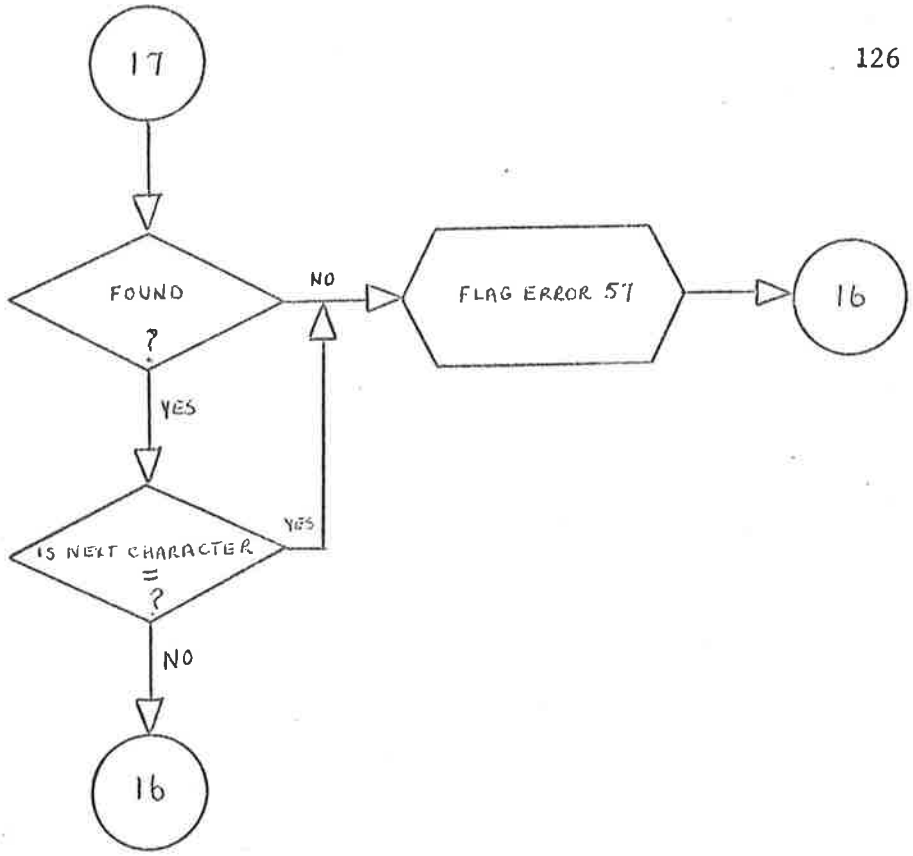

8
FOUND ?
NO
13
YES
12
YES
DEFINED AS VARIABLE ?
NO
9
"EXTERNAL" BIT ALREADY SET ?
YES
FLAG ERROR 76
NO
10
SET "EXTERNAL" BIT IN NAME TABLE ENTRY
WRITE ENTRY INTO NAME TABLE
XXI
EXTRACT NEXT NAME (IF ANY)
11

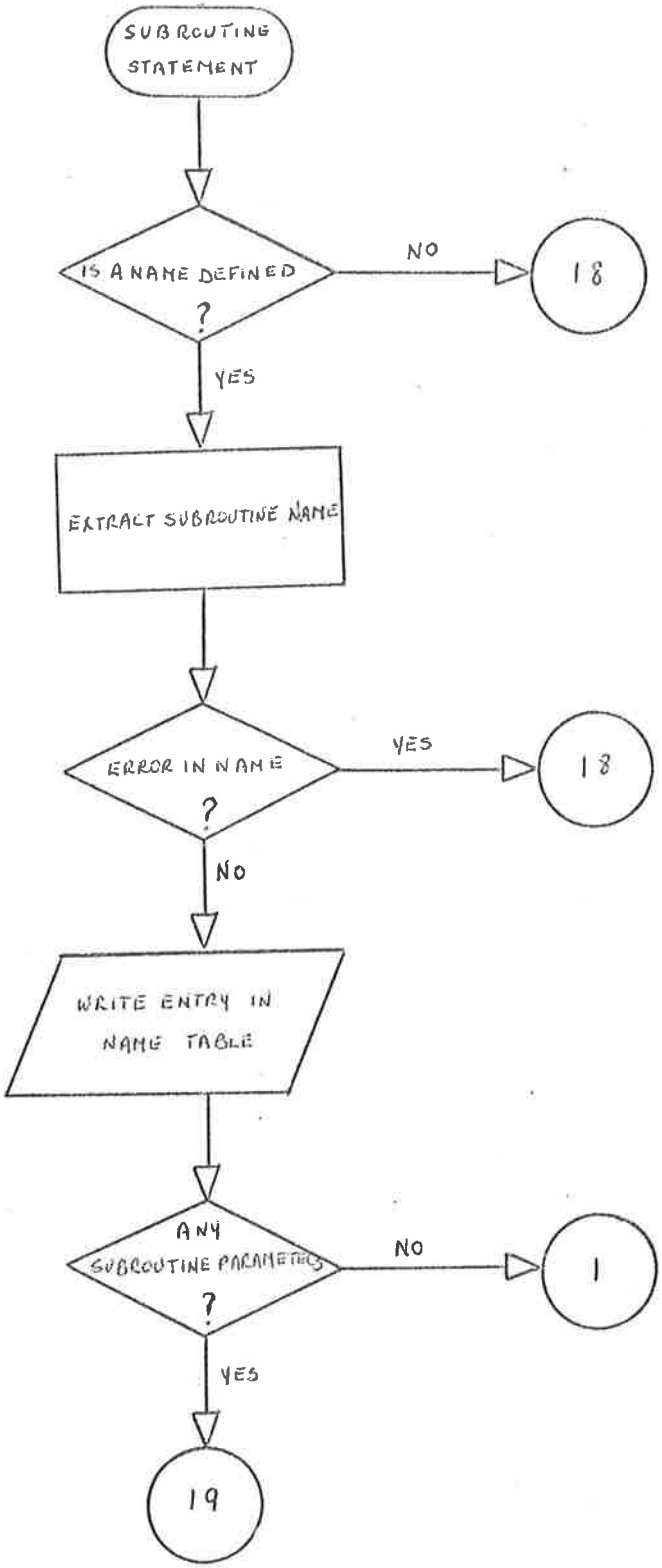

11
ERROR IN NAME ?
YES
FLAG ERROR 36
1
NO
7

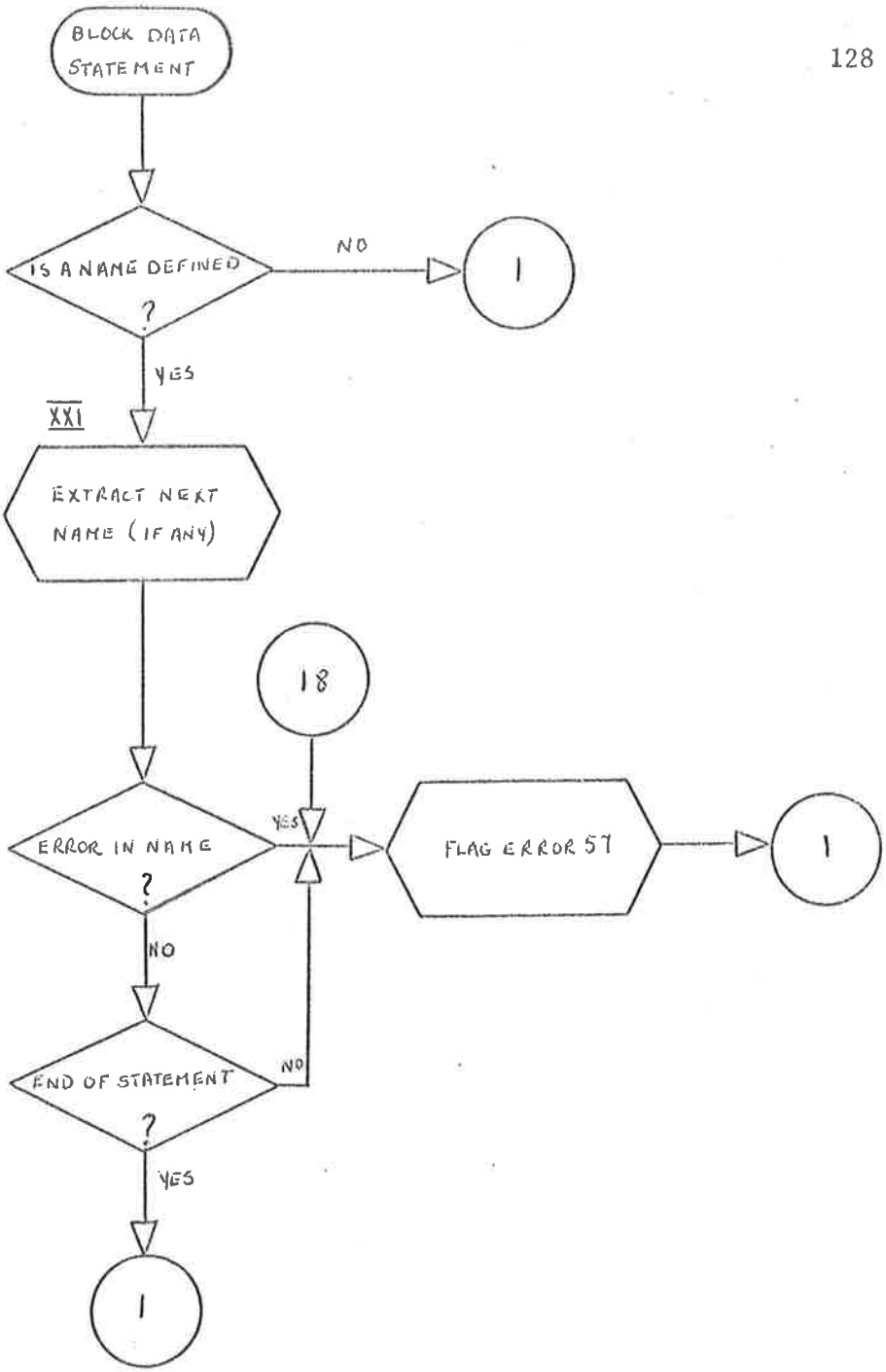

12
VII
ONLY TYPE EXPLICITLY DEFINED ?
NO
9
YES
13
CHANGE ENTRY TO 'FUNCTION' NAME TABLE ENTRY
10

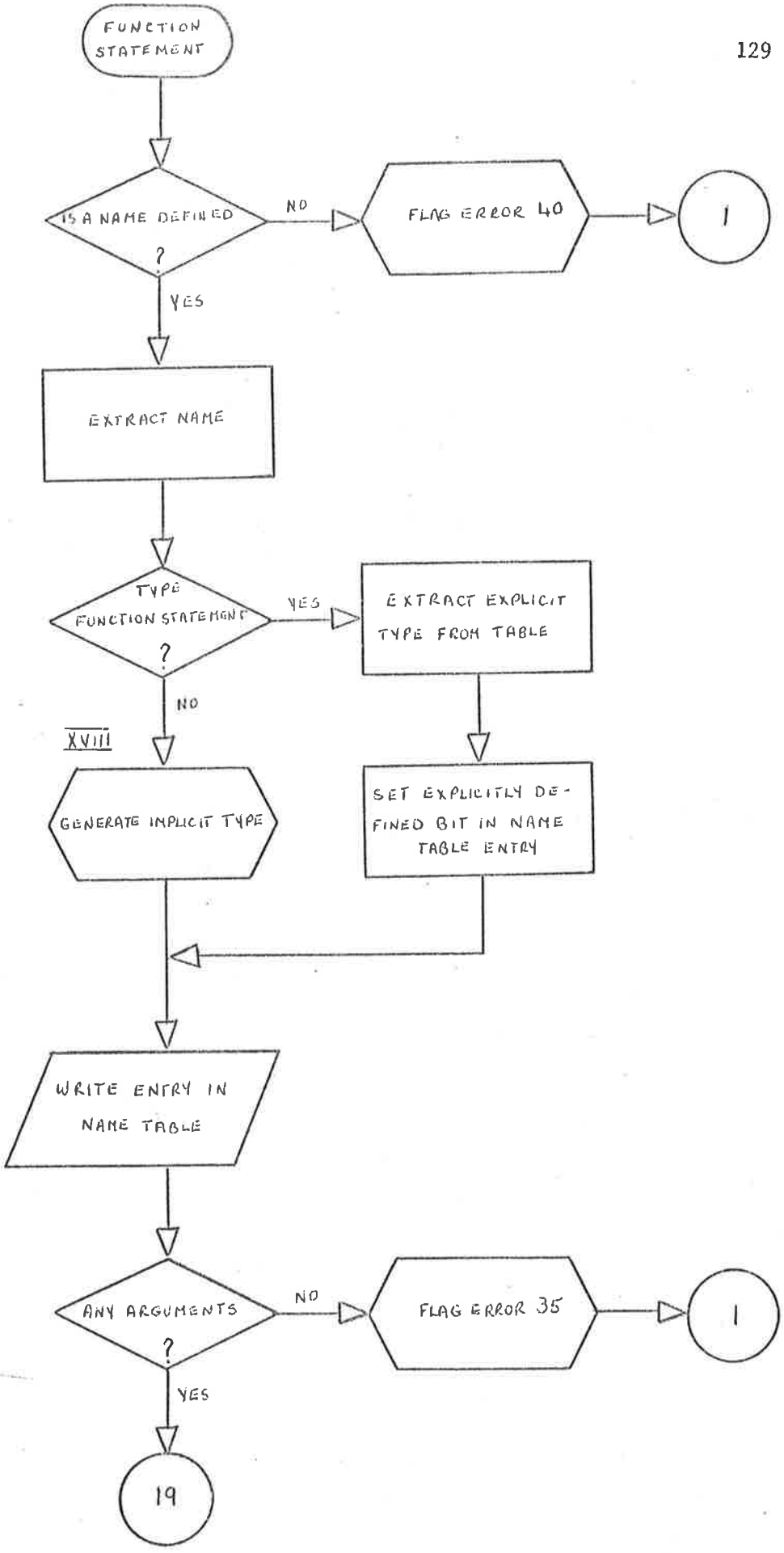

PROGRAM STATEMENT
PROGRAM NAME SPECIFIED ?
NO
1
YES
SKIP OVER CHARACTERS OF NAME
VIII
ANY FILE PARAMETERS ?
NO
1
YES
14
EXTRACT FILE-NAME
ERROR IN FILE-NAME ?
YES
FLAG ERROR 57
16
NO
MORE THAN 50 PARAMETERS ?
YES
FLAG ERROR 72
1
NO
SEARCH FILE-NAME LIST FOR SAME ENTRY
15

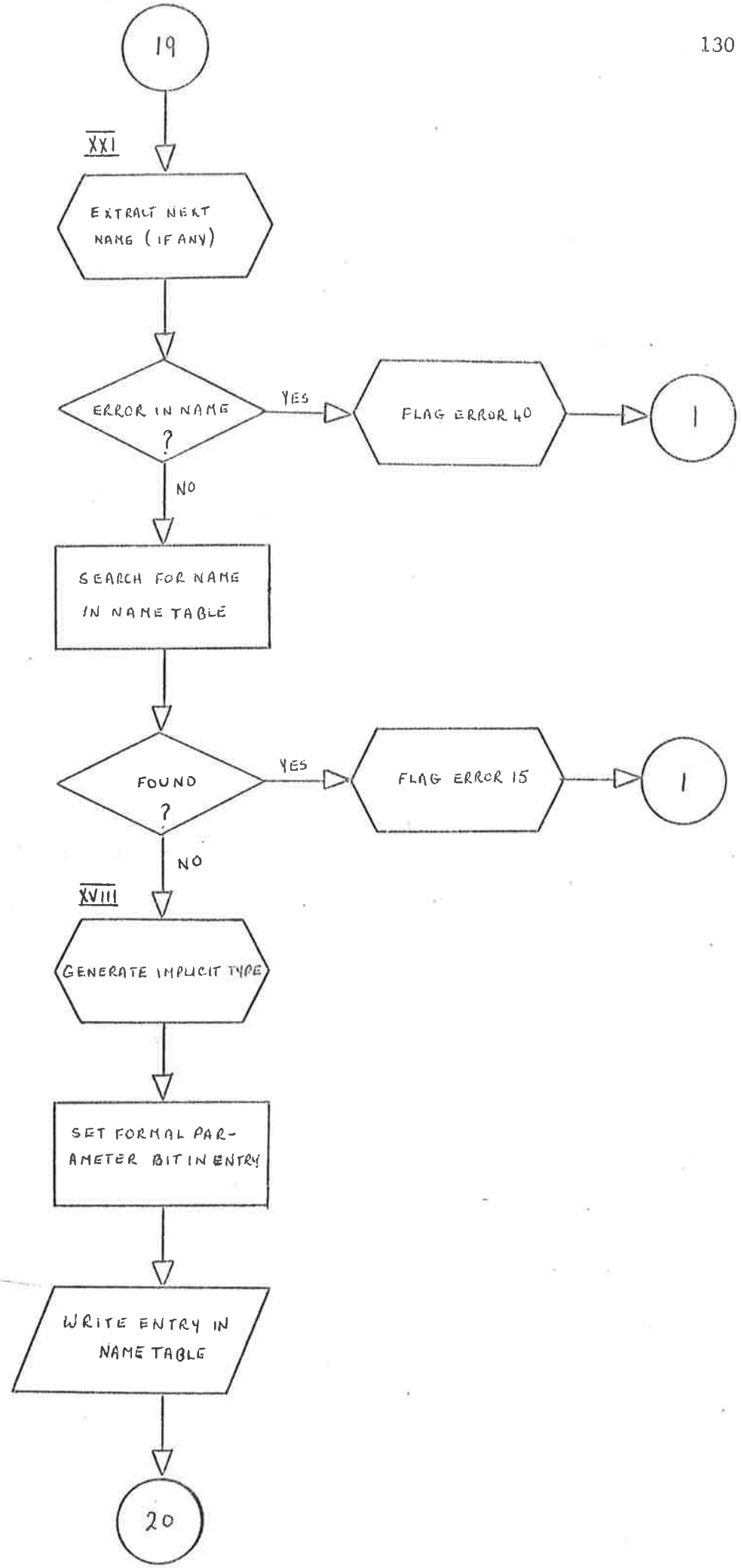

15
FOUND ?
YES
FLAG ERROR 51
NO
INSERT FILE-NAME AT END OF LIST
16
IS NEXT CHARACTER ?
YES
14
NO
VIII
IS IT = ?
NO
IS IT) ?
YES
1
YES
NO
XXI
EXTRACT NEXT NAME
FLAG ERROR 51
SEARCH FILE-NAME LIST FOR THIS NAME
1
17

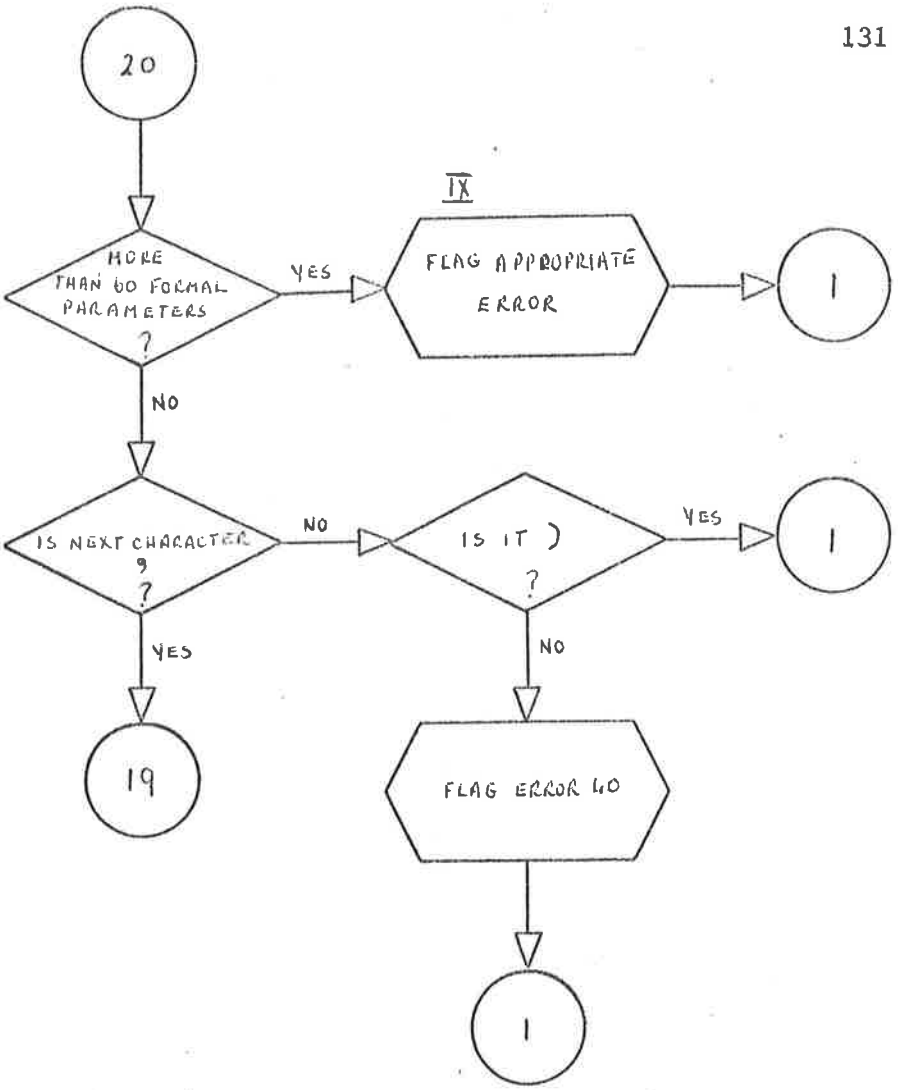

17
FOUND
?
NO
FLAG ERROR 57
16
YES
IS NEXT CHARACTER
=
?
YES
NO
16

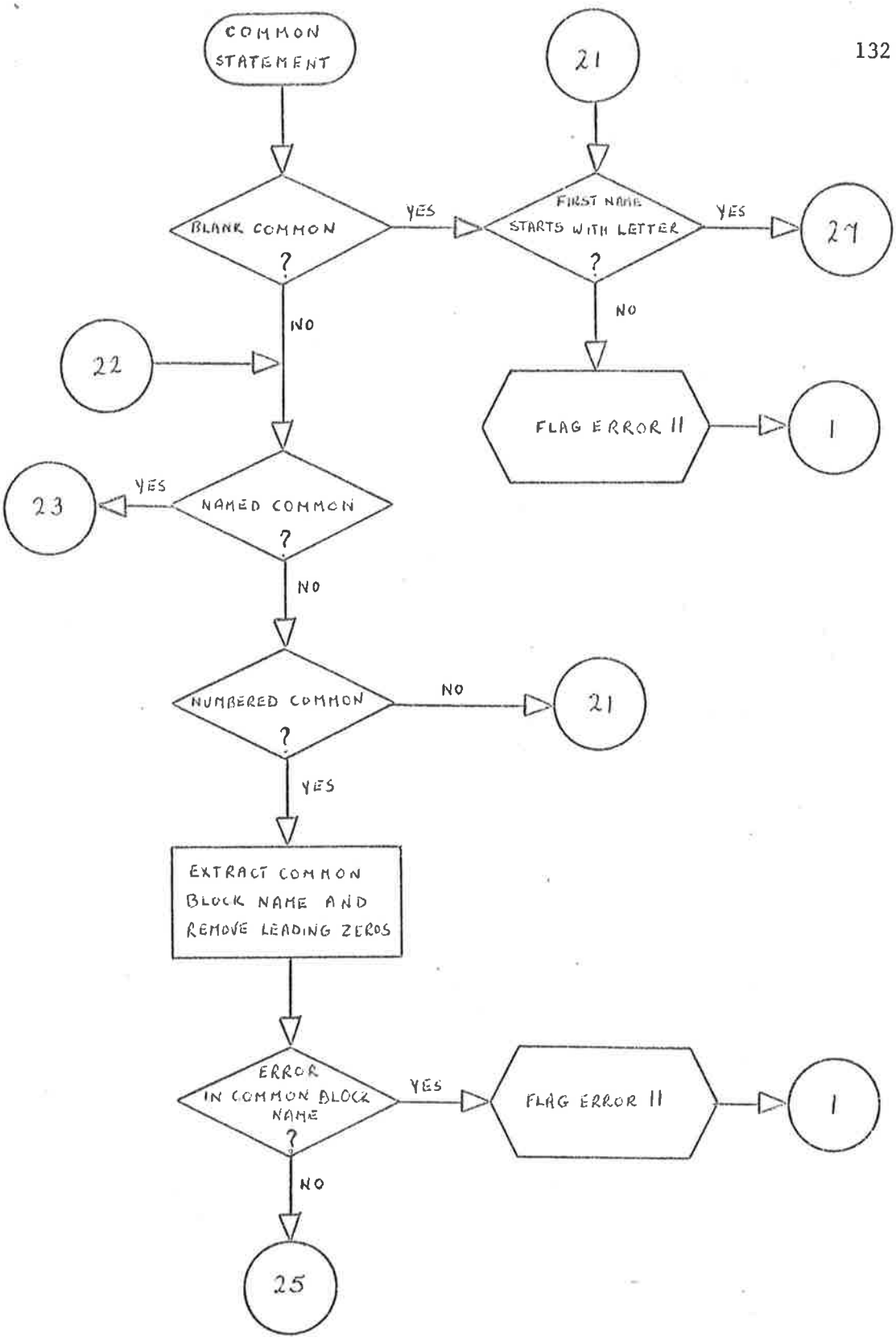

SUBROUTINE STATEMENT
IS A NAME DEFINED ?
NO
18
YES
EXTRACT SUBROUTINE NAME
ERROR IN NAME ?
YES
18
No
WRITE ENTRY IN NAME TABLE
ANY SUBROUTINE PARAMETERS ?
NO
1
YES
19

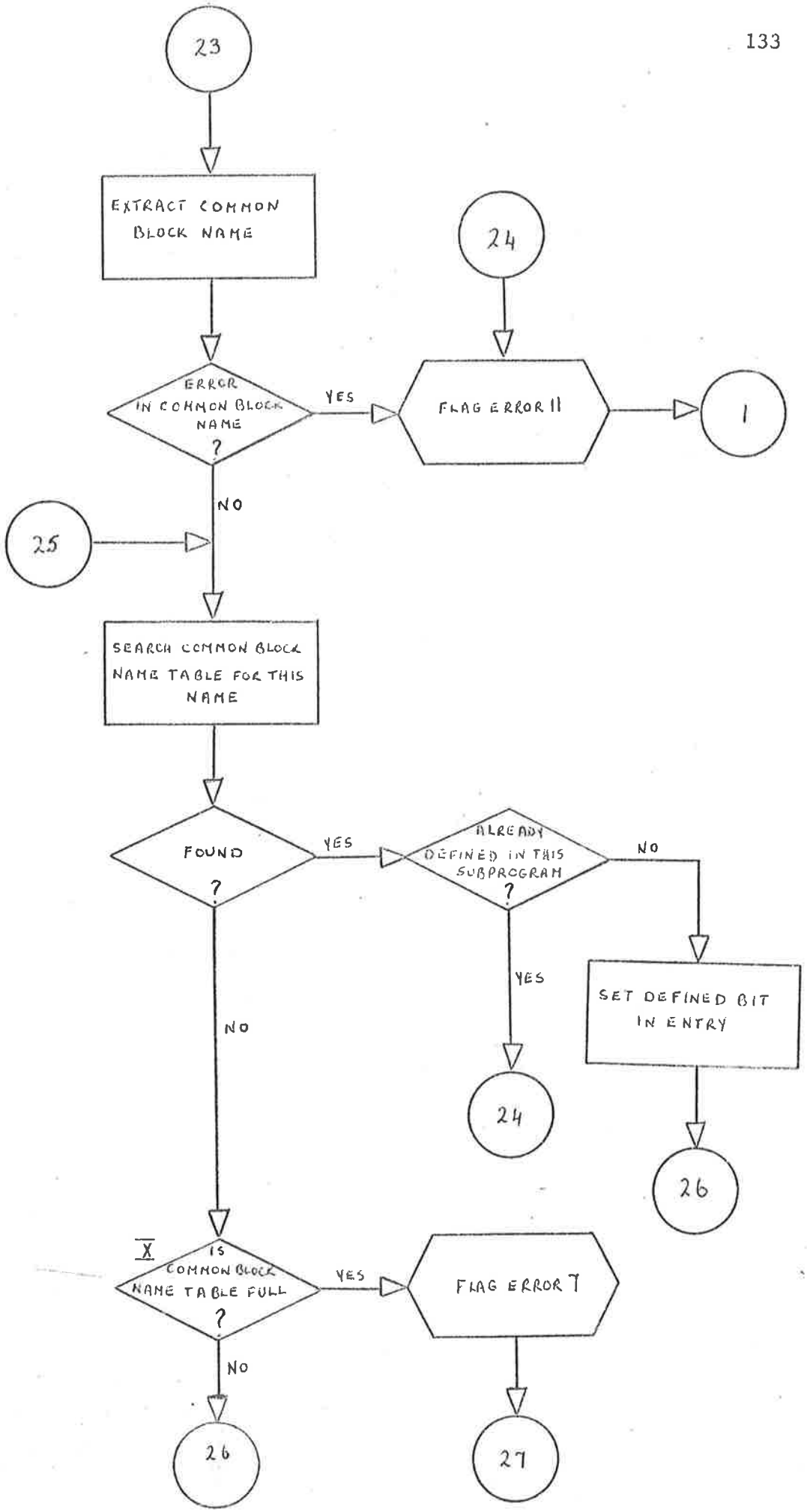

BLOCK DATA STATEMENT
IS A NAME DEFINED ?
NO
1
YES
XXI
EXTRACT NEXT NAME (IF ANY)
18
ERROR IN NAME ?
YES
FLAG ERROR 57
1
NO
END OF STATEMENT ?
NO
YES
1

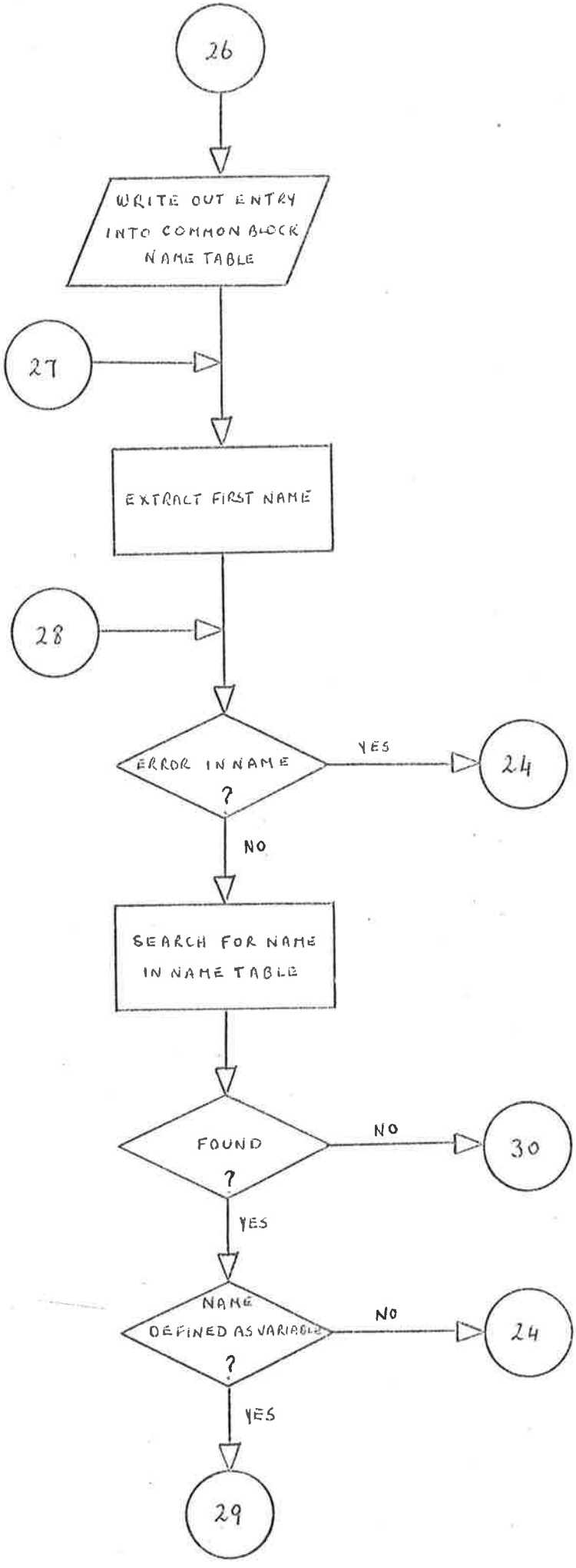

FUNCTION STATEMENT
IS A NAME DEFINED ?
NO
FLAG ERROR 40
1
YES
EXTRACT NAME
TYPE FUNCTION STATEMENT ?
YES
EXTRACT EXPLICIT TYPE FROM TABLE
NO
XVIII
GENERATE IMPLICIT TYPE
SET EXPLICITLY DEFINED BIT IN NAME TABLE ENTRY
WRITE ENTRY IN NAME TABLE
ANY ARGUMENTS ?
NO
FLAG ERROR 35
1
YES
19

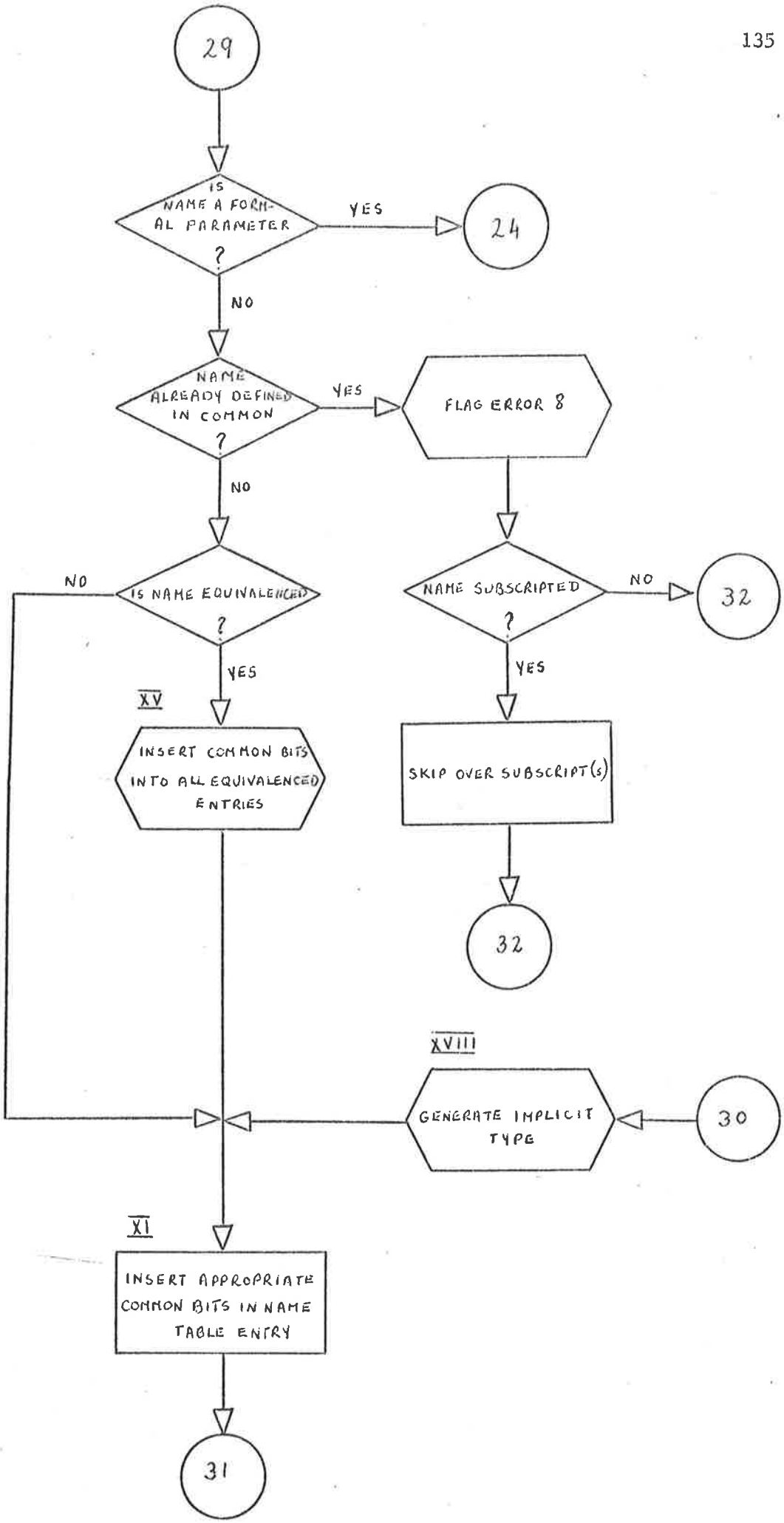

19
XXI
EXTRACT NEXT NAME (IF ANY)
ERROR IN NAME ?
YES
FLAG ERROR 40
1
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
YES
FLAG ERROR 15
1
NO
XVIII
GENERATE IMPLICIT TYPE
SET FORMAL PARAMETER BIT IN ENTRY
WRITE ENTRY IN NAME TABLE
20

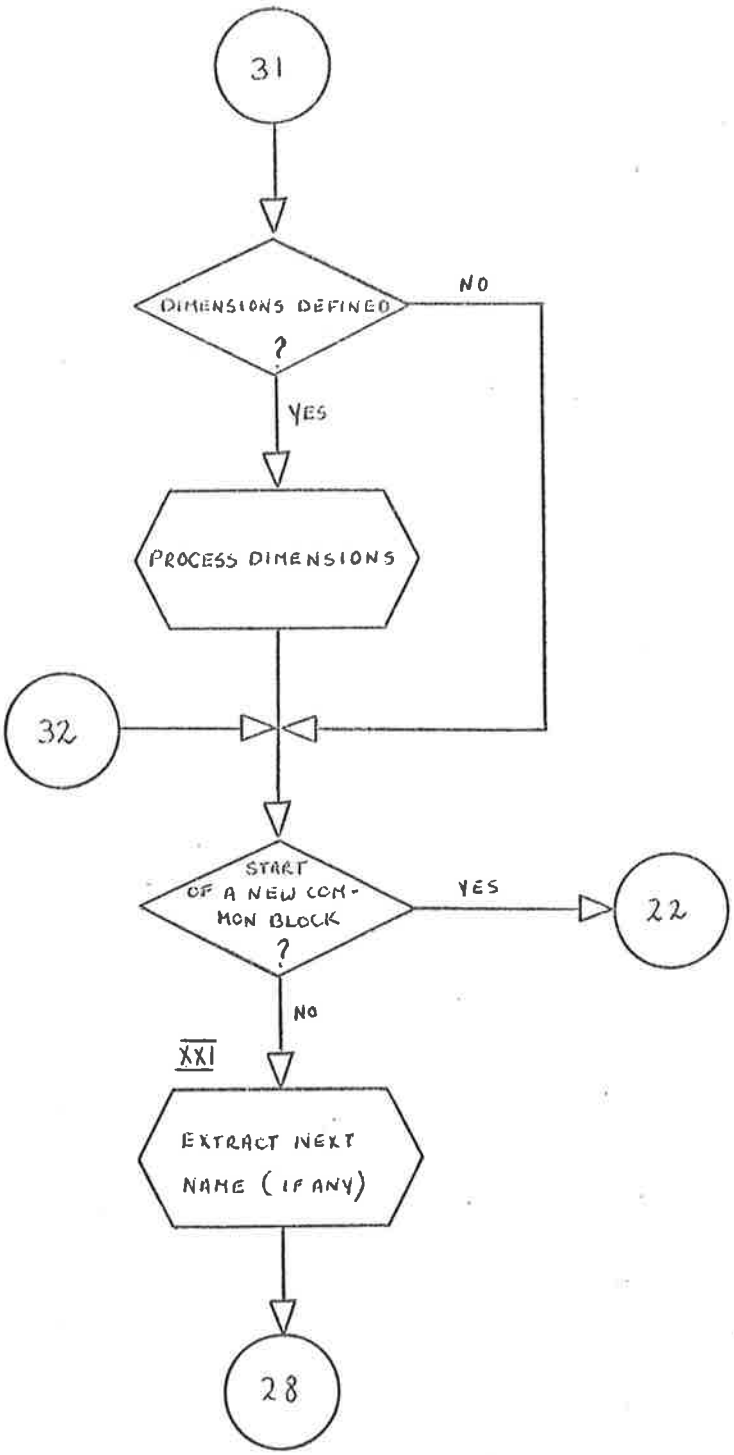

20
IX
MORE THAN 60 FORMAL PARAMETERS ?
YES
FLAG APPROPRIATE ERROR
1
NO
IS NEXT CHARACTER 9 ?
NO
IS IT) ?
YES
1
YES
NO
19
FLAG ERROR 40
1

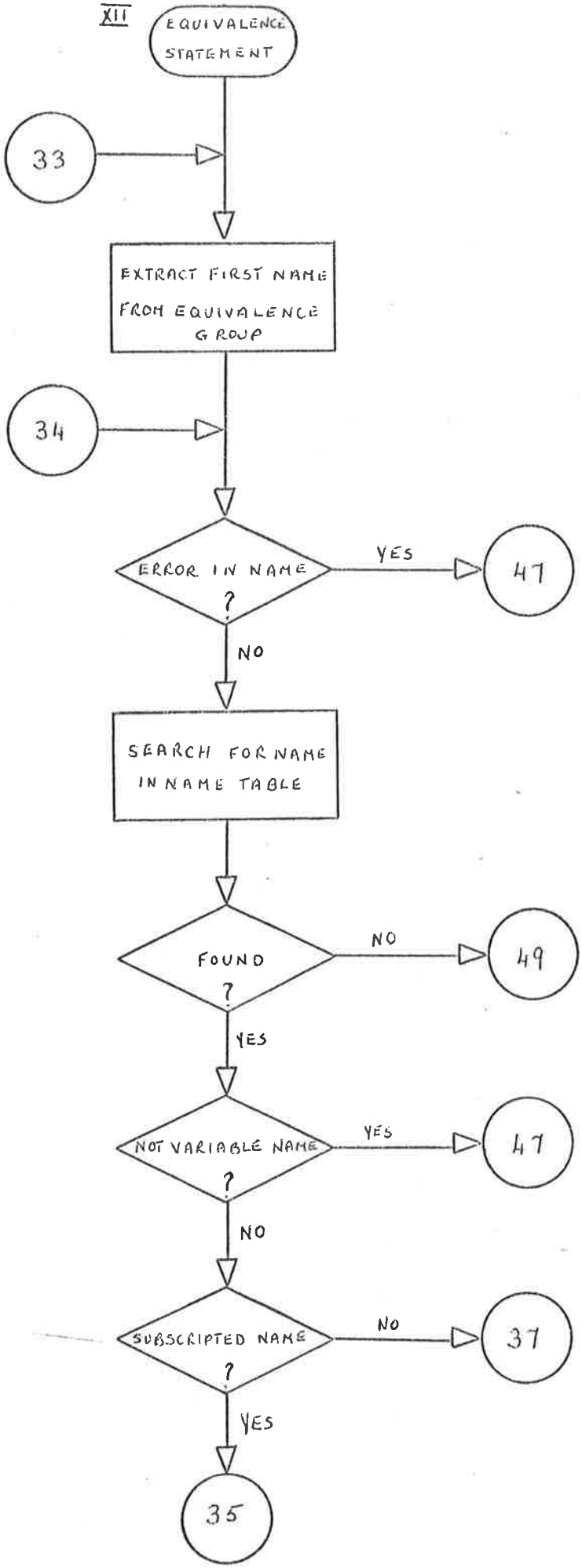

COMMON STATEMENT
21
BLANK COMMON ?
YES
FIRST NAME STARTS WITH LETTER ?
YES
24
NO
22
NO
FLAG ERROR 11
1
23
YES
NAMED COMMON ?
NO
NUMBERED COMMON ?
NO
21
YES
EXTRACT COMMON BLOCK NAME AND REMOVE LEADING ZEROS
ERROR IN COMMON BLOCK NAME ?
YES
FLAG ERROR 11
1
NO
25

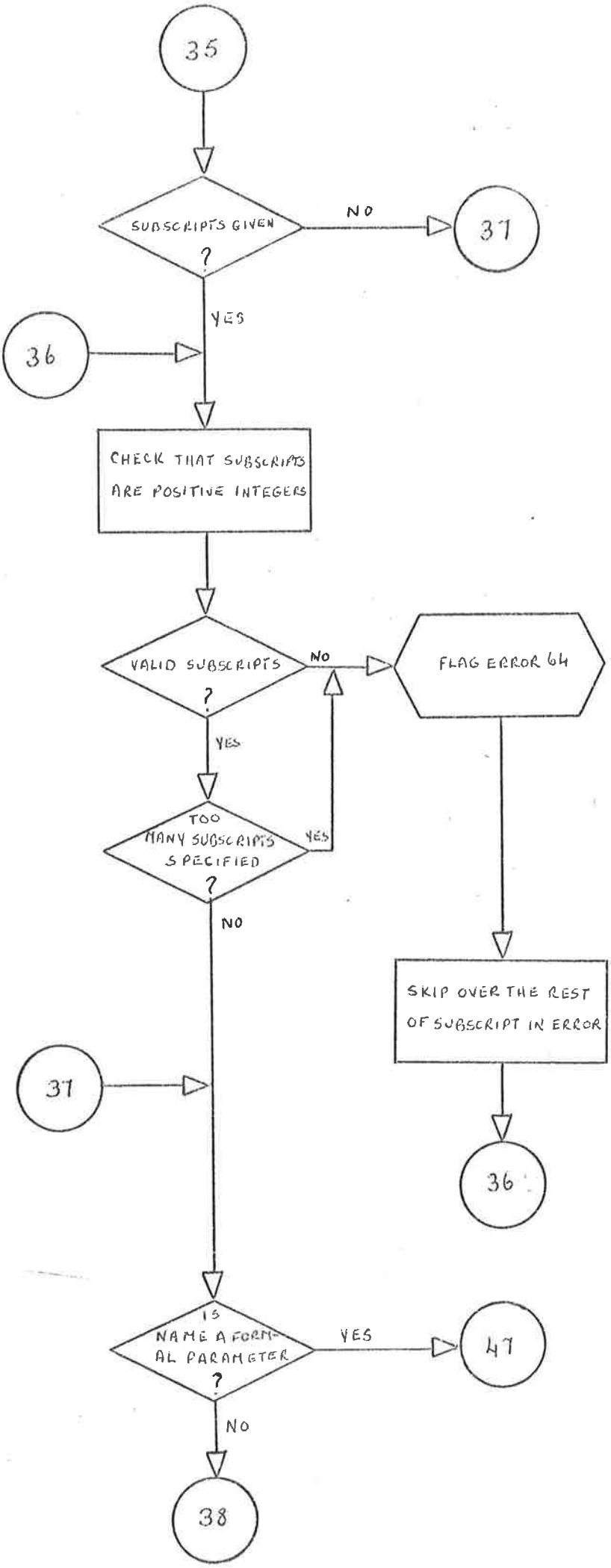

23
EXTRACT COMMON BLOCK NAME
24
ERROR IN COMMON BLOCK NAME ?
YES
FLAG ERROR 11
1
NO
25
SEARCH COMMON BLOCK NAME TABLE FOR THIS NAME
FOUND ?
YES
ALREADY DEFINED IN THIS SUBPROGRAM ?
NO
YES
SET DEFINED BIT IN ENTRY
24
26
NO
X
IS COMMON BLOCK NAME TABLE FULL ?
YES
FLAG ERROR 7
NO
26
27

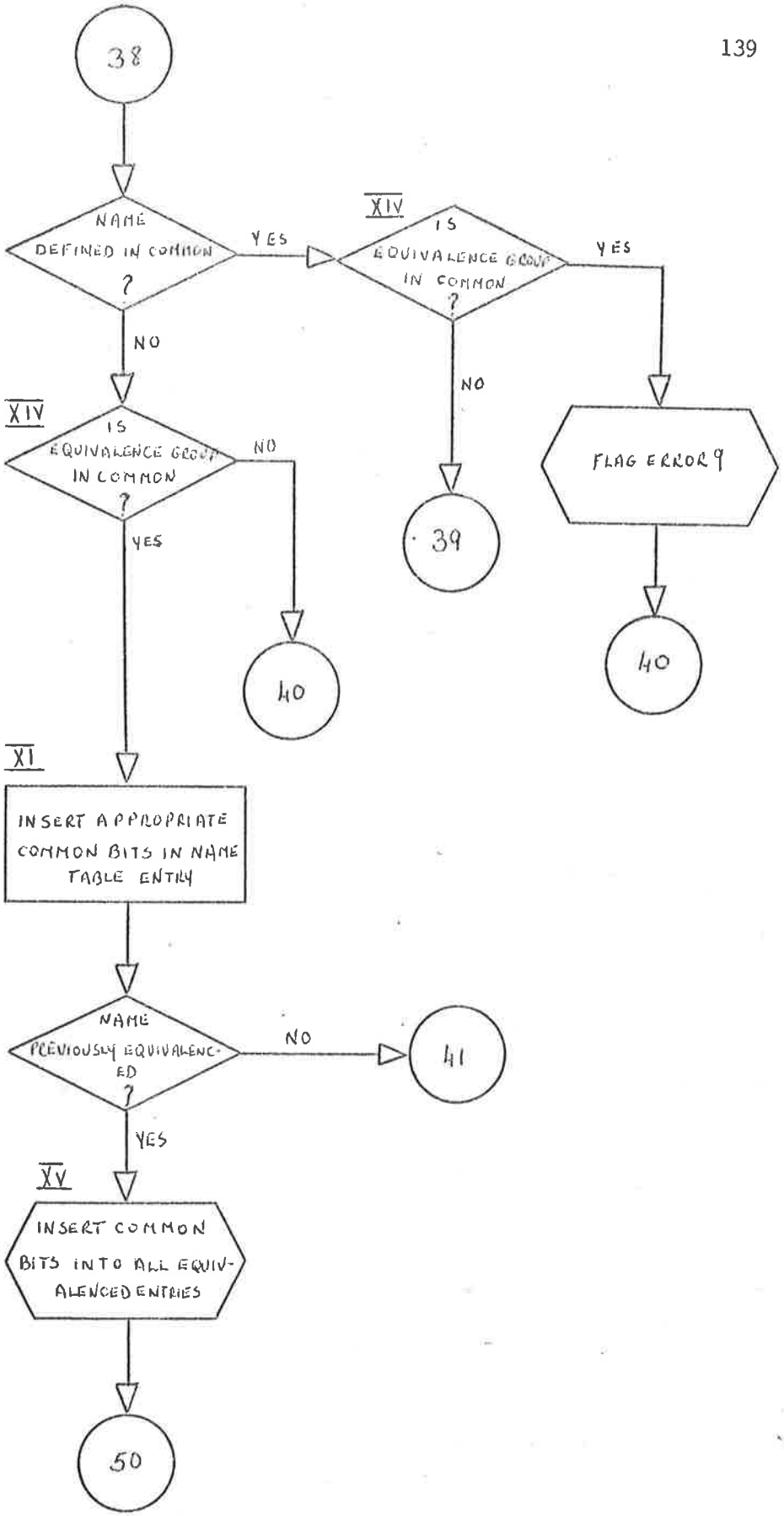

26
WRITE OUT ENTRY INTO COMMON BLOCK NAME TABLE
27
EXTRACT FIRST NAME
28
ERROR IN NAME ?
YES
24
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
30
YES
NAME DEFINED AS VARIABLE ?
NO
24
YES
29

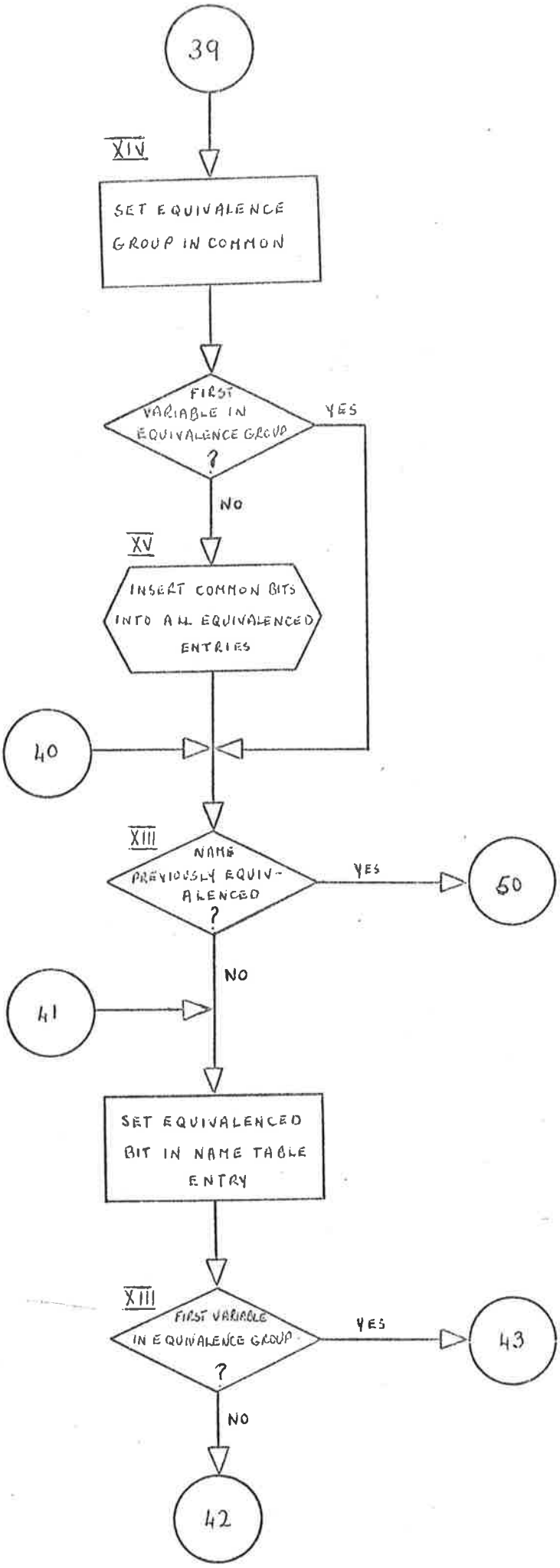

29
IS NAME A FORMAL PARAMETER ?
YES
24
NO
NAME ALREADY DEFINED IN COMMON ?
YES
FLAG ERROR 8
NO
NAME SUBSCRIPTED ?
NO
32
YES
IS NAME EQUIVALENCED ?
NO
YES
XV
INSERT COMMON BITS INTO ALL EQUIVALENCED ENTRIES
SKIP OVER SUBSCRIPT(s)
32
XVIII
GENERATE IMPLICIT TYPE
30
XI
INSERT APPROPRIATE COMMON BITS IN NAME TABLE ENTRY
31

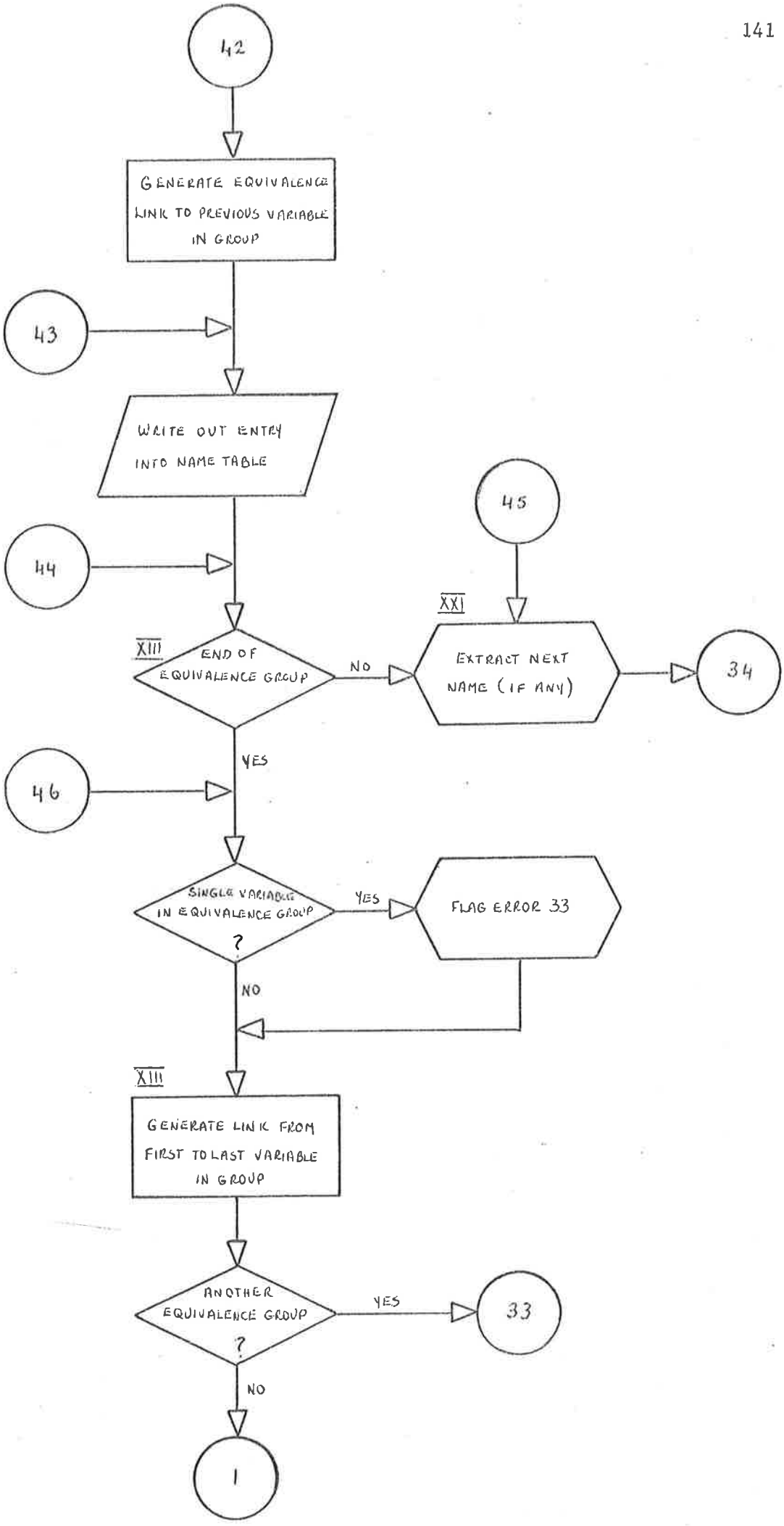

31
DIMENSIONS DEFINED ?
NO
YES
PROCESS DIMENSIONS
32
START OF A NEW COMMON BLOCK ?
YES
22
NO
XXI
EXTRACT NEXT NAME (IF ANY)
28

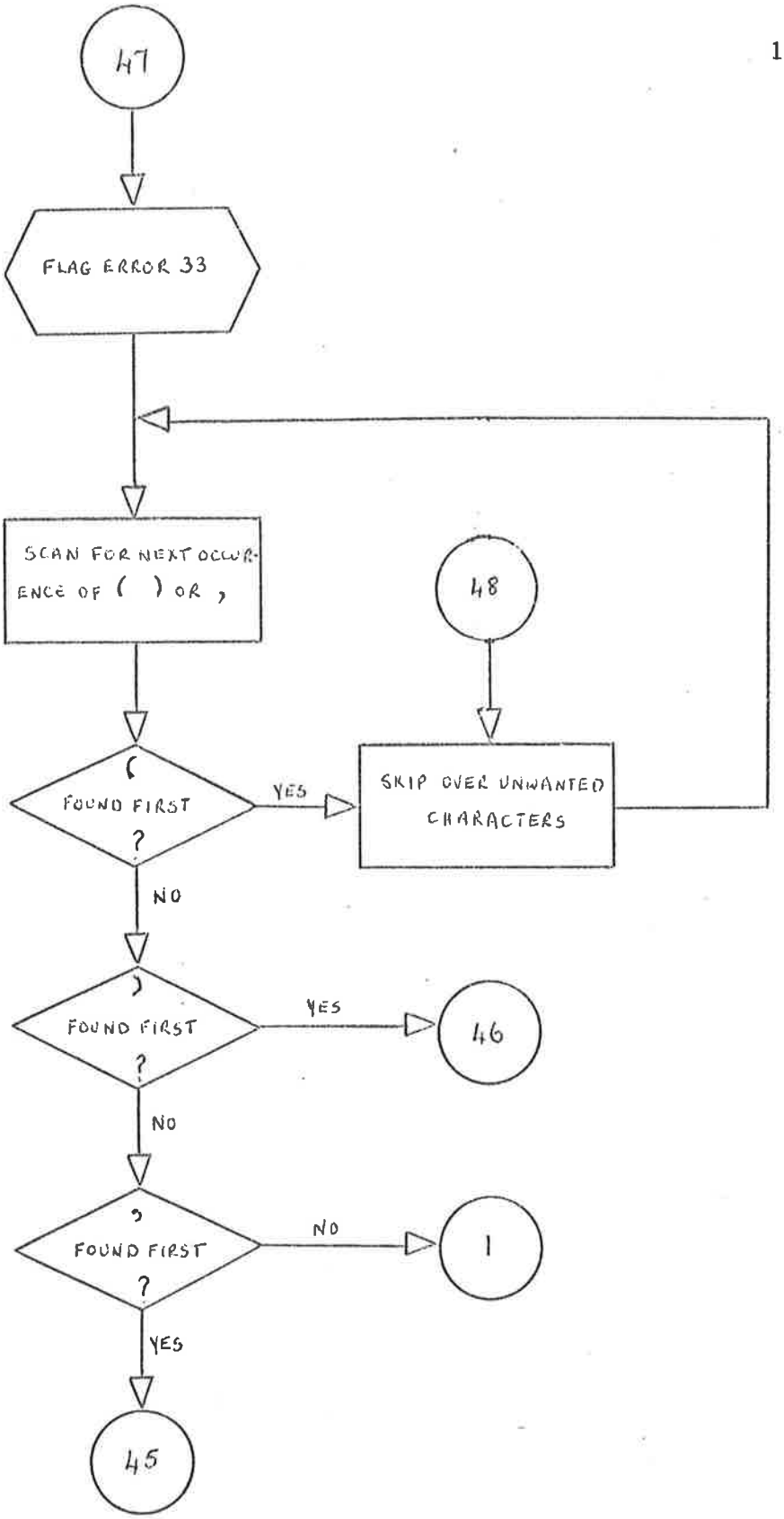

XII
EQUIVALENCE STATEMENT
33
EXTRACT FIRST NAME FROM EQUIVALENCE GROUP
34
ERROR IN NAME ?
YES
47
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
49
YES
NOT VARIABLE NAME ?
YES
47
NO
SUBSCRIPTED NAME ?
NO
37
YES
35

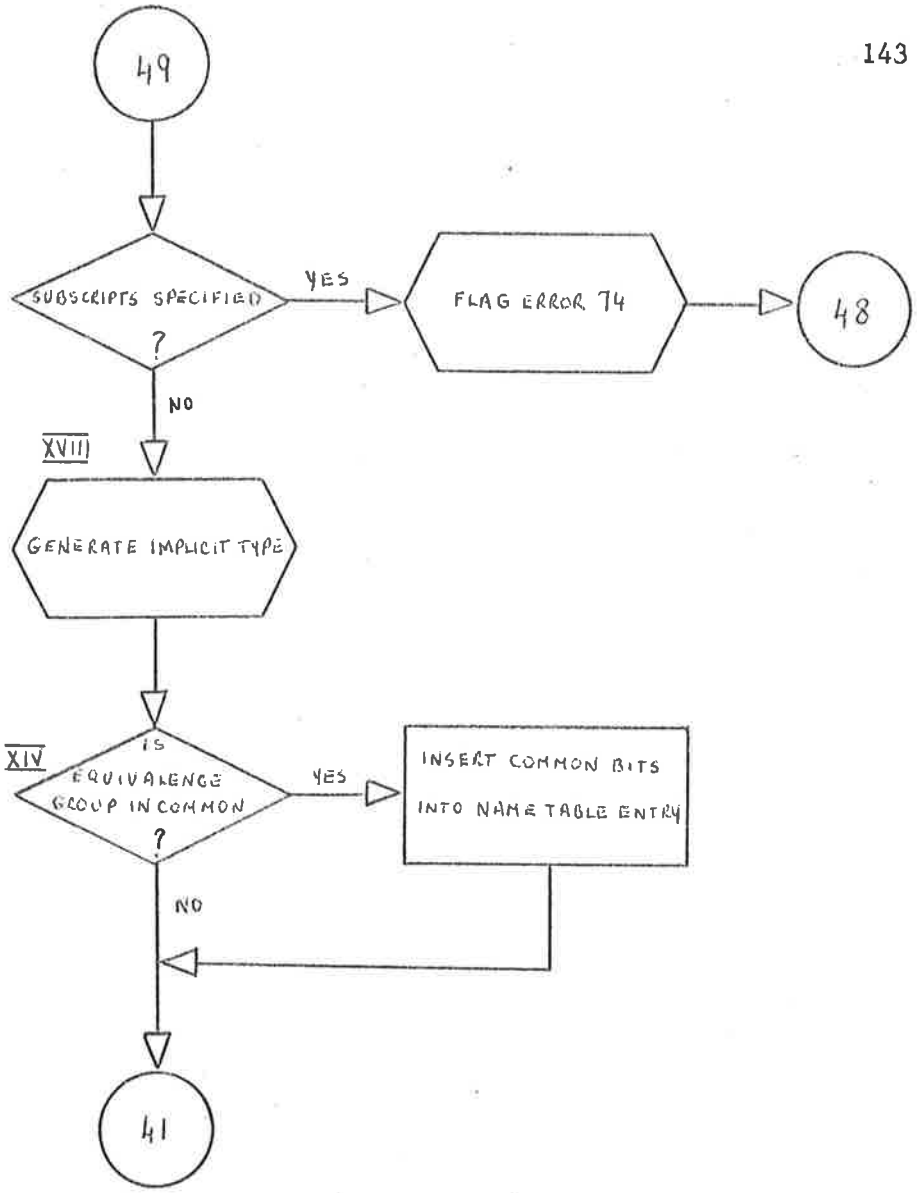

35
SUBSCRIPTS GIVEN ?
NO
37
YES
36
CHECK THAT SUBSCRIPTS ARE POSITIVE INTEGERS
VALID SUBSCRIPTS ?
NO
FLAG ERROR 64
YES
TOO MANY SUBSCRIPTS SPECIFIED ?
YES
NO
SKIP OVER THE REST OF SUBSCRIPT IN ERROR
37
36
IS NAME A FORMAL PARAMETER ?
YES
47
NO
38

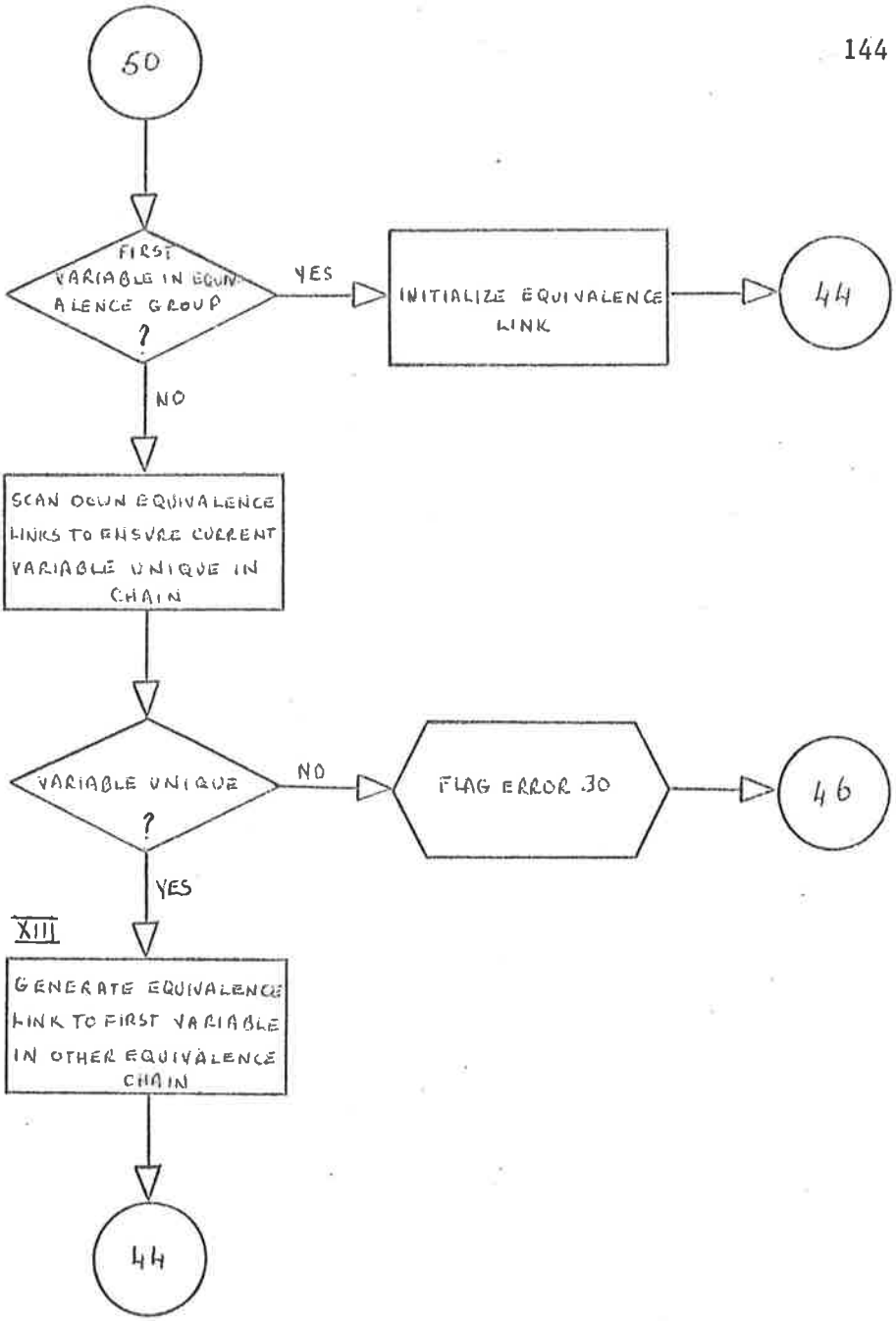

38
NAME DEFINED IN COMMON ?
YES
NO
XIV
IS EQUIVALENCE GROUP IN COMMON ?
YES
NO
39
FLAG ERROR 9
40
XIV
IS EQUIVALENCE GROUP IN COMMON ?
NO
YES
40
XI
INSERT APPROPRIATE COMMON BITS IN NAME TABLE ENTRY
NAME PREVIOUSLY EQUIVALENCED ?
NO
41
YES
XV
INSERT COMMON BITS INTO ALL EQUIVALENCED ENTRIES
50

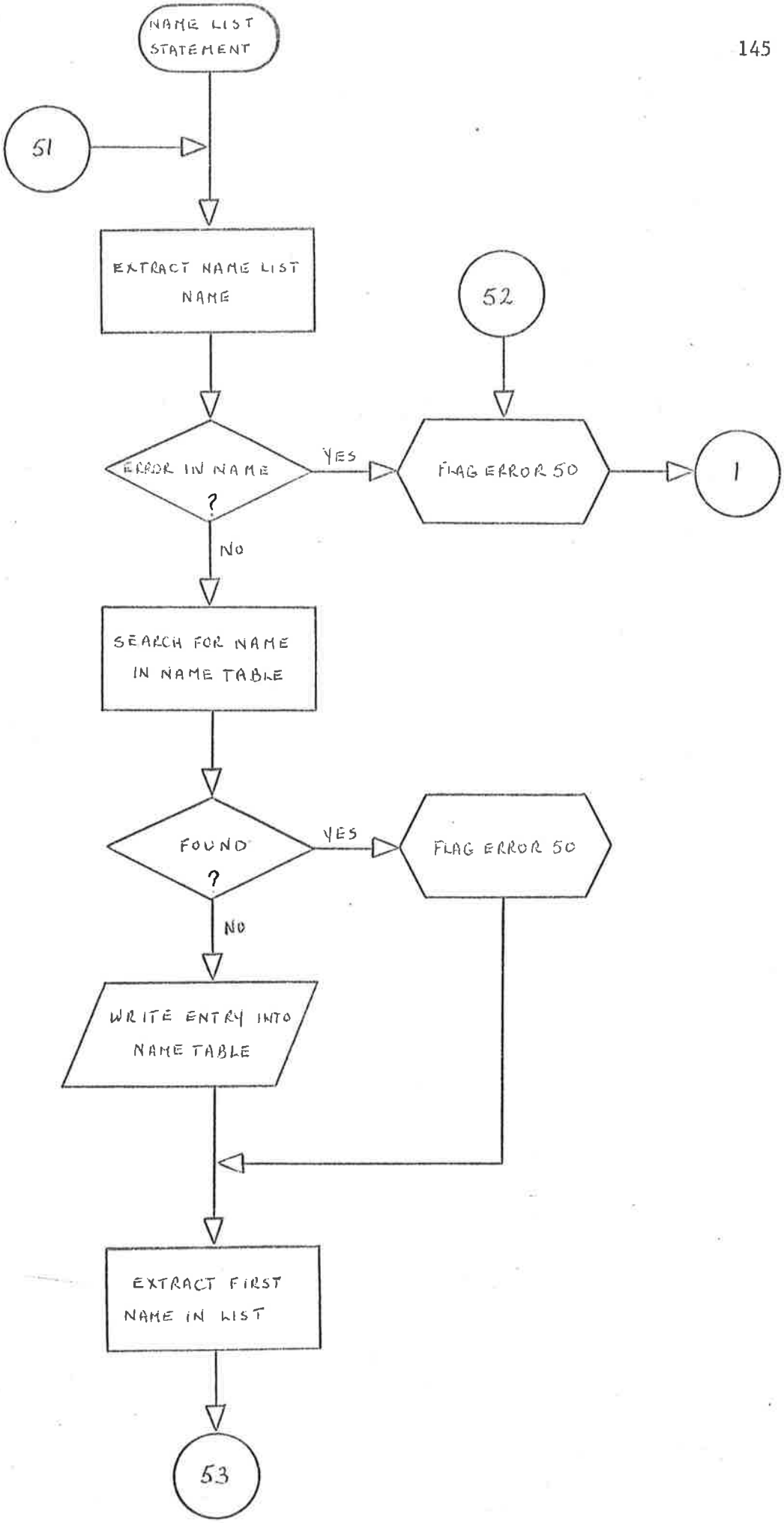

39
XIV
SET EQUIVALENCE GROUP IN COMMON
FIRST VARIABLE IN EQUIVALENCE GROUP ?
YES
NO
XV
INSERT COMMON BITS INTO ALL EQUIVALENCED ENTRIES
40
XIII
NAME PREVIOUSLY EQUIVALENCED ?
YES
50
NO
41
SET EQUIVALENCED BIT IN NAME TABLE ENTRY
XIII
FIRST VARIABLE IN EQUIVALENCE GROUP ?
YES
43
NO
42

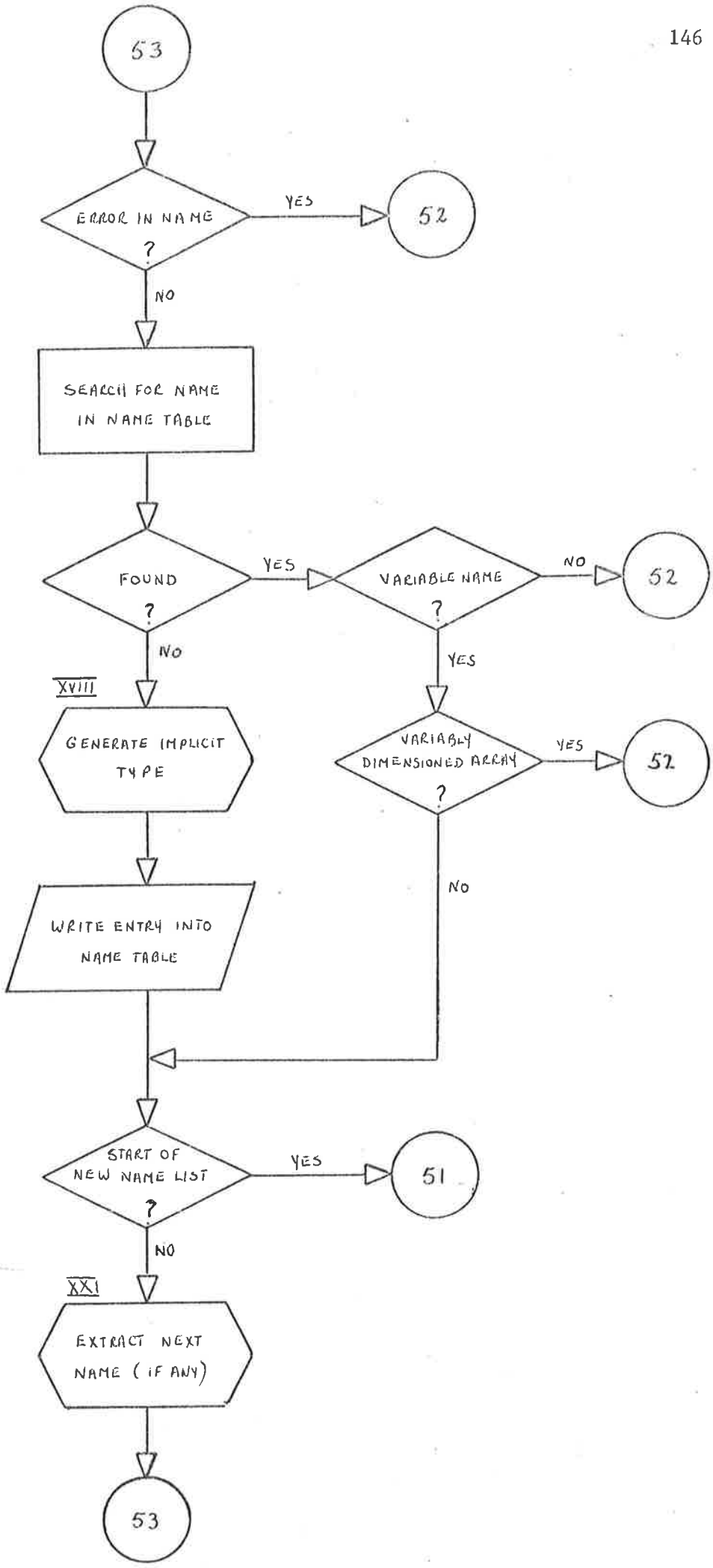

42
GENERATE EQUIVALENCE LINK TO PREVIOUS VARIABLE IN GROUP
43
WRITE OUT ENTRY INTO NAME TABLE
44
45
XXI
XIII
END OF EQUIVALENCE GROUP
NO
EXTRACT NEXT NAME (IF ANY)
34
YES
46
SINGLE VARIABLE IN EQUIVALENCE GROUP ?
YES
FLAG ERROR 33
NO
XIII
GENERATE LINK FROM FIRST TO LAST VARIABLE IN GROUP
ANOTHER EQUIVALENCE GROUP ?
YES
33
NO
1

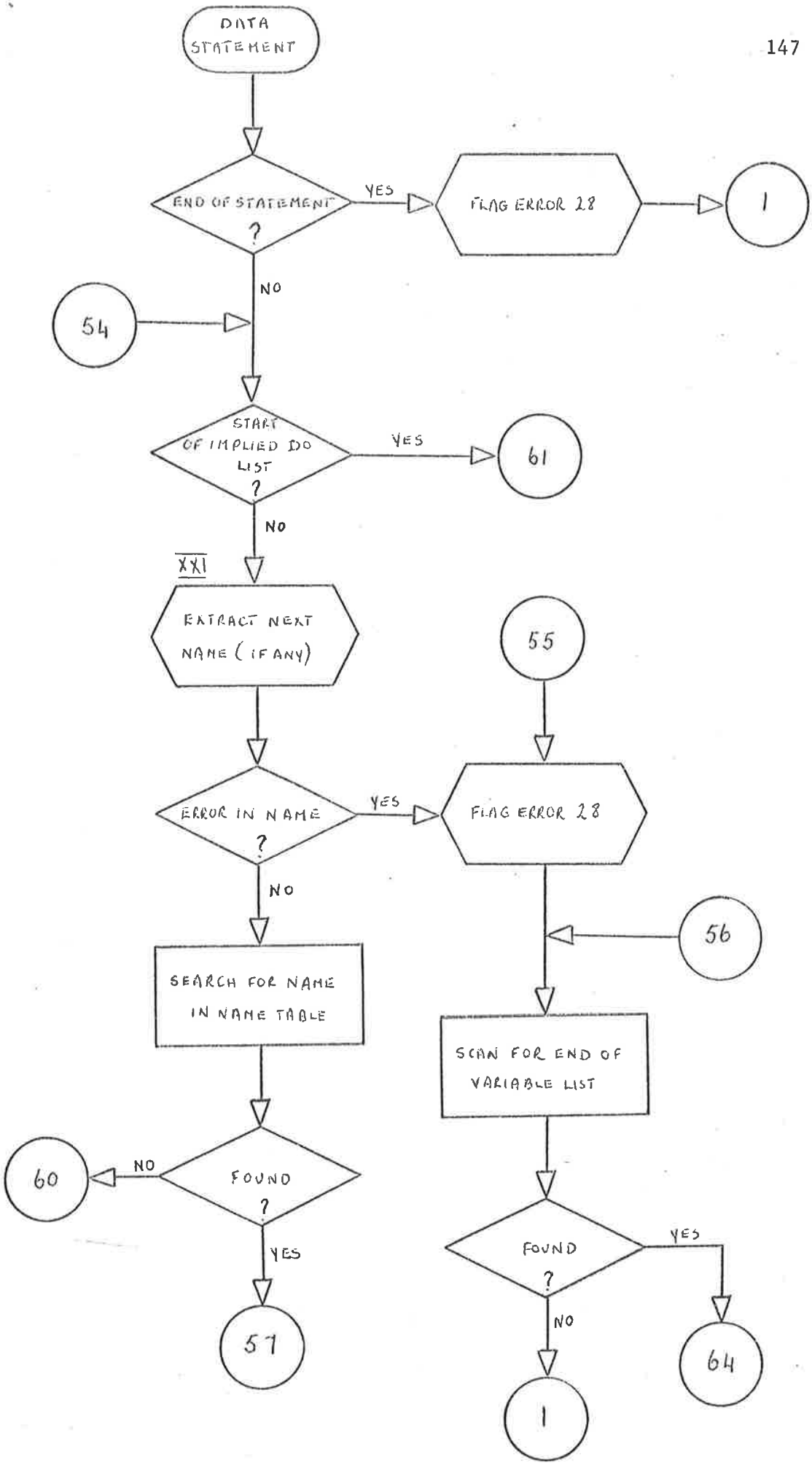

47
FLAG ERROR 33
SCAN FOR NEXT OCCURENCE OF ( ) OR ,
48
(
FOUND FIRST
?
YES
SKIP OVER UNWANTED CHARACTERS
NO
)
FOUND FIRST
?
YES
46
NO
,
FOUND FIRST
?
NO
1
YES
45

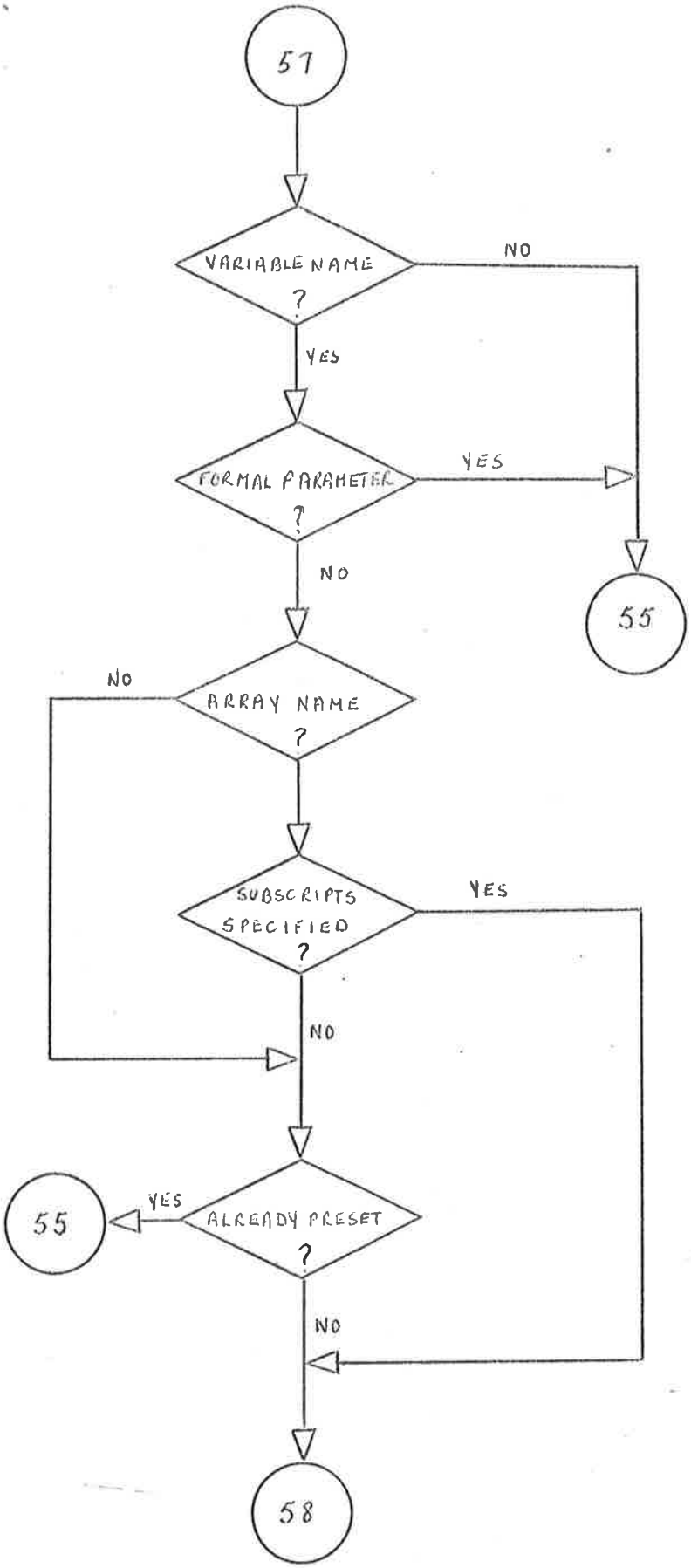

49
SUBSCRIPTS SPECIFIED ?
YES
FLAG ERROR 74
48
NO
XVIII
GENERATE IMPLICIT TYPE
XIV
IS EQUIVALENCE GROUP IN COMMON ?
YES
INSERT COMMON BITS INTO NAME TABLE ENTRY
NO
41

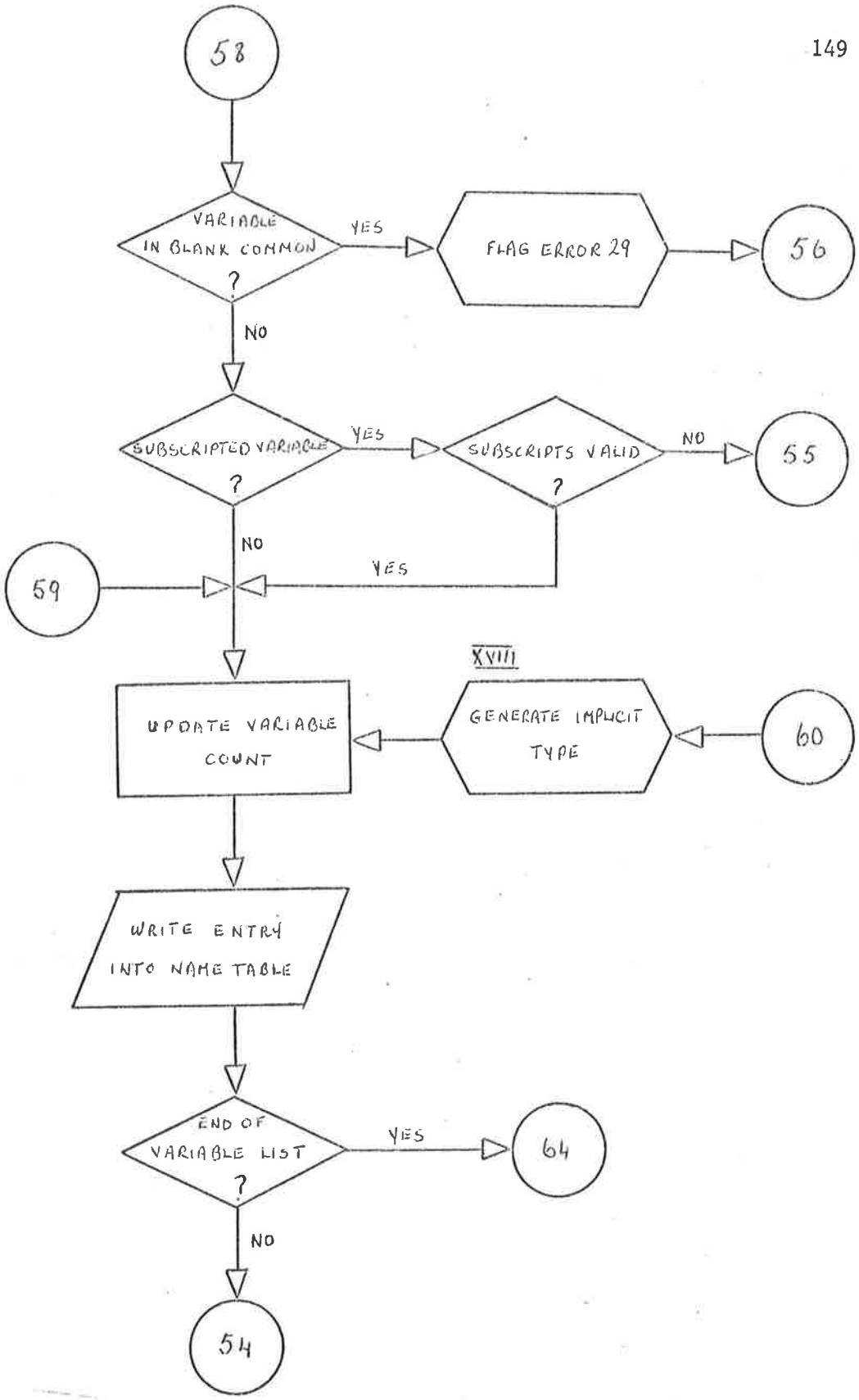

50
FIRST VARIABLE IN EQUIVALENCE GROUP ?
YES
INITIALIZE EQUIVALENCE LINK
44
NO
SCAN OCUN EQUIVALENCE LINKS TO ENSURE CURRENT VARIABLE UNIQUE IN CHAIN
VARIABLE UNIQUE ?
NO
FLAG ERROR 30
46
YES
XIII
GENERATE EQUIVALENCE LINK TO FIRST VARIABLE IN OTHER EQUIVALENCE CHAIN
44

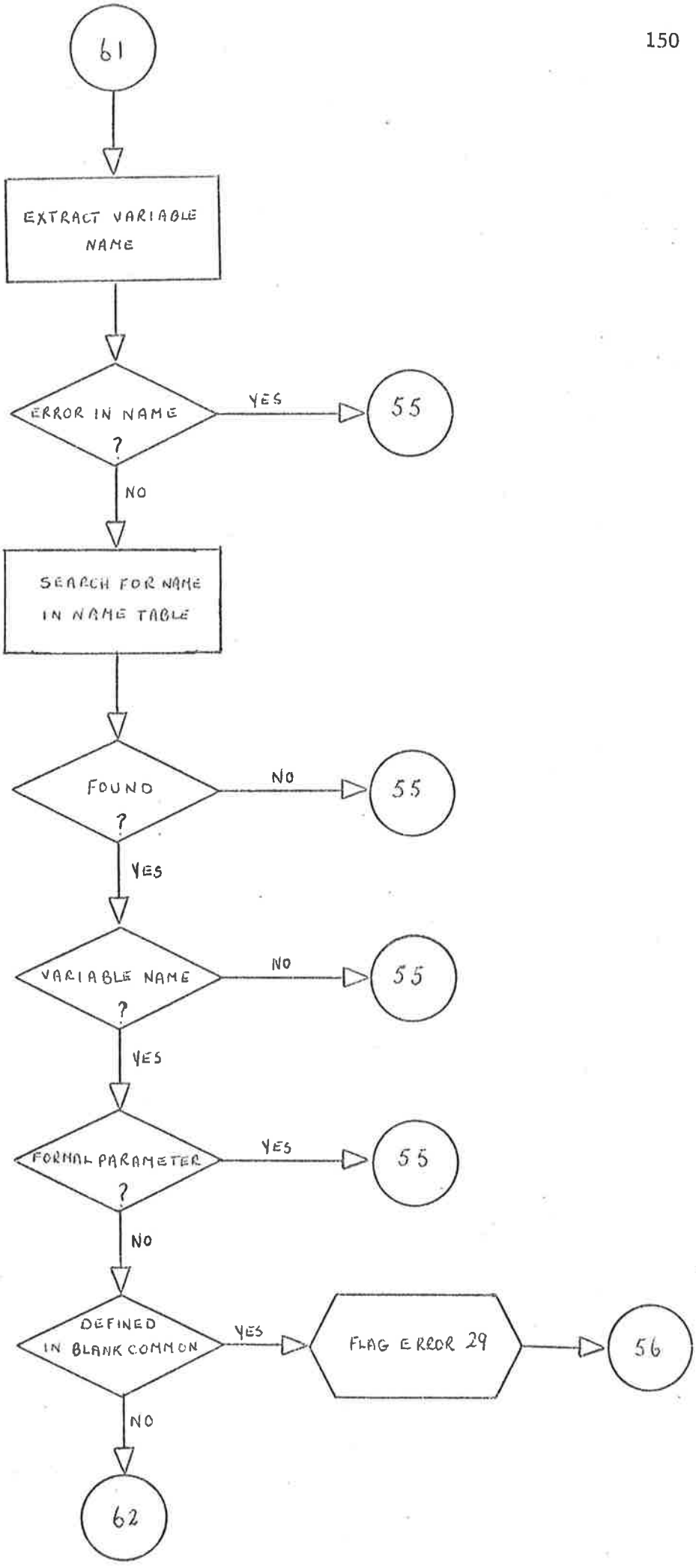

NAME LIST STATEMENT
51
EXTRACT NAME LIST NAME
52
ERROR IN NAME ?
YES
FLAG ERROR 50
1
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
YES
FLAG ERROR 50
NO
WRITE ENTRY INTO NAME TABLE
EXTRACT FIRST NAME IN LIST
53

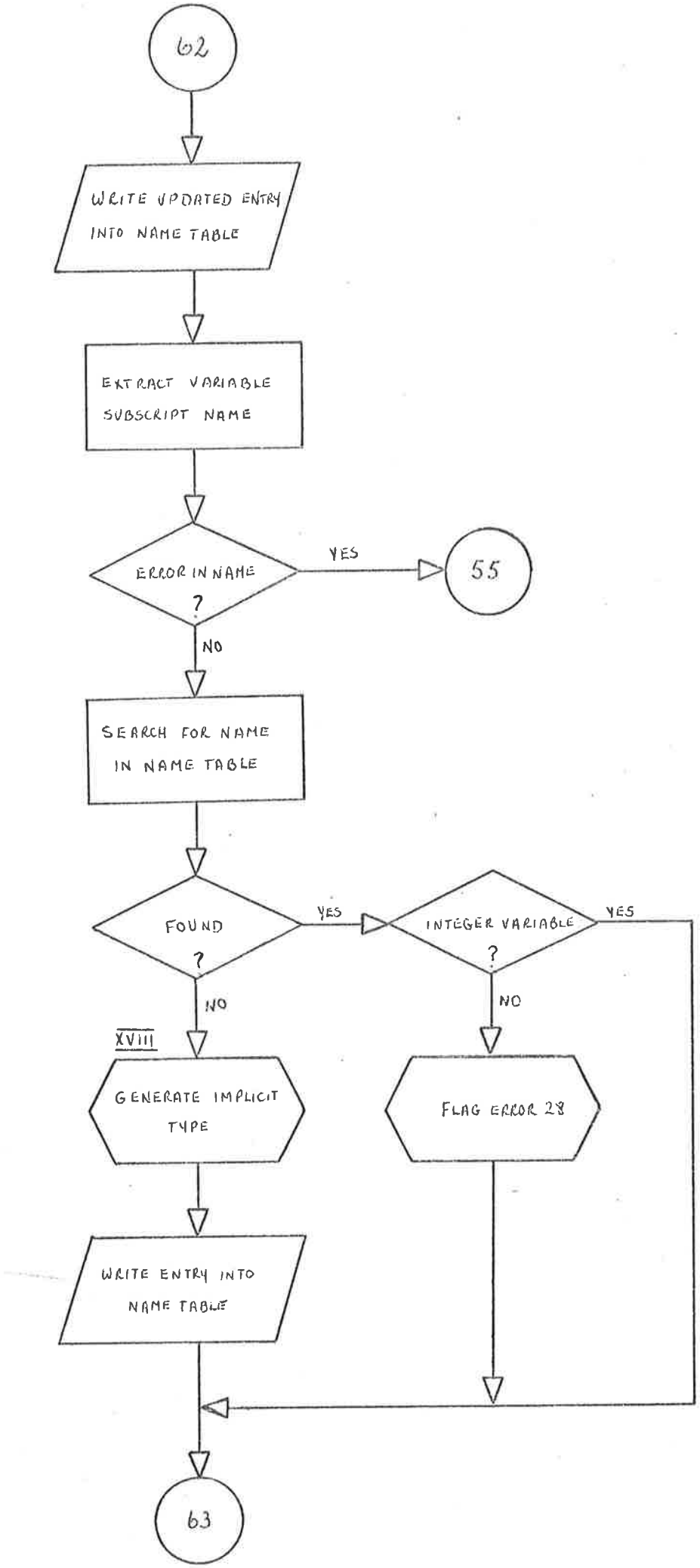

53
ERROR IN NAME ?
YES
52
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
YES
VARIABLE NAME ?
NO
52
NO
YES
XVIII
GENERATE IMPLICIT TYPE
VARIABLY DIMENSIONED ARRAY ?
YES
52
NO
WRITE ENTRY INTO NAME TABLE
START OF NEW NAME LIST ?
YES
51
NO
XXI
EXTRACT NEXT NAME (IF ANY)
53

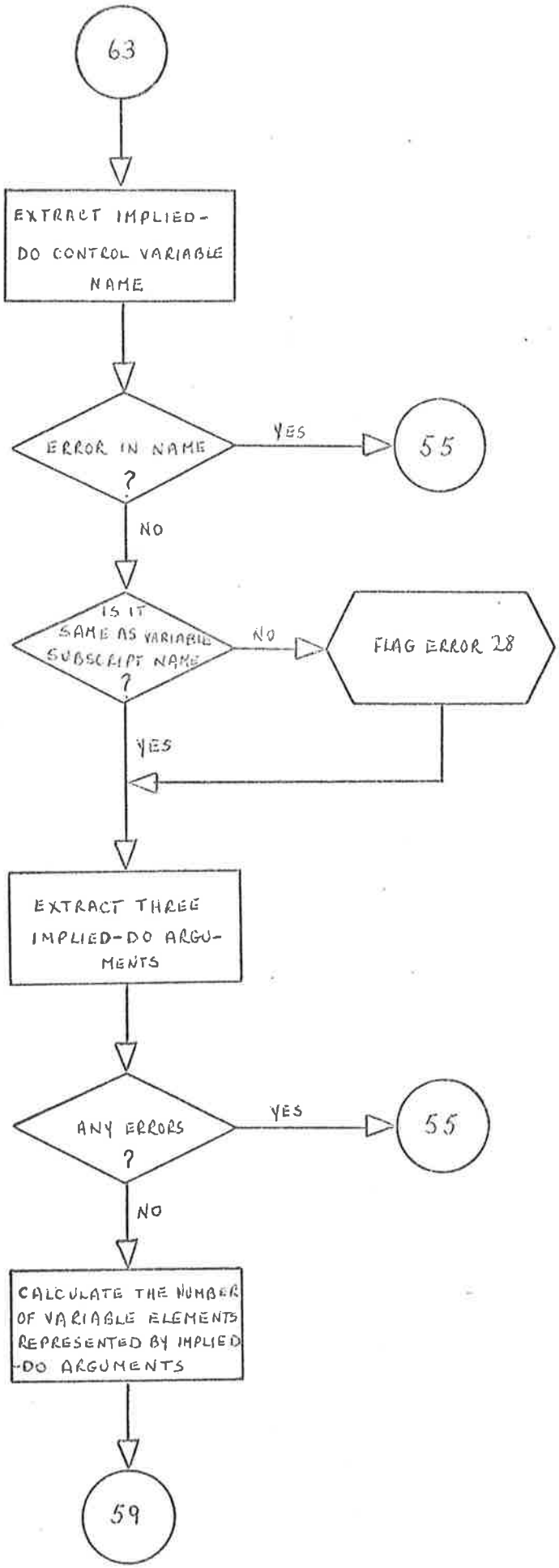

DATA STATEMENT
END OF STATEMENT ?
YES
FLAG ERROR 28
1
NO
54
START OF IMPLIED DO LIST ?
YES
61
NO
XXI
EXTRACT NEXT NAME (IF ANY)
55
ERROR IN NAME ?
YES
FLAG ERROR 28
NO
56
SEARCH FOR NAME IN NAME TABLE
SCAN FOR END OF VARIABLE LIST
60
NO
FOUND ?
YES
57
FOUND ?
YES
NO
64
1

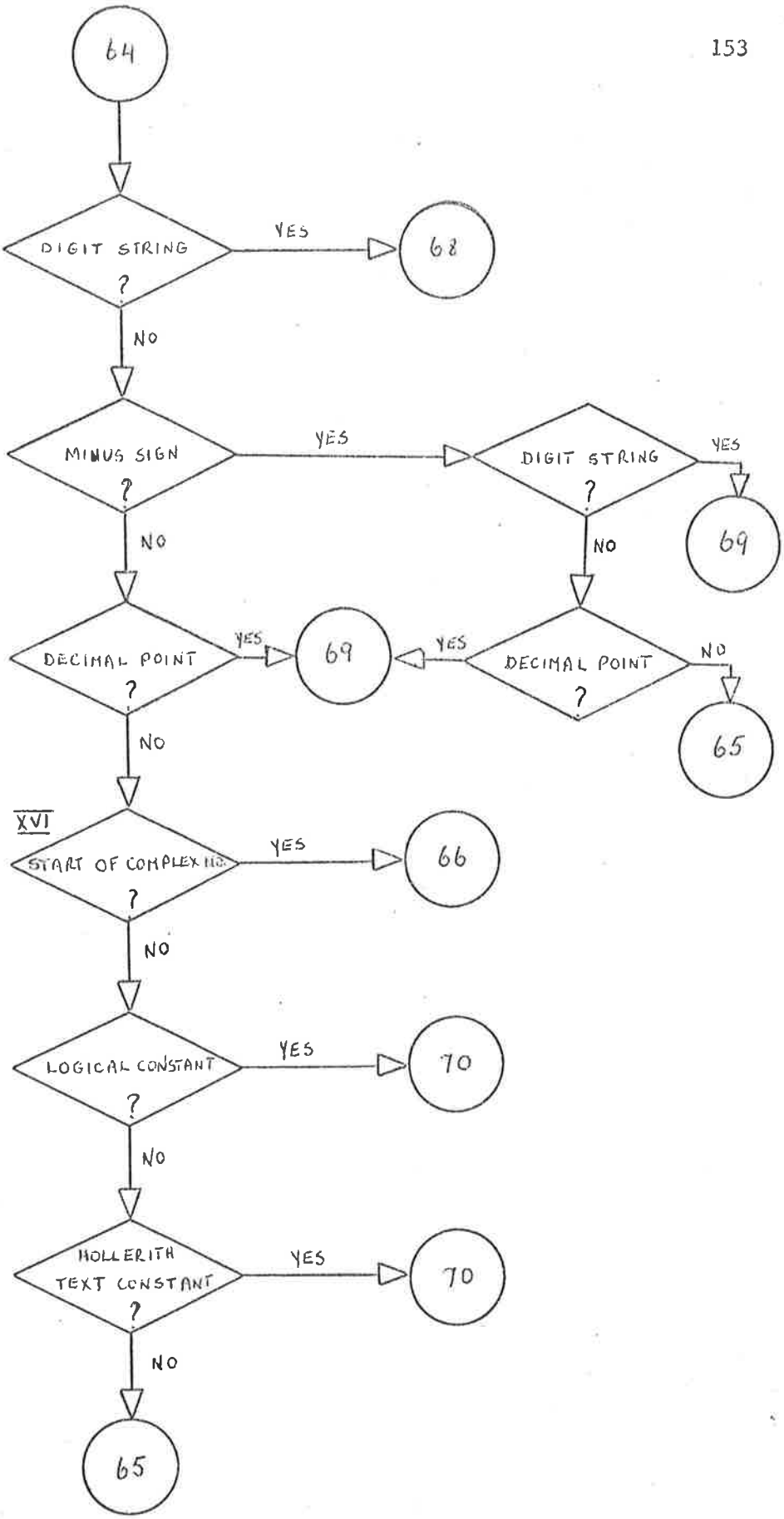

57
VARIABLE NAME ?
NO
YES
FORMAL PARAMETER ?
YES
NO
55
NO
ARRAY NAME ?
SUBSCRIPTS SPECIFIED ?
YES
NO
55
YES
ALREADY PRESET ?
NO
58

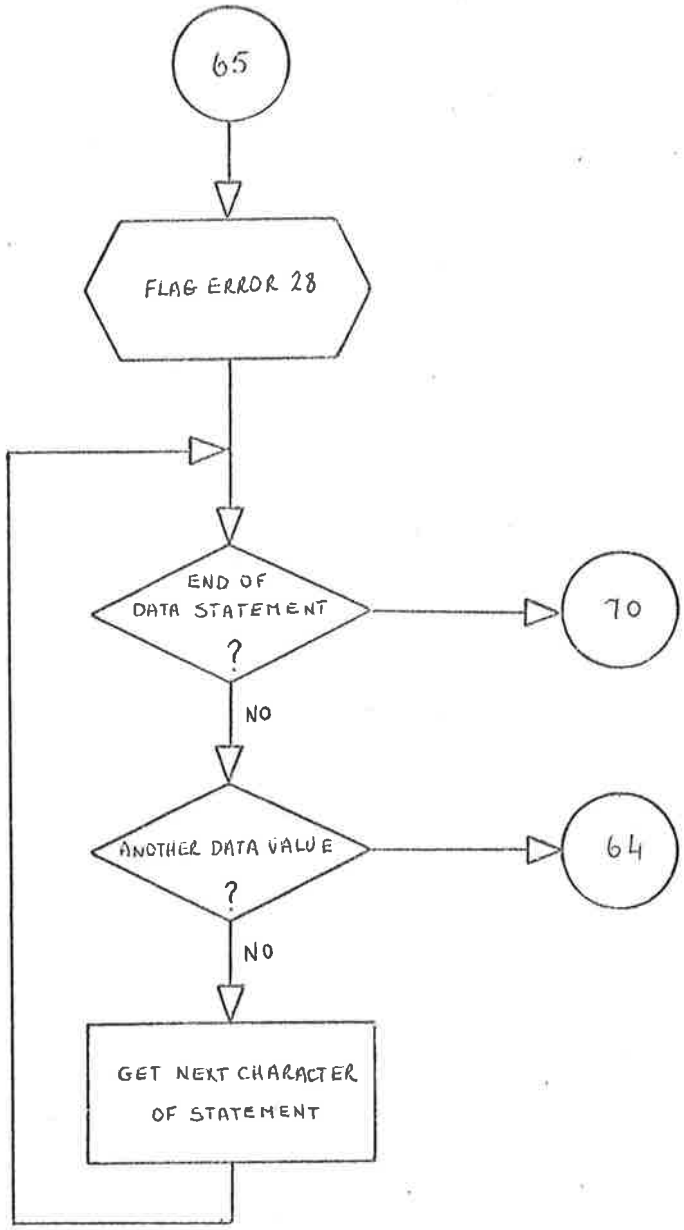

58
VARIABLE IN BLANK COMMON ?
YES
FLAG ERROR 29
56
NO
SUBSCRIPTED VARIABLE ?
YES
SUBSCRIPTS VALID ?
NO
55
NO
YES
59
XVIII
UPDATE VARIABLE COUNT
GENERATE IMPLICIT TYPE
60
WRITE ENTRY INTO NAME TABLE
END OF VARIABLE LIST ?
YES
64
NO
54

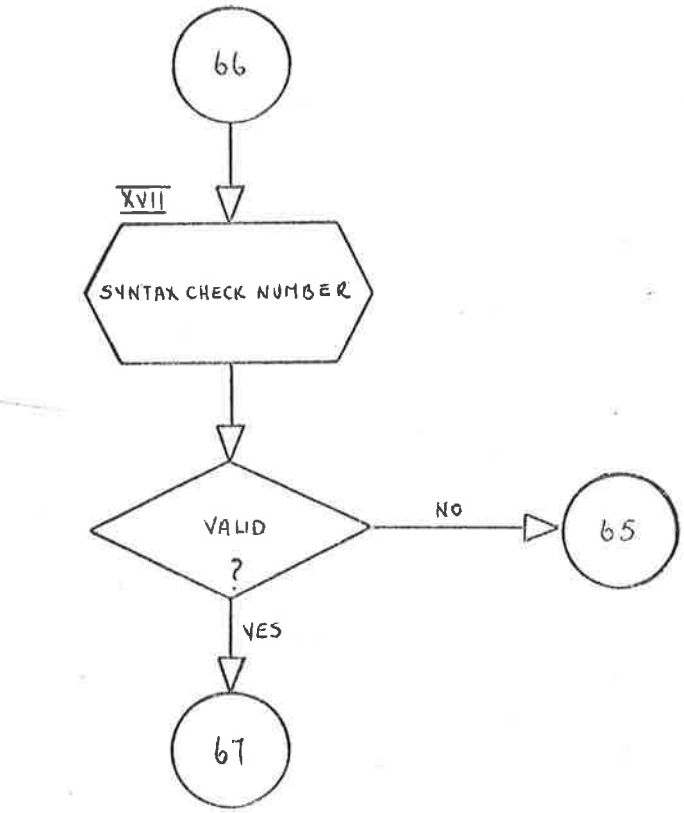

61
EXTRACT VARIABLE NAME
ERROR IN NAME ?
YES
55
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
55
YES
VARIABLE NAME ?
NO
55
YES
FORMAL PARAMETER ?
YES
55
NO
DEFINED IN BLANK COMMON
YES
FLAG ERROR 29
56
NO
62

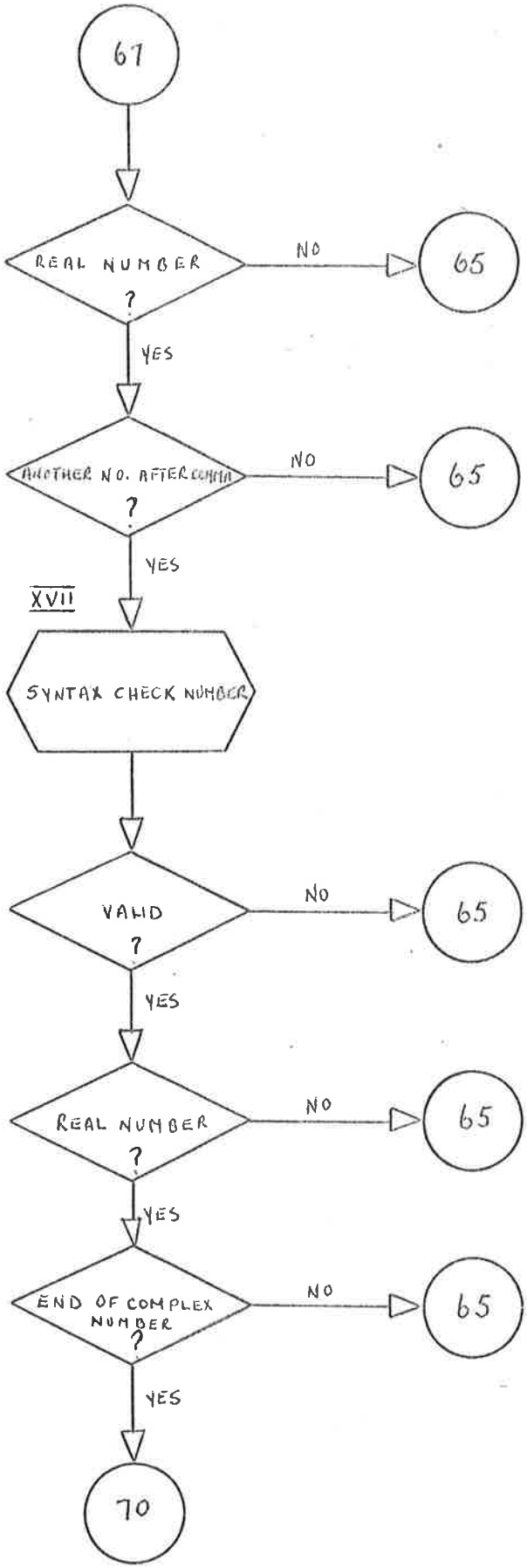

62
WRITE UPDATED ENTRY INTO NAME TABLE
EXTRACT VARIABLE SUBSCRIPT NAME
ERROR IN NAME ?
YES
55
NO
SEARCH FOR NAME IN NAME TABLE
FOUND ?
YES
INTEGER VARIABLE ?
YES
NO
NO
XVIII
GENERATE IMPLICIT TYPE
FLAG ERROR 28
WRITE ENTRY INTO NAME TABLE
63

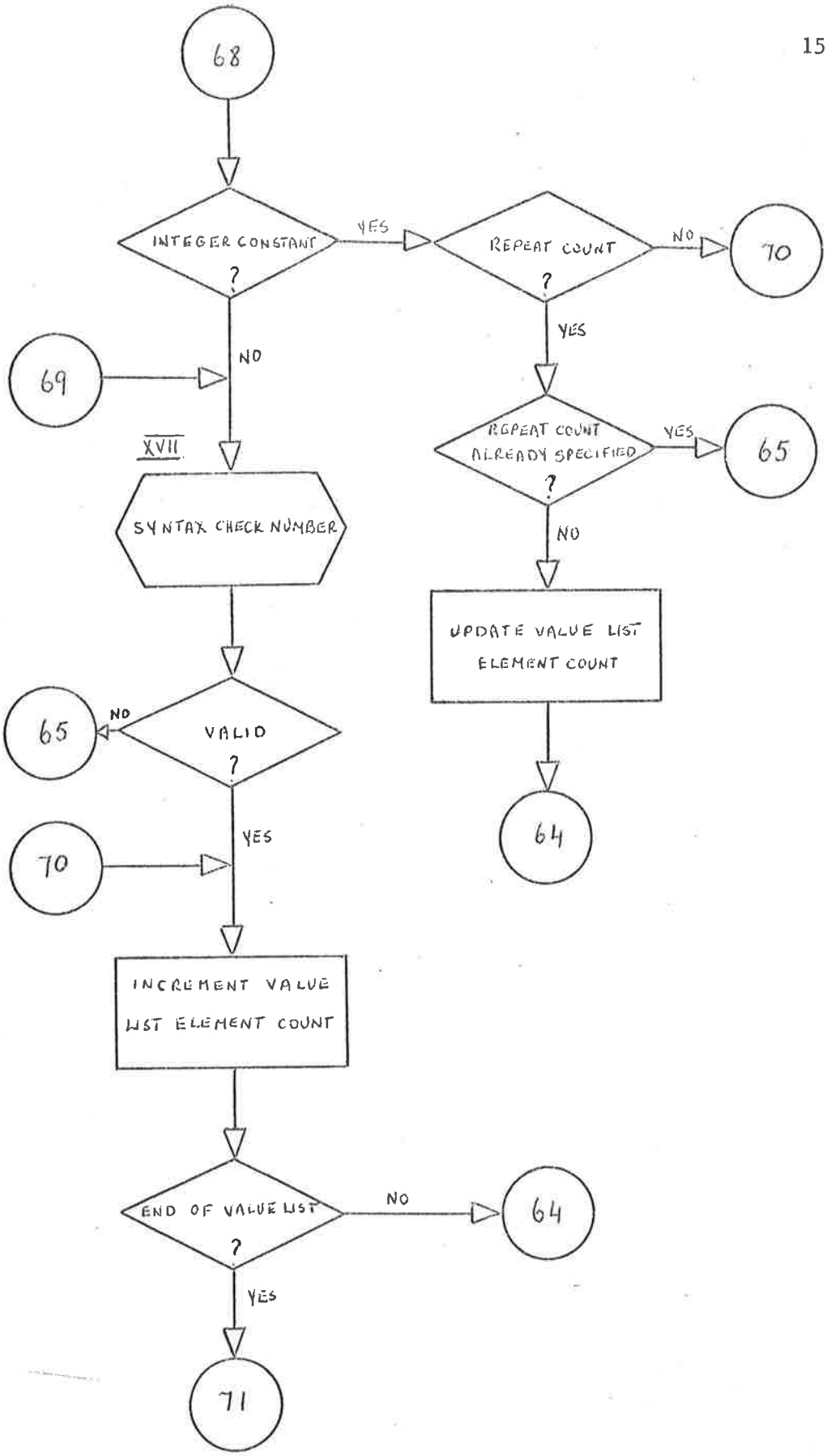

63
EXTRACT IMPLIED-DO CONTROL VARIABLE NAME
ERROR IN NAME ?
YES
55
NO
IS IT SAME AS VARIABLE SUBSCRIPT NAME ?
NO
FLAG ERROR 28
YES
EXTRACT THREE IMPLIED-DO ARGUMENTS
ANY ERRORS ?
YES
55
NO
CALCULATE THE NUMBER OF VARIABLE ELEMENTS REPRESENTED BY IMPLIED-DO ARGUMENTS
59

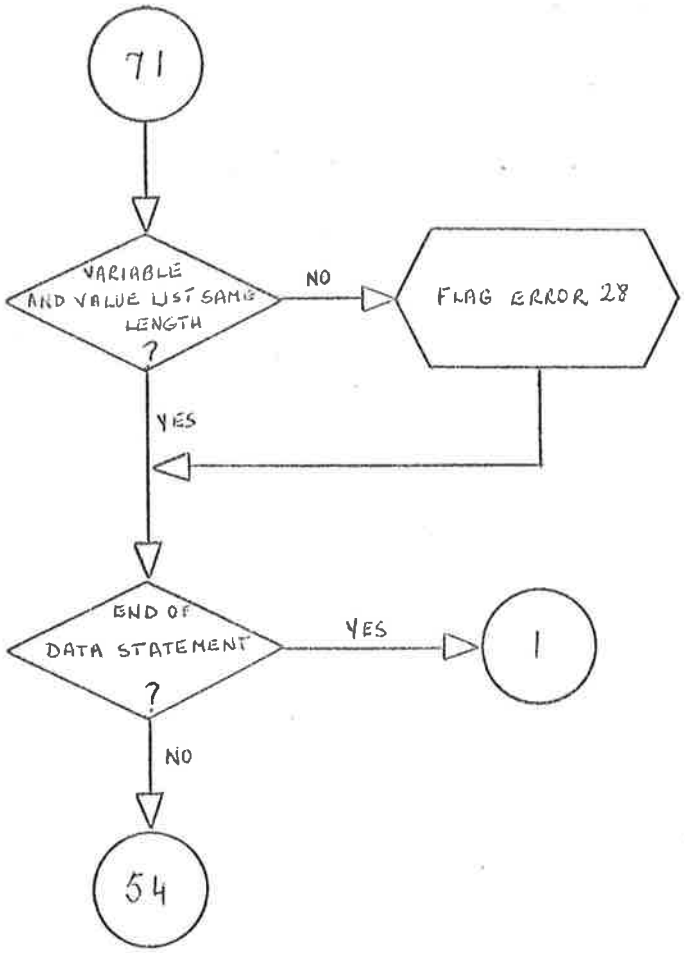

64
DIGIT STRING ?
YES
68
NO
MINUS SIGN ?
YES
DIGIT STRING ?
YES
69
NO
NO
DECIMAL POINT ?
YES
69
YES
DECIMAL POINT ?
NO
65
NO
XVI
START OF COMPLEX NO. ?
YES
66
NO
LOGICAL CONSTANT ?
YES
70
NO
HOLLERITH TEXT CONSTANT ?
YES
70
NO
65

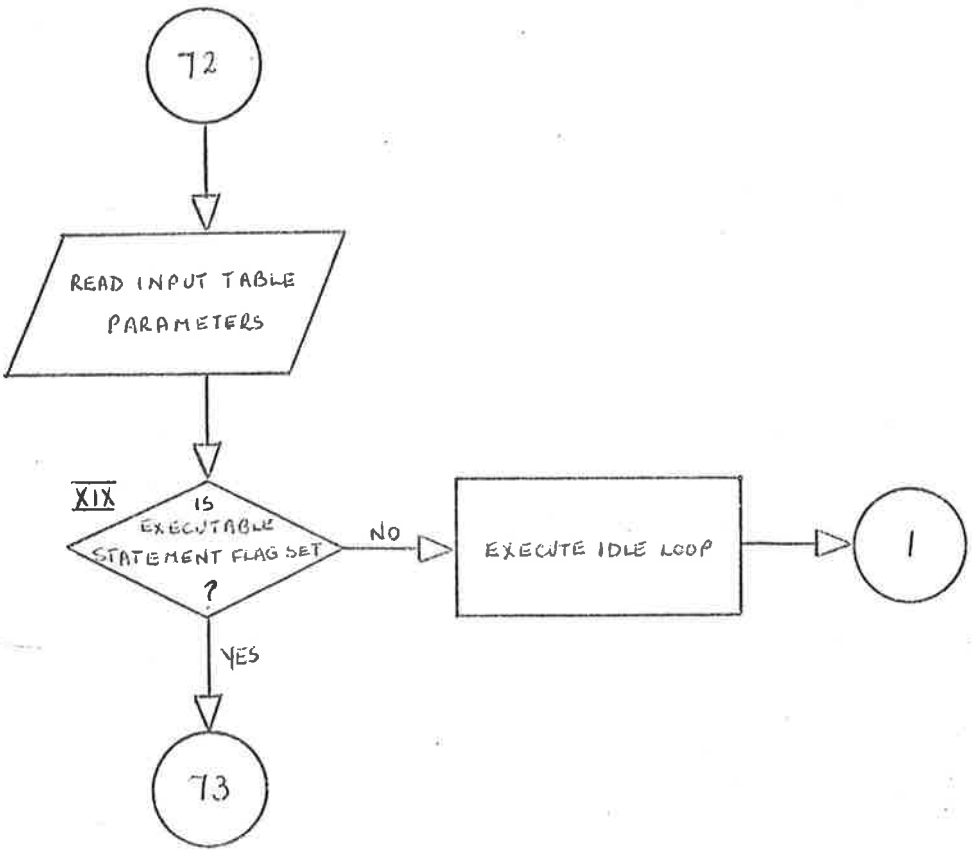

65
FLAG ERROR 28
END OF DATA STATEMENT ?
70
NO
ANOTHER DATA VALUE ?
64
NO
GET NEXT CHARACTER OF STATEMENT

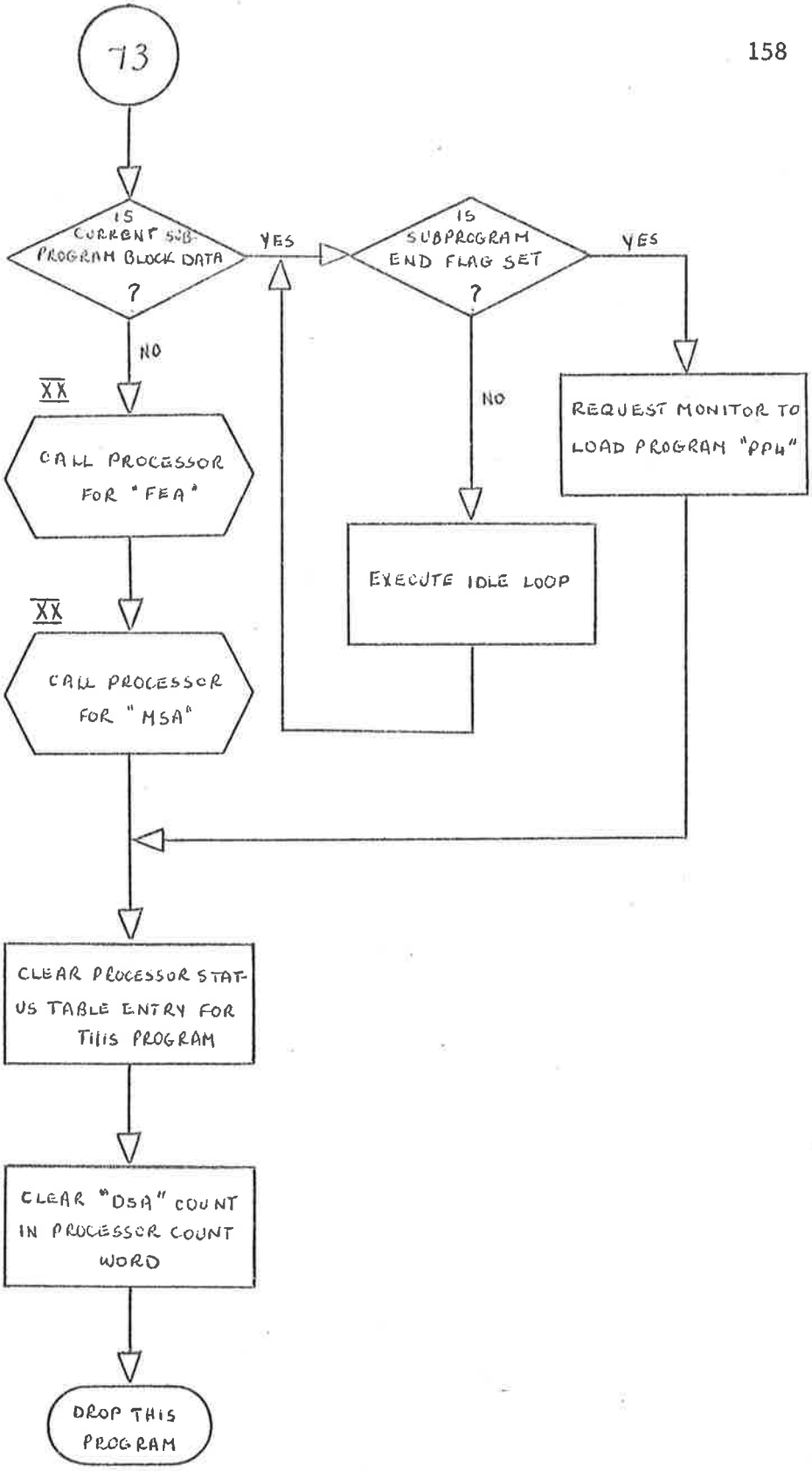

66
XVII
SYNTAX CHECK NUMBER
VALID ?
NO
65
YES
67

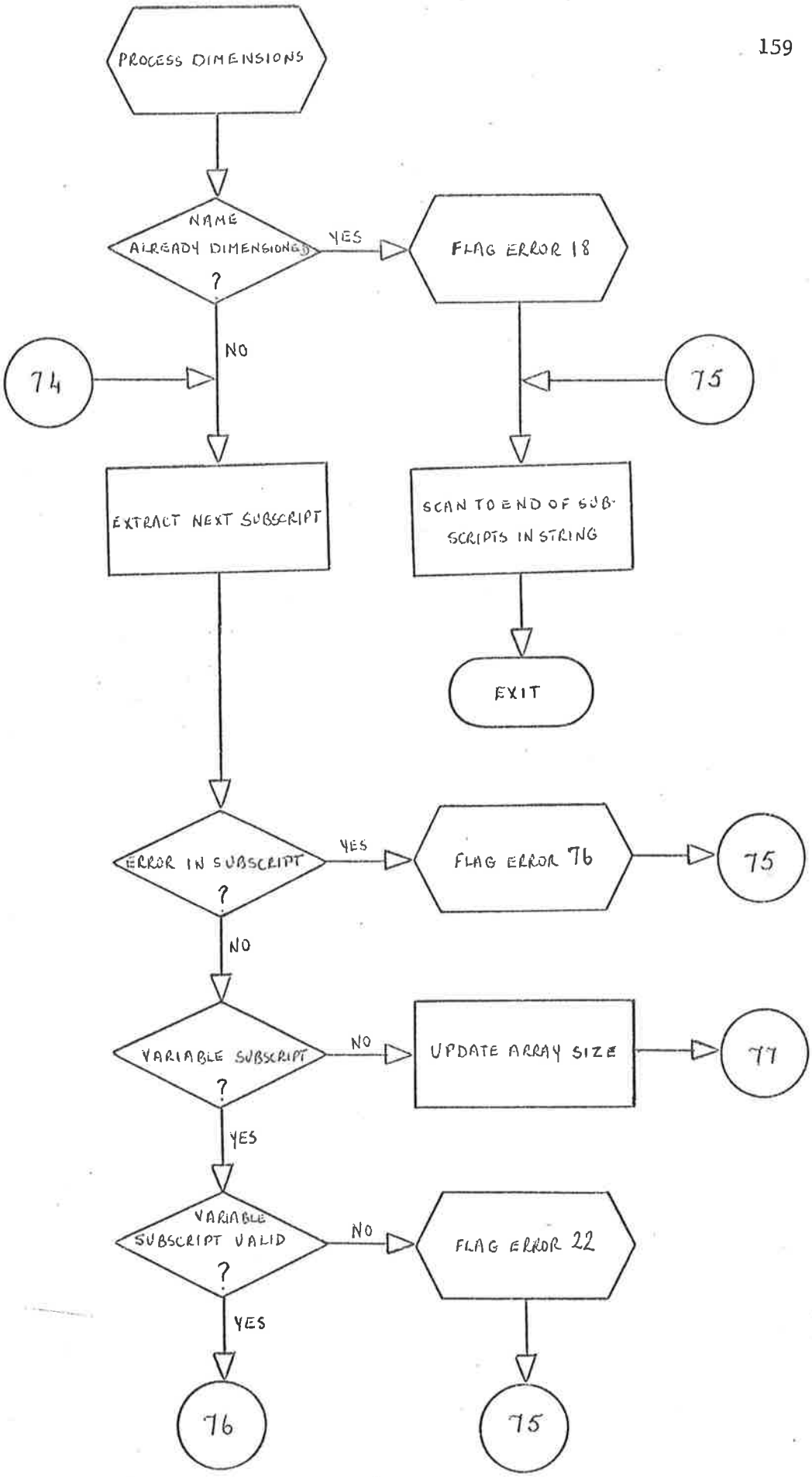

67
REAL NUMBER ?
NO
65
YES
ANOTHER NO. AFTER COMMA ?
NO
65
YES
XVII
SYNTAX CHECK NUMBER
VALID ?
NO
65
YES
REAL NUMBER ?
NO
65
YES
END OF COMPLEX NUMBER ?
NO
65
YES
70

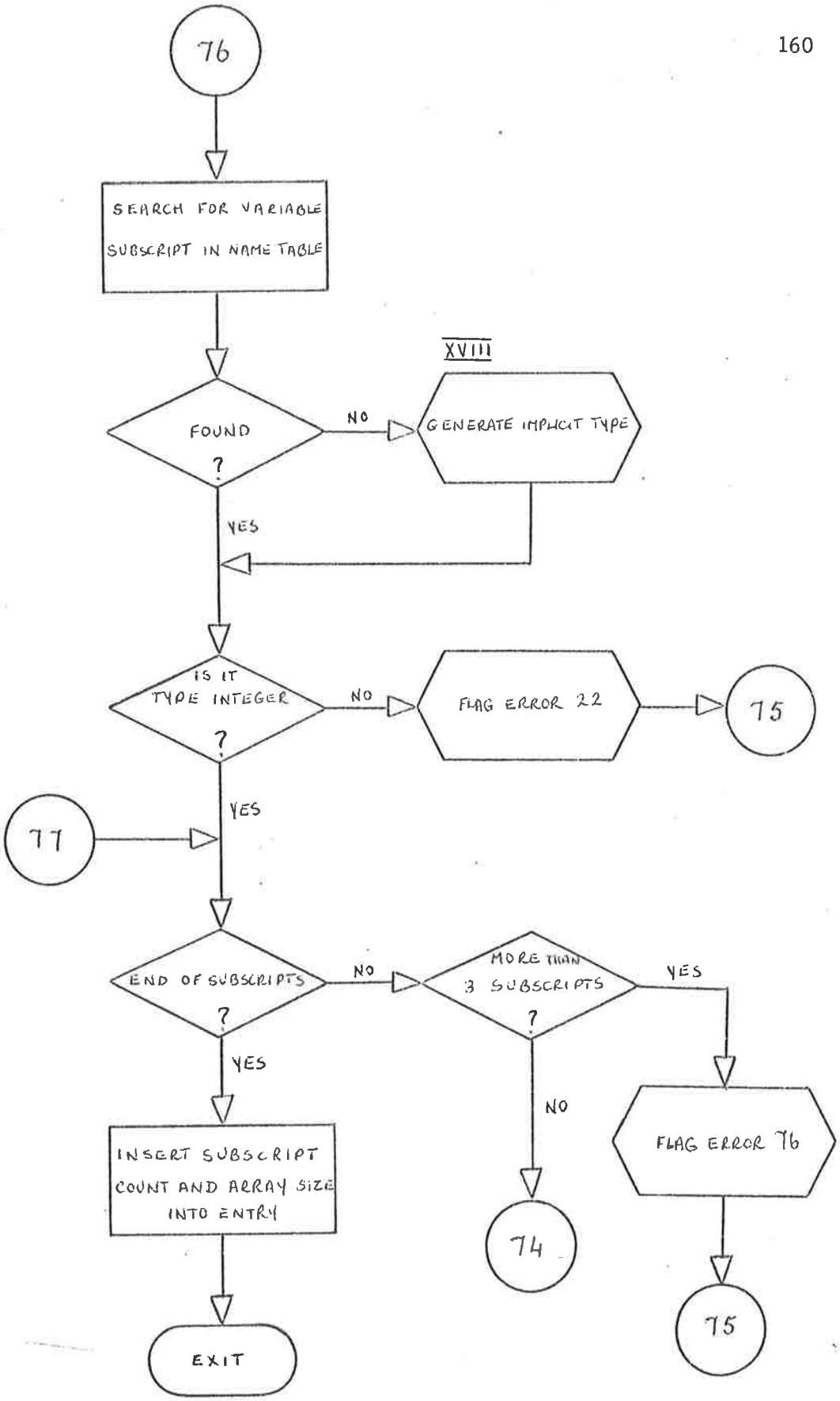

68
INTEGER CONSTANT ?
YES
REPEAT COUNT ?
NO
70
YES
69
NO
XVII
REPEAT COUNT ALREADY SPECIFIED ?
YES
65
SYNTAX CHECK NUMBER
NO
UPDATE VALUE LIST ELEMENT COUNT
65
NO
VALID ?
64
70
YES
INCREMENT VALUE LIST ELEMENT COUNT
END OF VALUE LIST ?
NO
64
YES
71


71
VARIABLE AND VALUE LIST SAME LENGTH ?
NO
FLAG ERROR 28
YES
END OF DATA STATEMENT ?
YES
1
NO
54


72
READ INPUT TABLE PARAMETERS
XIX
IS EXECUTABLE STATEMENT FLAG SET ?
NO
EXECUTE IDLE LOOP
1
YES
73


73
IS CURRENT SUB-PROGRAM BLOCK DATA ?
YES
IS SUBPROGRAM END FLAG SET ?
YES
NO
NO
XX
CALL PROCESSOR FOR "FEA"
REQUEST MONITOR TO LOAD PROGRAM "PPH"
EXECUTE IDLE LOOP
XX
CALL PROCESSOR FOR "MSA"
CLEAR PROCESSOR STATUS TABLE ENTRY FOR THIS PROGRAM
CLEAR "DSA" COUNT IN PROCESSOR COUNT WORD
DROP THIS PROGRAM


PROCESS DIMENSIONS
NAME ALREADY DIMENSIONED ?
YES
FLAG ERROR 18
NO
74
75
EXTRACT NEXT SUBSCRIPT
SCAN TO END OF SUB-SCRIPTS IN STRING
EXIT
ERROR IN SUBSCRIPT ?
YES
FLAG ERROR 76
75
NO
VARIABLE SUBSCRIPT ?
NO
UPDATE ARRAY SIZE
77
YES
VARIABLE SUBSCRIPT VALID ?
NO
FLAG ERROR 22
YES
76
75


76
SEARCH FOR VARIABLE SUBSCRIPT IN NAME TABLE
XVIII
FOUND ?
NO
GENERATE IMPLICIT TYPE
YES
IS IT TYPE INTEGER ?
NO
FLAG ERROR 22
75
77
YES
END OF SUBSCRIPTS ?
NO
MORE THAN 3 SUBSCRIPTS ?
YES
YES
NO
INSERT SUBSCRIPT COUNT AND ARRAY SIZE INTO ENTRY
74
FLAG ERROR 76
EXIT
75

9. PROGRAM FEA - FORTRAN EXPRESSION ANALYSER

9.0 Introduction

This program further processes statements containing FORTRAN expressions, such as assignment statements, CALL statements and the expression parts of IF statements. It is loaded on request by programs DSA and PCR when either of them detects that the processor 2 communication buffer is non-empty.

This program continuously executes its major loop, processing one processor communication buffer entry each time, until the buffer is empty. It then checks whether any other processors are active and if not, requests the windup routine, program PP4, to be loaded in order to complete the compilation of the current subprogram and produce a listing. In either case, it drops itself.

Originally the program was written to syntax check all expressions by applying a series of productions (expressed in Floyd Production Language - see Appendix 1) to them in an attempt to reduce them to a "distinguished" symbol, representing a syntactically valid expression in the language. It was felt that this approach was warranted because of the inherent complexity of logical and masking (see note XIV of 9.1) expressions involving logical and relational operators in addition to arithmetic operators. However, on testing the program, it was found that simple expressions, even as simple as I + 1, required large numbers of comparisons and reductions before their complete reduction. Since it is possible to syntax check pure arithmetic expressions, and logical expressions involving only one relational operator, far more directly, and hence

rapidly, program FEA was rewritten. In its new form, only expressions involving logical operators and/or the exponentiation operator are processed via the production technique. In this respect, the exponentiation operator is treated differently from the other arithmetic operators. This is because in the source language as implemented (that is, Control Data Corporation RUN FORTRAN), any combination of operand types is allowable for simple operators (see note XII of 9.1) whereas only certain combinations of operand types are valid in the case of exponentiation (see note XIII of 9.1). Thus, for expressions containing the latter, it is necessary to determine the type of each partial result in order that the validity of the types of the operands to the exponentiation may be checked. Therefore, since the method used to syntax check simple expressions is unable to keep track of the types of partial results whereas this is easily implemented in the production technique by means of a last-in-first-out stack, called the type stack (see note IV of 9.1), expressions containing the exponentiation operator are processed by the latter technique. During its lexical analysis of statements, program GSR sets a flag in the last byte of the string header of processor 3 type entries containing logical and/or exponentiation operators. Program FEA uses this flag to determine whether or not to process the associated entry by means of the Floyd Production Language program. The same applies to statement function references (except that these are not recognised by program GSR but rather by this

program) for which a check must be made that the type of each actual argument is the same as the corresponding formal argument (see note XVIII of 9.1).

There is another last-in-first-out stack, called the function stack, which is used to save all relevant data regarding the current function reference, within a group of nested function references, when another nested reference is detected.

Statement function definition statements are processed by this program rather than program DSA for two reasons. Firstly, program GSR is unable to distinguish between statement function definition statements and assignment statements. This is because the distinction depends on the nature of the element to the left of the equals. If this element is a name followed by an open parenthesis and the name has not been dimensioned in a declarative, that name is being defined (correctly or otherwise) as a statement function name. Program GSR cannot determine this as it does not have access to the name table. Secondly, statement function definition statements involve expressions and only this program can process these.

9.1 Explanatory Notes to the Flow-Chart

Although not explicitly stated in the accompanying flow-chart, references to all tables are made only after access to the required table has been successfully requested, using the procedure described in 5.1.0.

The following notes refer to the flow-chart:

I. The central memory table full flag is set to indicate that the main area of the name table is full. Eventually program CTM (see 11.0) will be loaded to perform the central memory table move operation.

II. An open parenthesis, which is not the start of a function argument list, is assumed to be the possible beginning of a complex constant. It is confirmed as a complex constant (although not necessarily a syntactically valid one) if a comma is detected in the rest of the string body before the next occurrence of a close parenthesis.

III. The syntax checking of a number is performed by the use of the same state table as in program DSA. See note XVII of 8.1 for details.

IV. If the syntactic analysis of the current expression is being carried out by the simulated "execution" of the Floyd Production Language program, the type of the current constant or identifier is stored at the top of the type stack. See also note XI below.

V. The pointers are saved in case the open parenthesis is not the beginning of a complex constant. In this case, it will be necessary to backtrack to this position in the string body. See also note II above.

VI. The complex constant is syntax checked by checking each component individually, using the routine described in note III above.

VII. Nested function references are handled by means of a last-in-first-out stack, in which each entry holds information about the previous function reference, such as the number of arguments defined for the function (if the function had previously been defined within the current subprogram), the number of actual arguments detected so far in the current reference, and a pointer to the name table entry for the function name. The stack is popped at the end of the current function reference.

VIII. When a name appears in a type declarative, it is assumed to be a variable and is defined accordingly in the name table. Provided that its definition did not specify subscripts and that it has not subsequently been actually used as a variable, it is permissible for the name to be used as a function name after which its name table entry is appropriately altered.

IX. Subscripts may be expressions in which operands are simple integer variables and integer constants while operators are addition, subtraction, multiplication and division only.

X. If it is not a concatenation character, it must be a text constant.

XI. The syntactic analysis of the FORTRAN expression in the string body will be performed by the simulated "execution" of the Floyd Product Language program, if the FPL switch in the last byte of the string header is set. See also 9.0.

XII. When a simple operator (that is, +, -, * or /) is reduced from the stack, it is necessary to determine the type of the partial result produced by the execution of that operation. Since the respective types of the two operands involved in the operation appear at the top of the type stack, the required type is simply that one of the top two type stack entries which appears highest in the following hierarchical list of types:

COMPLEX

DOUBLE PRECISION

REAL

INTEGER

LOGICAL

The type, once determined, replaces the top two entries in the type stack.

XIII. The type of the partial result in the case of exponentiation reduction is according to the following table, which gives the type of A**B:

Type of A	Type of B: I	R	D	C	L
I	I	*	*	*	*
R	R	R	D	*	*
D	D	D	D	*	*
C	C	*	*	*	*
L	I	*	*	*	*

where

I is INTEGER

R is REAL

D is DOUBLE PRECISION

C is COMPLEX

L is LOGICAL

* is an invalid combination of operand types.

Again, the type, once determined, replaces the top two entries in the type stack.

XIV. An immediate exit is made from the routine PROCESS EXPRESSION, on the successful reduction of the string to either an arithmetic, logical or masking expression. A masking expression is a generalized form of the logical expression. It may be used as the right-hand-side of an assignment statement and is evaluated by performing bit-by-bit logical arithmetic on the operands within the expression.

XV. The "parametric function symbol" indicates the occurrence of a function or subroutine name, without parameters, as an actual argument of some other function or subroutine.

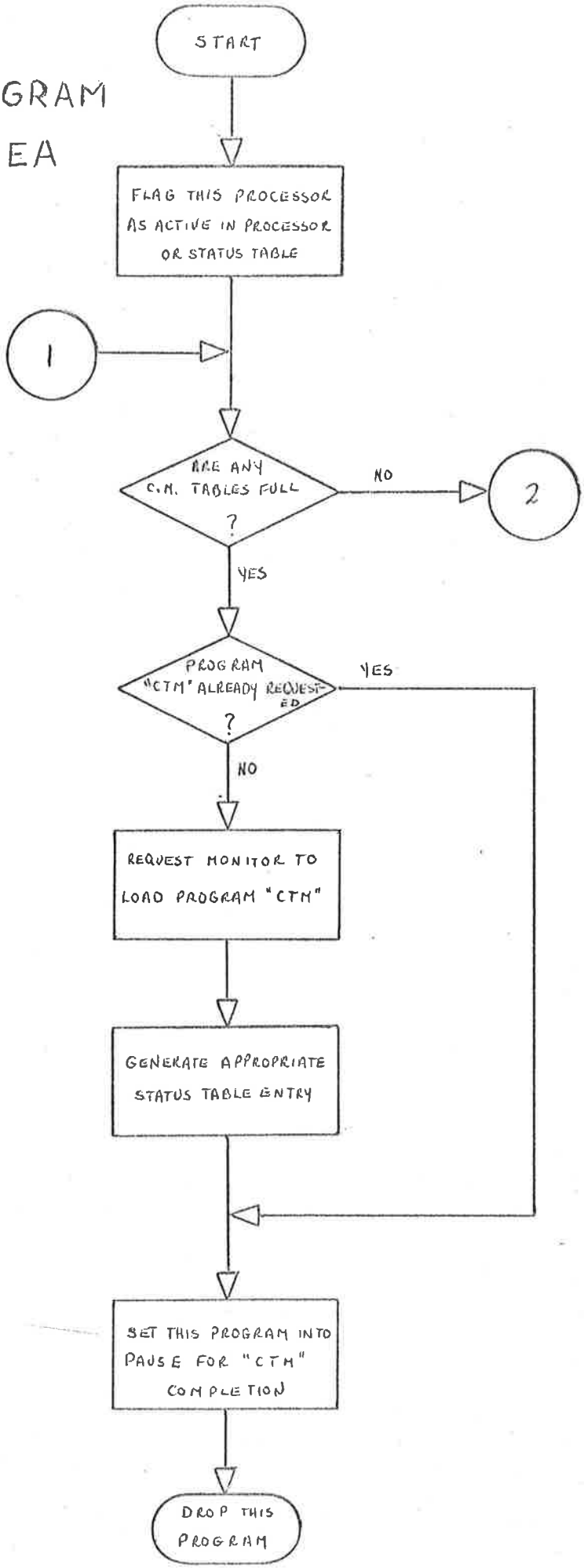
XVI. If a name is not already defined in the name table, it is assumed to be a variable name and an entry is accordingly created for it in the table. The entry may later be altered to a function or subroutine name type should it become necessary (see note VIII above). This has been done in order to reduce the number and duration of accesses to the name table by program FEA without the problems which would

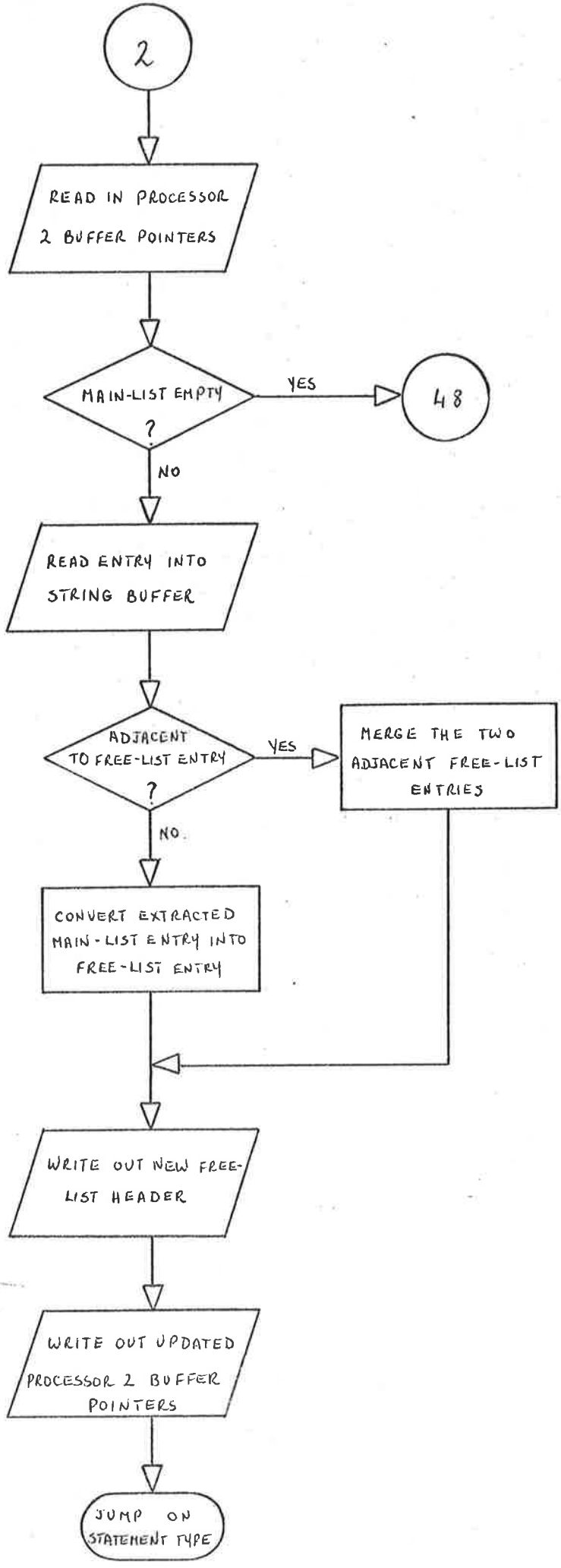
ensue from the generation of partial entries, which may then be accessed by other programs before completion, with dire results.

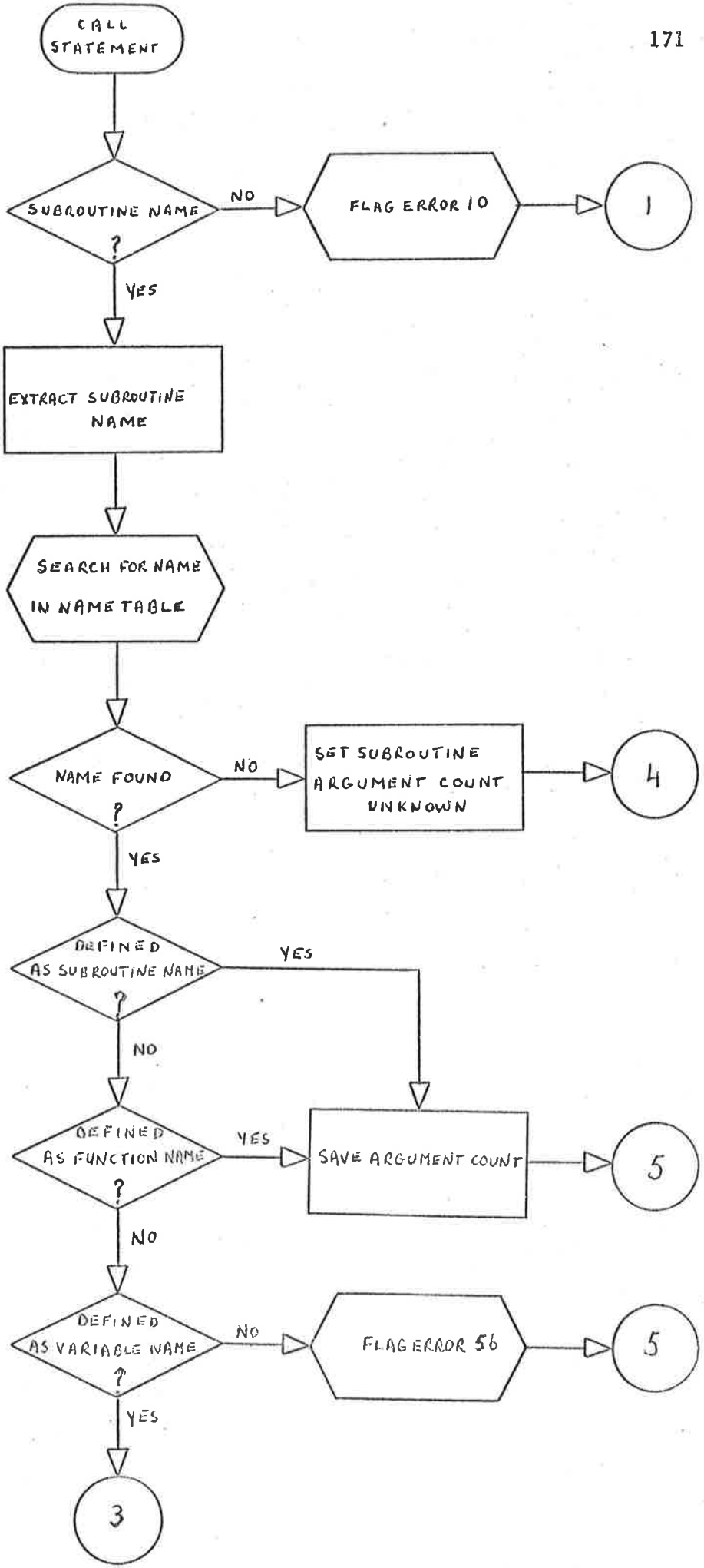
XVII. Since the types of the actual arguments of a statement function reference must be checked against the type of the corresponding formal argument and since the type of an expression can only be determined if it is being processed by the Floyd Production Language program technique, the FPL switch is turned on (if it is not already on) as soon as the first statement function reference within a group of nested function references is detected. When the function stack entry for this first statement function reference is popped, the FPL switch will be cleared (if it was not originally set).

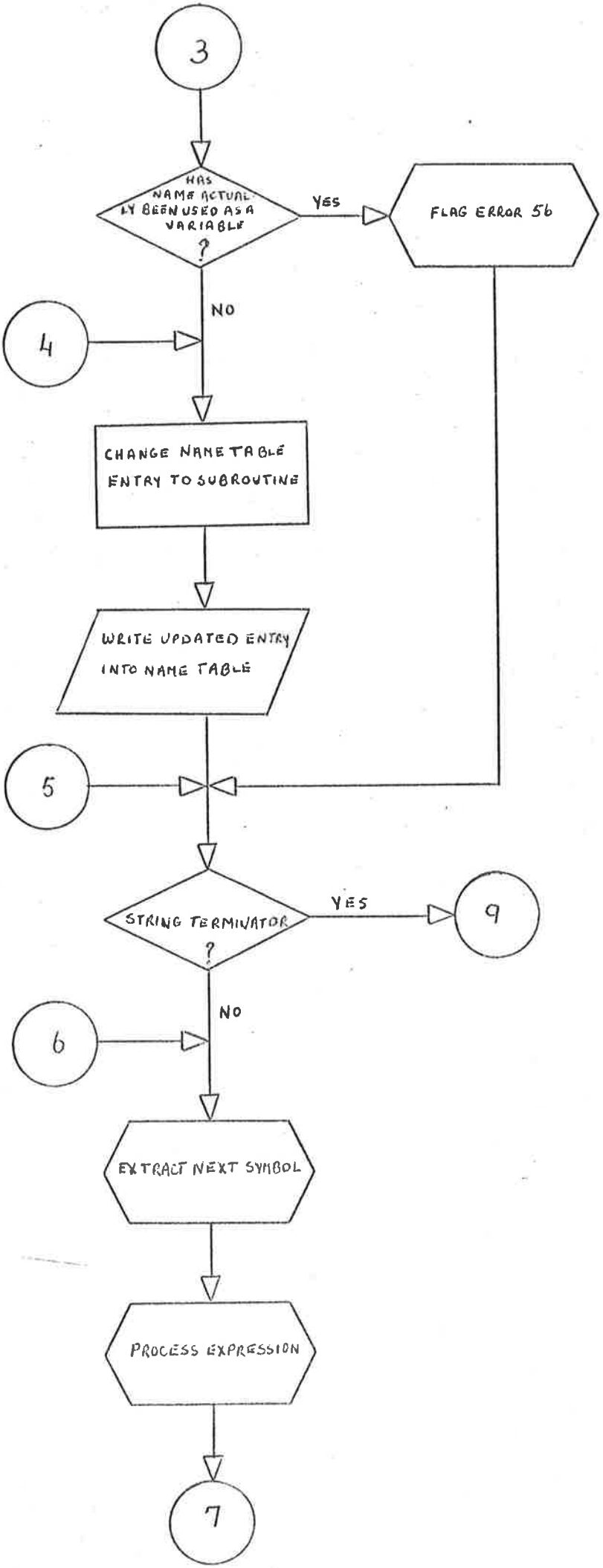
PROGRAM

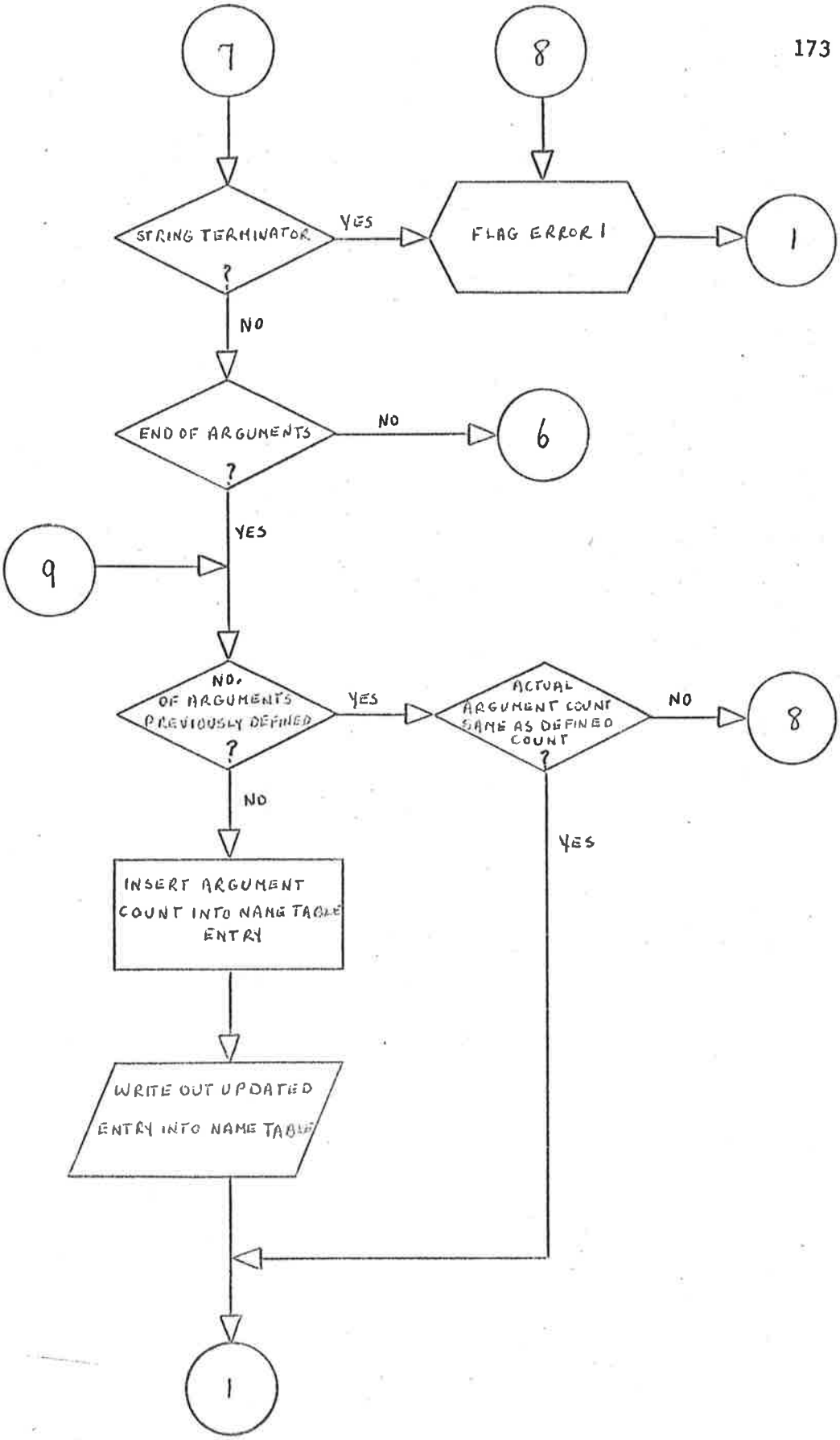
FEA

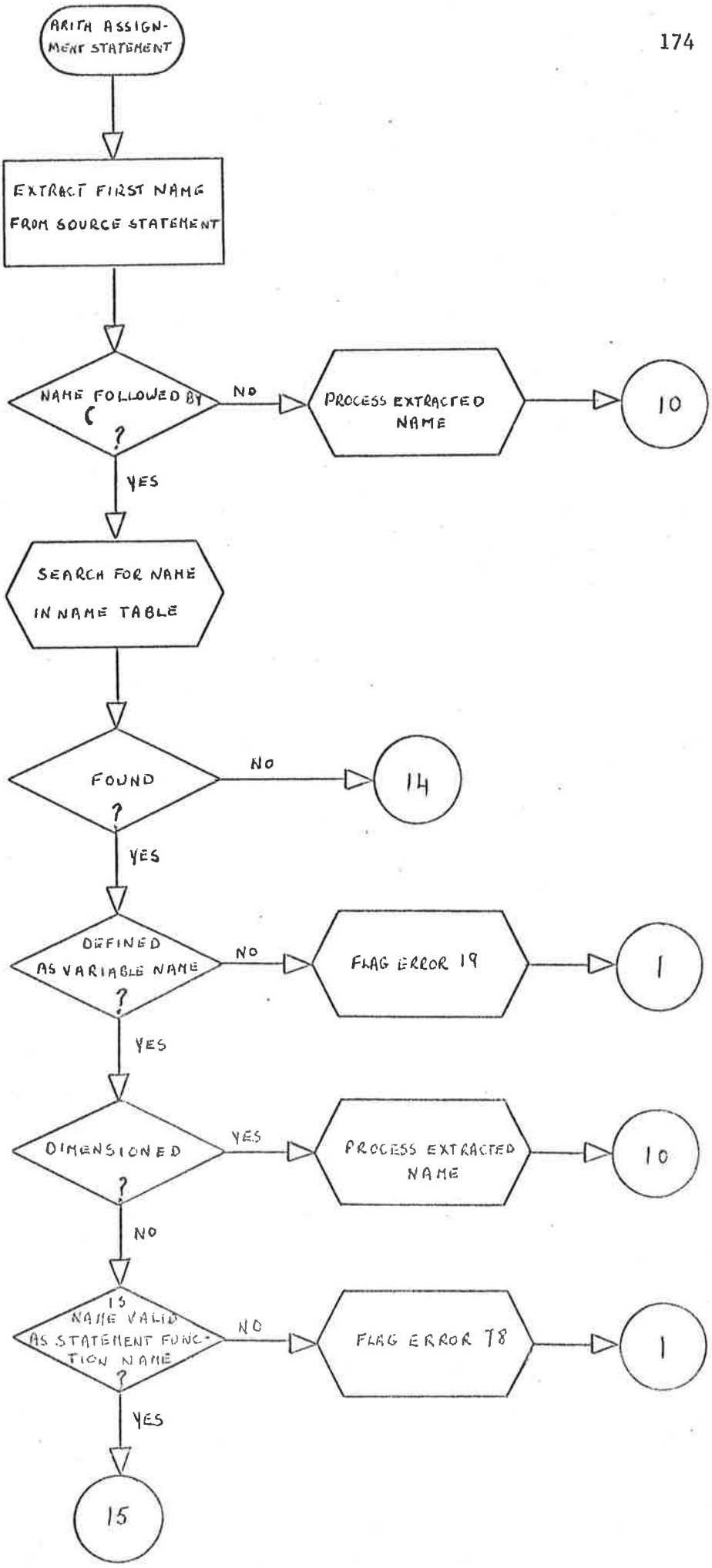

START
FLAG THIS PROCESSOR AS ACTIVE IN PROCESSOR OR STATUS TABLE
1
ARE ANY C.M. TABLES FULL ?
NO
2
YES
PROGRAM "CTM" ALREADY REQUESTED ?
YES
NO
REQUEST MONITOR TO LOAD PROGRAM "CTM"
GENERATE APPROPRIATE STATUS TABLE ENTRY
SET THIS PROGRAM INTO PAUSE FOR "CTM" COMPLETION
DROP THIS PROGRAM

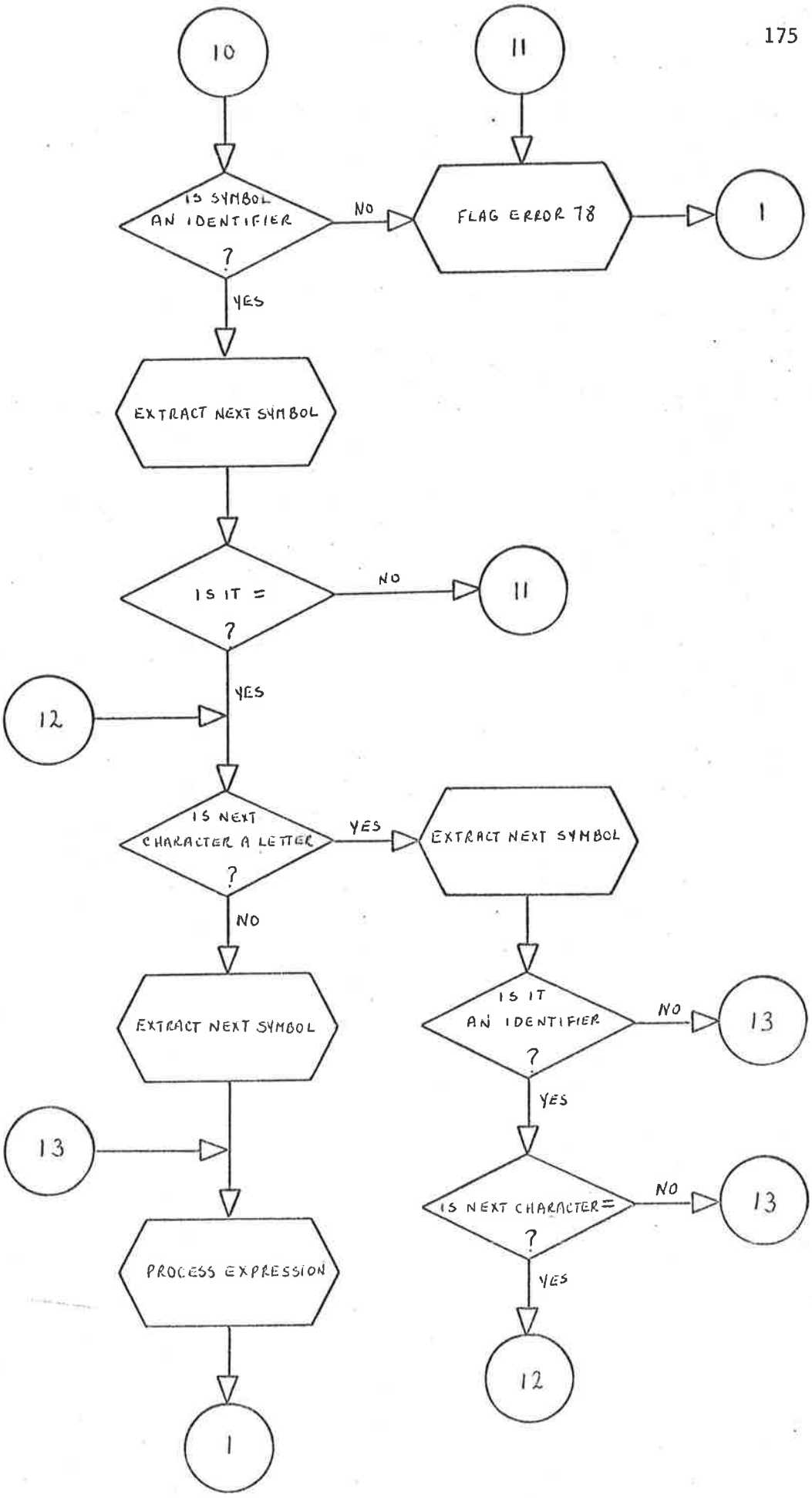

2
READ IN PROCESSOR 2 BUFFER POINTERS
MAIN-LIST EMPTY ?
YES
48
NO
READ ENTRY INTO STRING BUFFER
ADJACENT TO FREE-LIST ENTRY ?
YES
MERGE THE TWO ADJACENT FREE-LIST ENTRIES
NO
CONVERT EXTRACTED MAIN-LIST ENTRY INTO FREE-LIST ENTRY
WRITE OUT NEW FREE-LIST HEADER
WRITE OUT UPDATED PROCESSOR 2 BUFFER POINTERS
JUMP ON STATEMENT TYPE

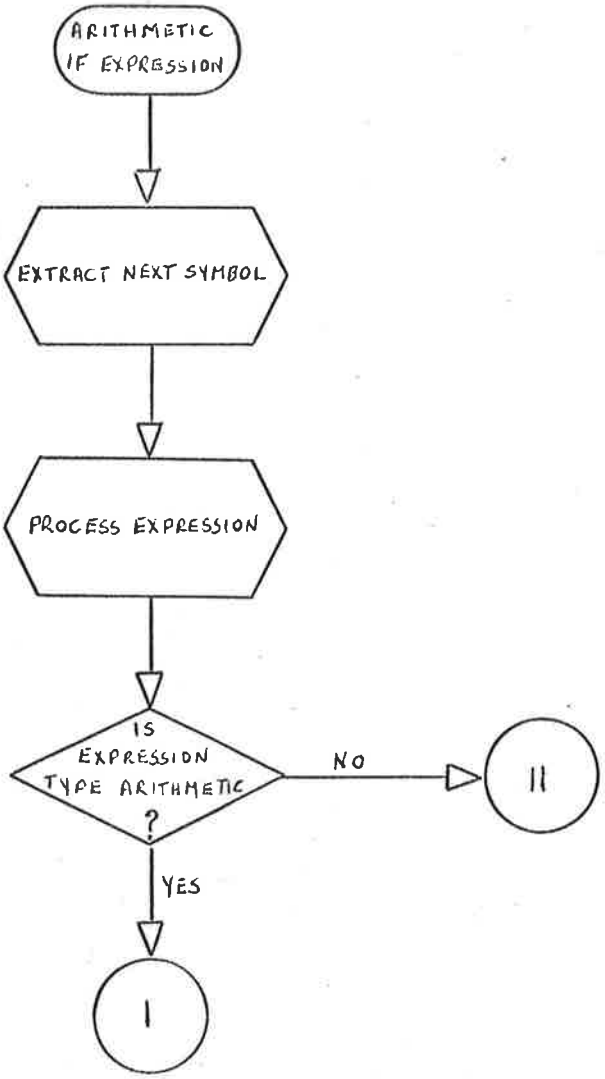

CALL STATEMENT
SUBROUTINE NAME ?
NO
FLAG ERROR 10
1
YES
EXTRACT SUBROUTINE NAME
SEARCH FOR NAME IN NAME TABLE
NAME FOUND ?
NO
SET SUBROUTINE ARGUMENT COUNT UNKNOWN
4
YES
DEFINED AS SUBROUTINE NAME ?
YES
NO
DEFINED AS FUNCTION NAME ?
YES
SAVE ARGUMENT COUNT
5
NO
DEFINED AS VARIABLE NAME ?
NO
FLAG ERROR 56
5
YES
3

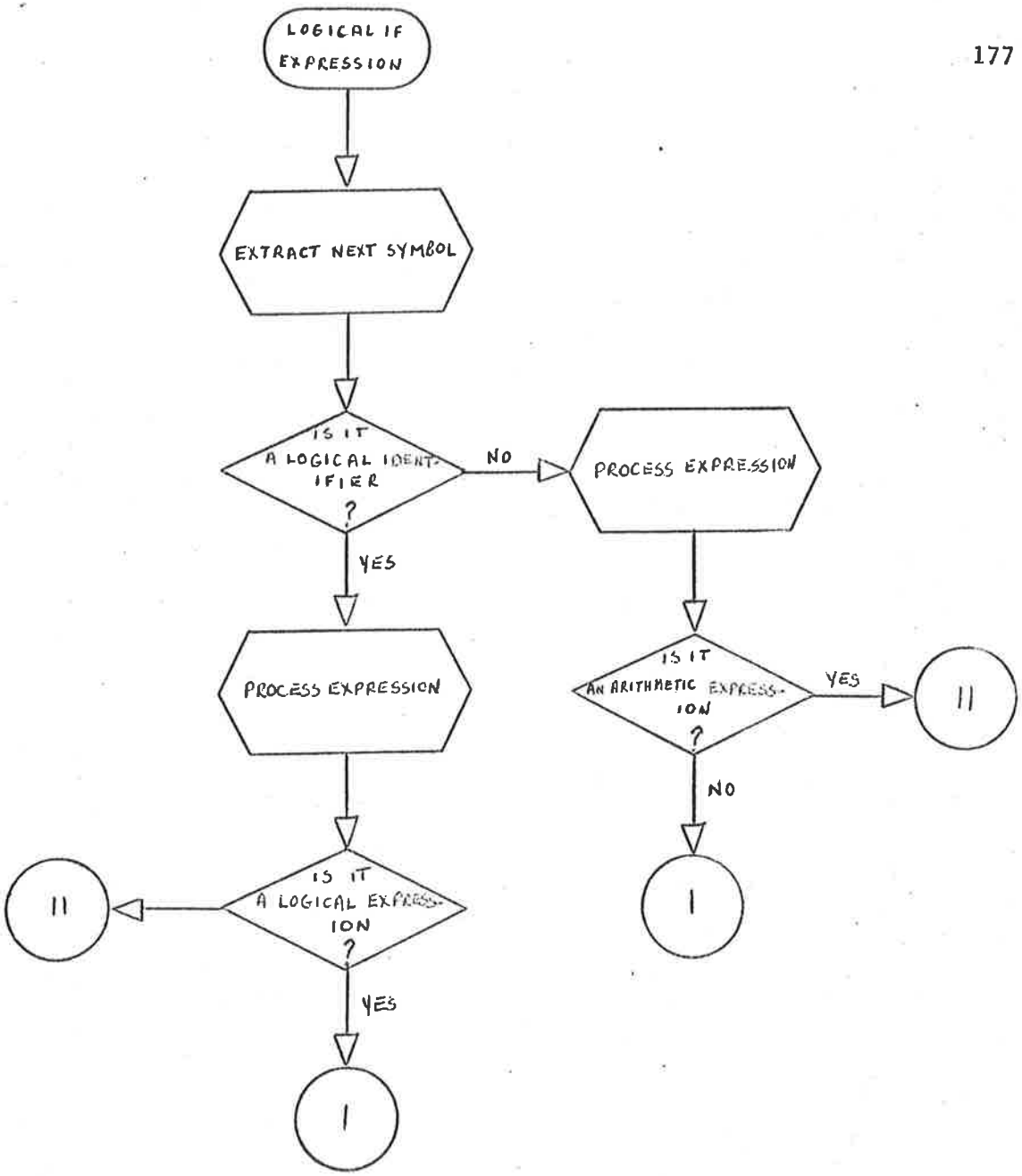

3
HAS NAME ACTUALLY BEEN USED AS A VARIABLE ?
YES
FLAG ERROR 56
NO
4
CHANGE NAME TABLE ENTRY TO SUBROUTINE
WRITE UPDATED ENTRY INTO NAME TABLE
5
STRING TERMINATOR ?
YES
9
NO
6
EXTRACT NEXT SYMBOL
PROCESS EXPRESSION
7

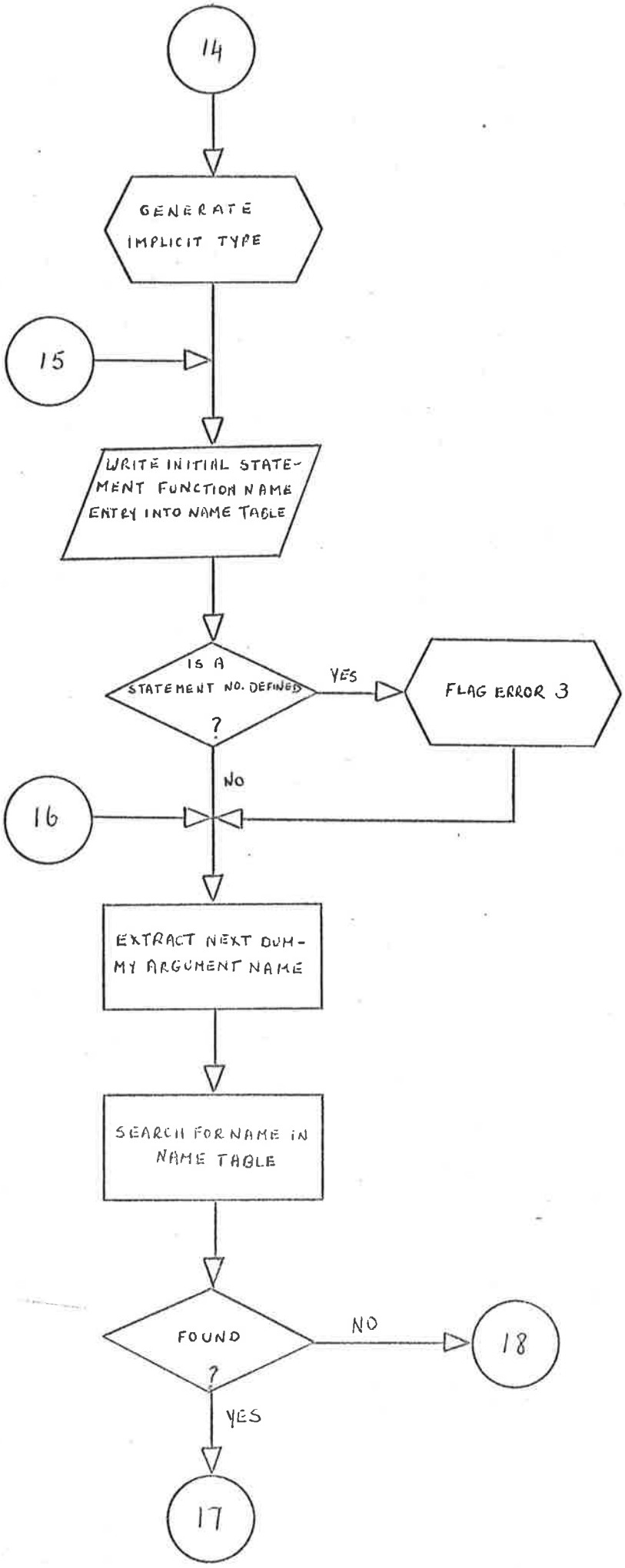

7
8
STRING TERMINATOR ?
YES
FLAG ERROR 1
1
NO
END OF ARGUMENTS ?
NO
6
YES
9
NO. OF ARGUMENTS PREVIOUSLY DEFINED ?
YES
ACTUAL ARGUMENT COUNT SAME AS DEFINED COUNT ?
NO
8
NO
YES
INSERT ARGUMENT COUNT INTO NAME TABLE ENTRY
WRITE OUT UPDATED ENTRY INTO NAME TABLE
1

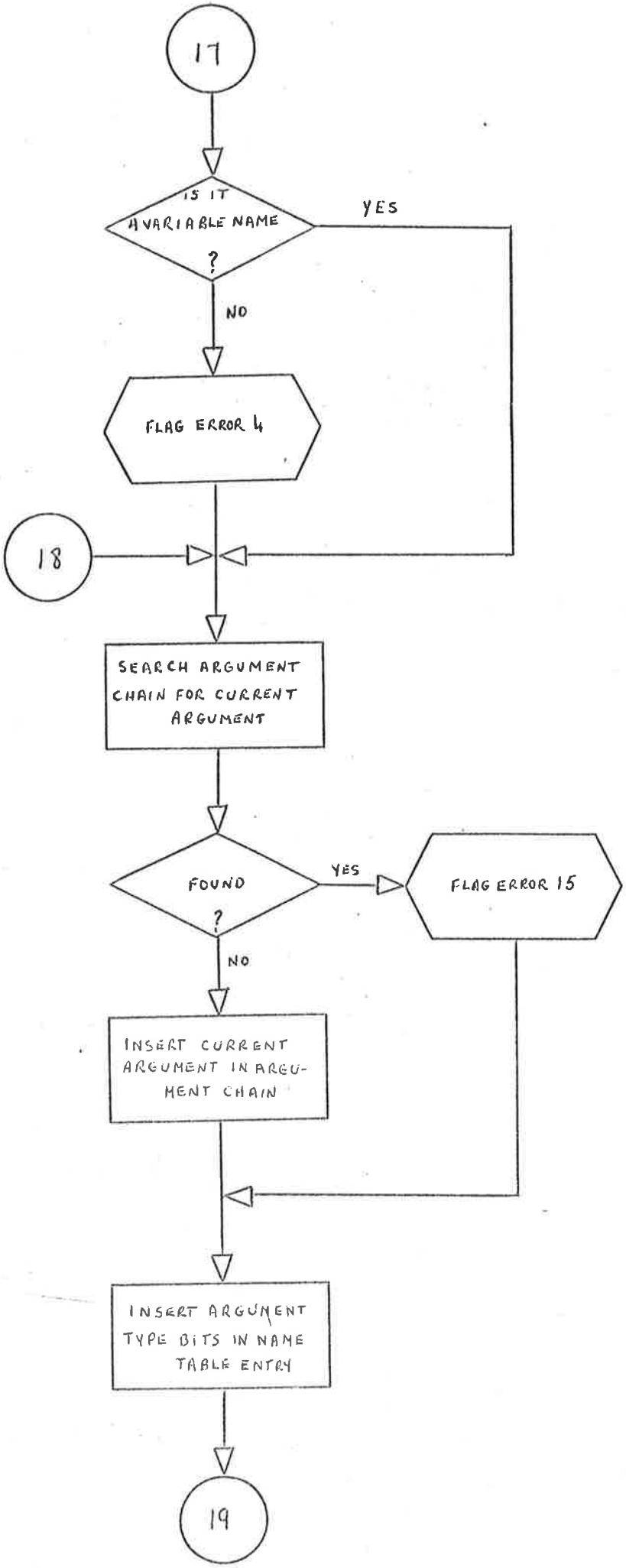

ARITH ASSIGN-MENT STATEMENT
EXTRACT FIRST NAME FROM SOURCE STATEMENT
NAME FOLLOWED BY (?
NO
PROCESS EXTRACTED NAME
10
YES
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
14
YES
DEFINED AS VARIABLE NAME ?
NO
FLAG ERROR 19
1
YES
DIMENSIONED ?
YES
PROCESS EXTRACTED NAME
10
NO
IS NAME VALID AS STATEMENT FUNC-TION NAME ?
NO
FLAG ERROR 78
1
YES
15

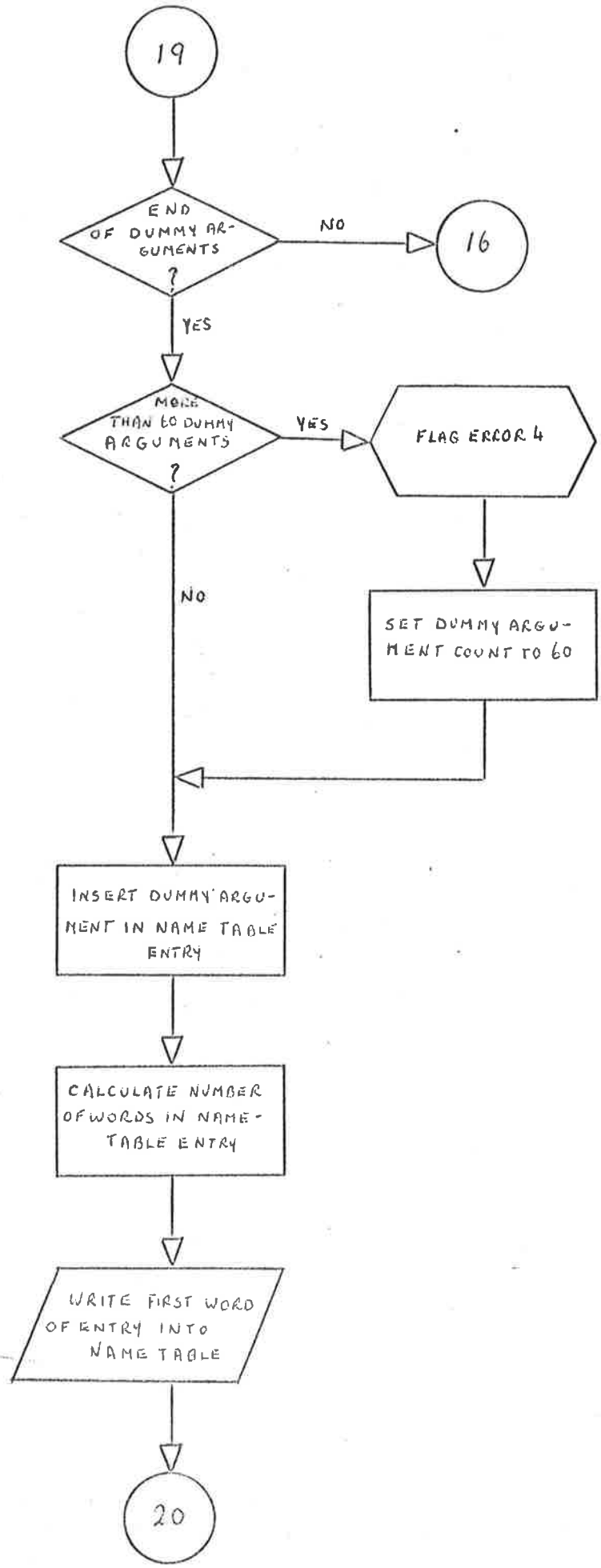

10
11
IS SYMBOL AN IDENTIFIER ?
NO
FLAG ERROR 78
1
YES
EXTRACT NEXT SYMBOL
IS IT = ?
NO
11
12
YES
IS NEXT CHARACTER A LETTER ?
YES
EXTRACT NEXT SYMBOL
NO
EXTRACT NEXT SYMBOL
IS IT AN IDENTIFIER ?
NO
13
YES
13
IS NEXT CHARACTER = ?
NO
13
PROCESS EXPRESSION
YES
12
1

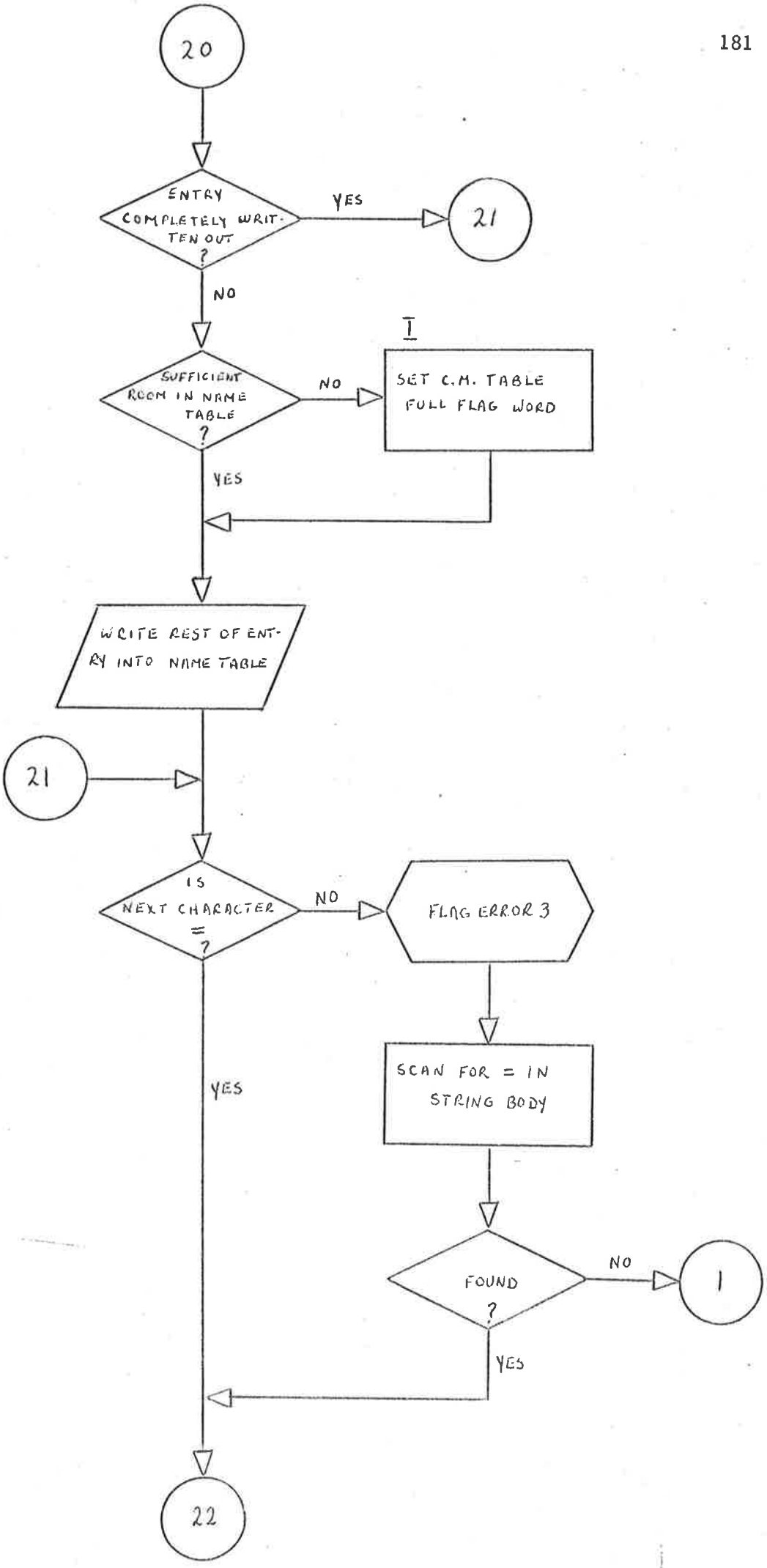

ARITHMETIC IF EXPRESSION
EXTRACT NEXT SYMBOL
PROCESS EXPRESSION
IS EXPRESSION TYPE ARITHMETIC ?
NO
II
YES
I

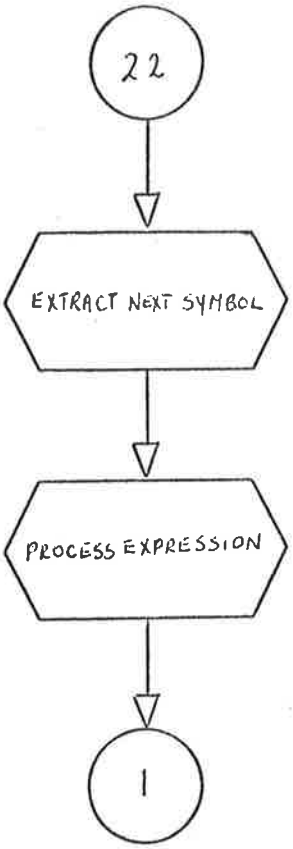

LOGICAL IF EXPRESSION
EXTRACT NEXT SYMBOL
IS IT A LOGICAL IDENTIFIER ?
NO
PROCESS EXPRESSION
YES
PROCESS EXPRESSION
IS IT AN ARITHMETIC EXPRESSION ?
YES
II
NO
I
IS IT A LOGICAL EXPRESSION ?
II
YES
I

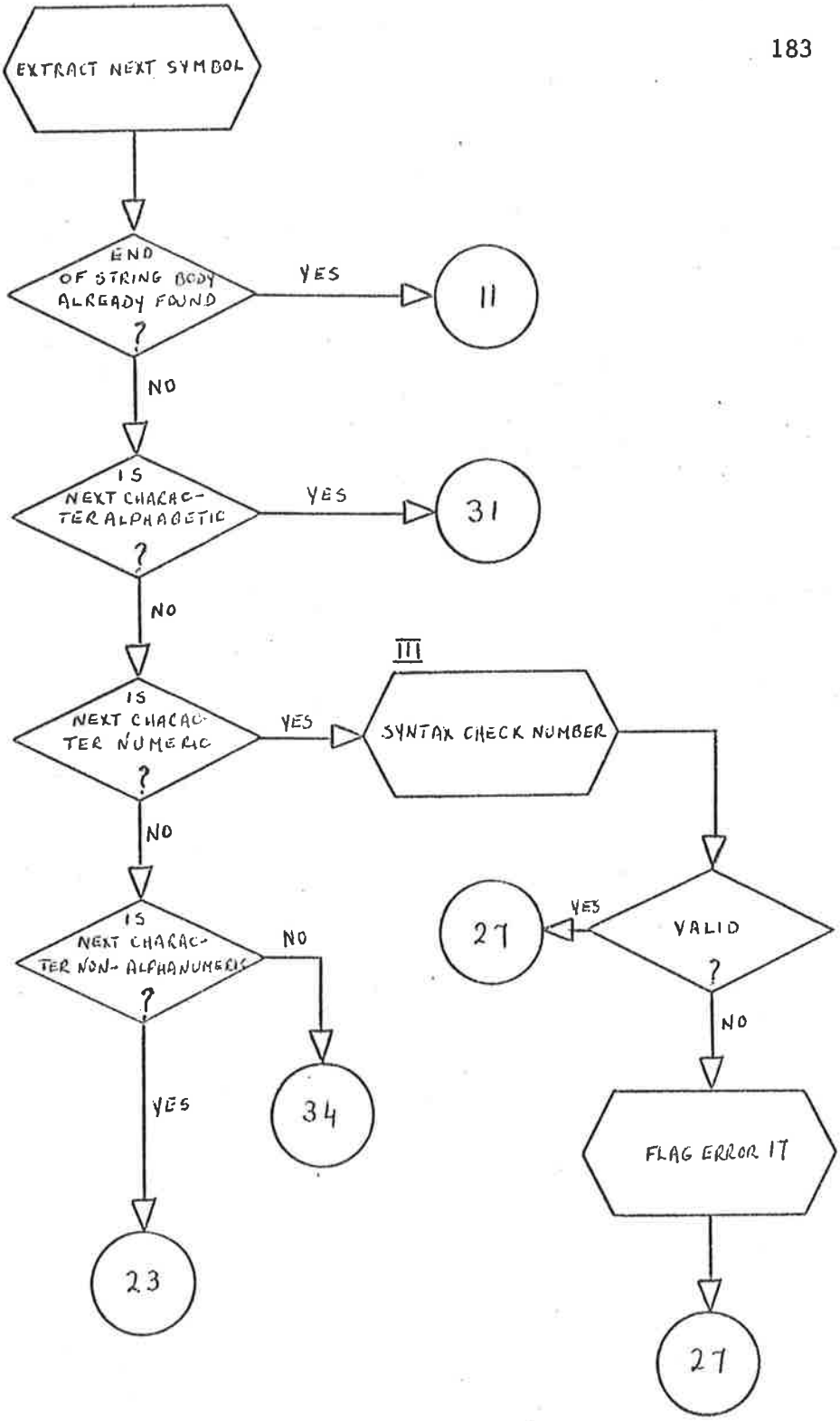

14
GENERATE IMPLICIT TYPE
15
WRITE INITIAL STATEMENT FUNCTION NAME ENTRY INTO NAME TABLE
IS A STATEMENT NO. DEFINED ?
YES
FLAG ERROR 3
NO
16
EXTRACT NEXT DUMMY ARGUMENT NAME
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
18
YES
17

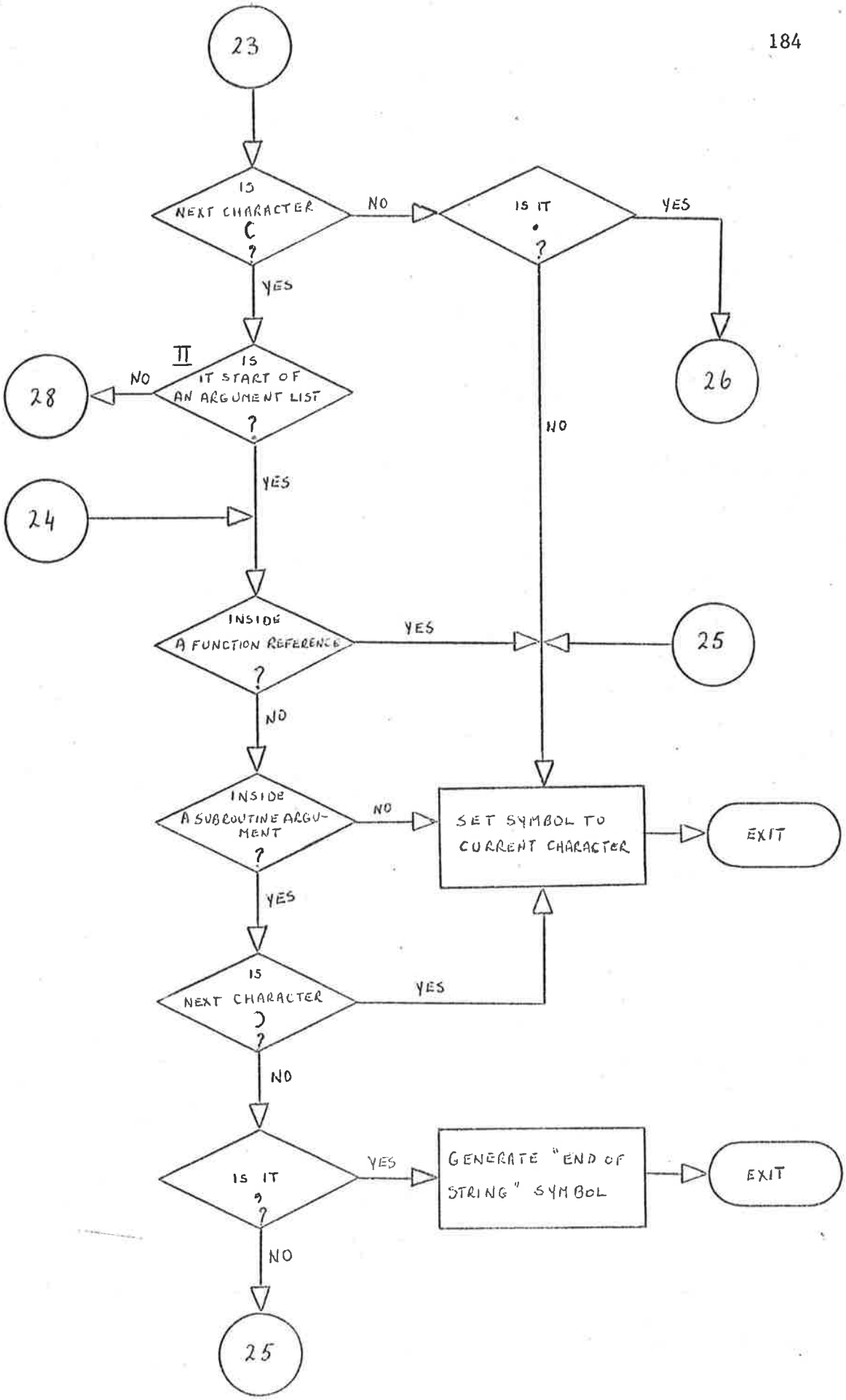

17
IS IT A VARIABLE NAME ?
YES
NO
FLAG ERROR 4
18
SEARCH ARGUMENT CHAIN FOR CURRENT ARGUMENT
FOUND ?
YES
FLAG ERROR 15
NO
INSERT CURRENT ARGUMENT IN ARGUMENT CHAIN
INSERT ARGUMENT TYPE BITS IN NAME TABLE ENTRY
19

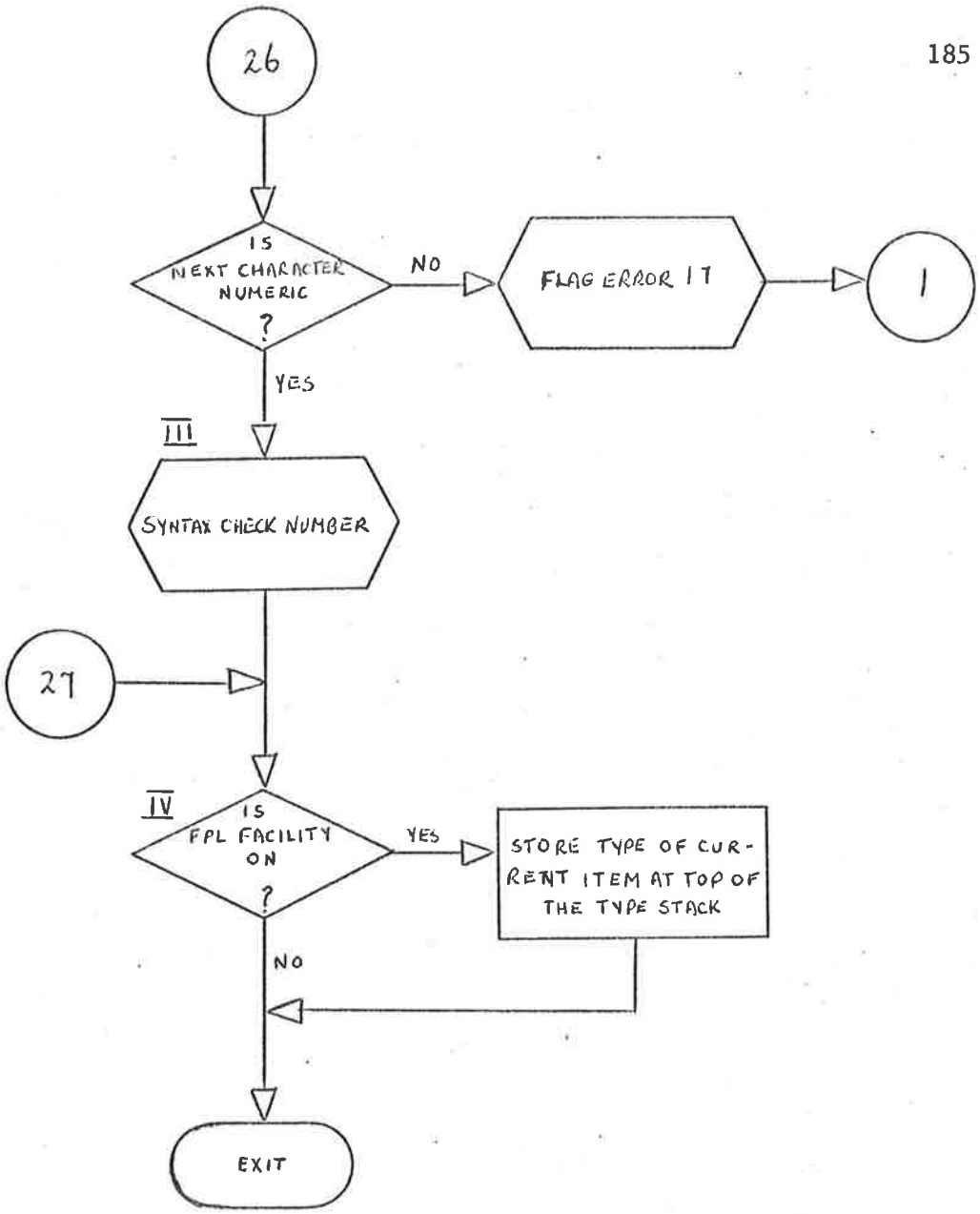

19
END OF DUMMY ARGUMENTS ?
NO
16
YES
MORE THAN 60 DUMMY ARGUMENTS ?
YES
FLAG ERROR 4
NO
SET DUMMY ARGUMENT COUNT TO 60
INSERT DUMMY ARGUMENT IN NAME TABLE ENTRY
CALCULATE NUMBER OF WORDS IN NAME TABLE ENTRY
WRITE FIRST WORD OF ENTRY INTO NAME TABLE
20

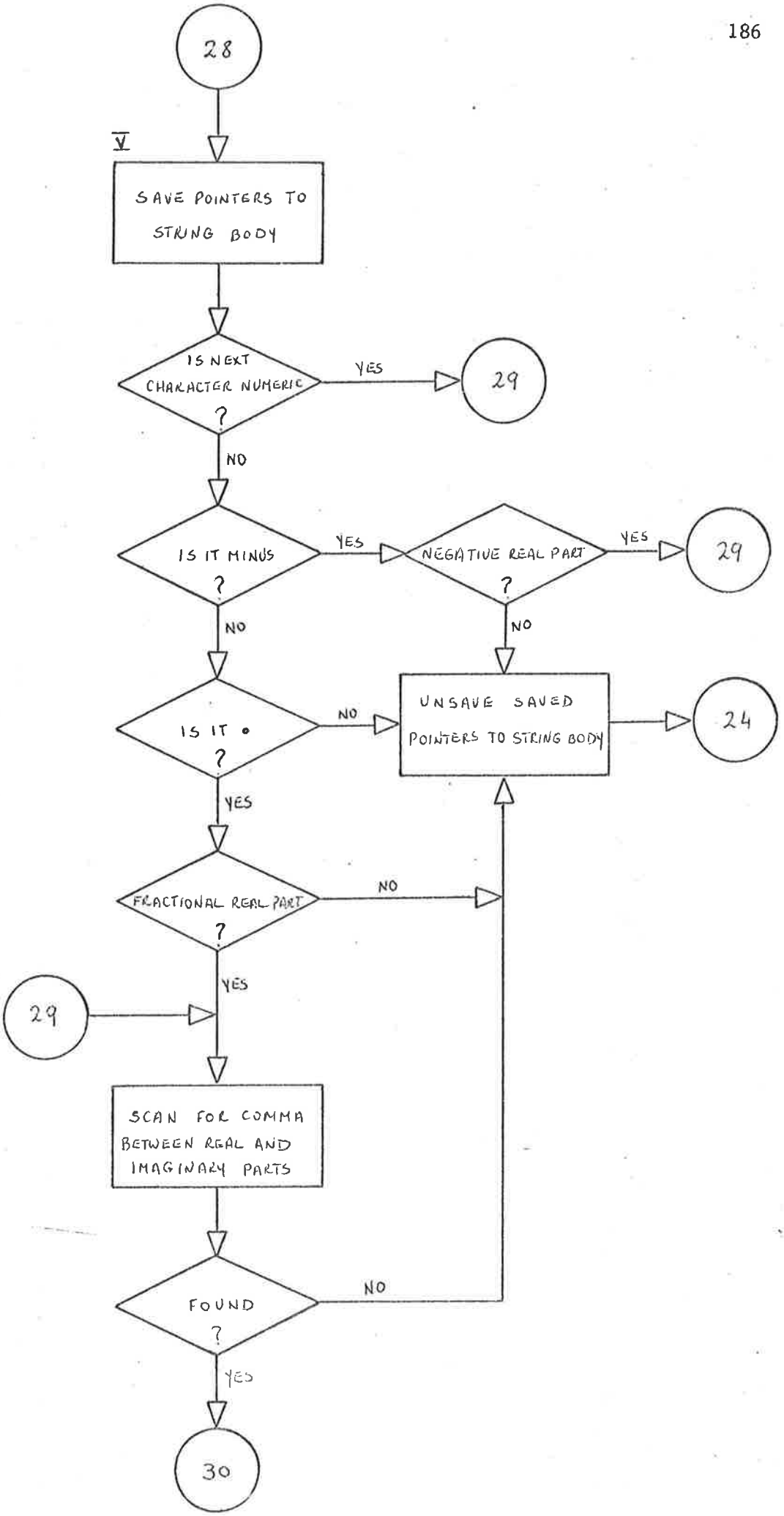

20
ENTRY COMPLETELY WRITTEN OUT ?
YES
21
NO
I
SUFFICIENT ROOM IN NAME TABLE ?
NO
SET C.M. TABLE FULL FLAG WORD
YES
WRITE REST OF ENTRY INTO NAME TABLE
21
IS NEXT CHARACTER = ?
NO
FLAG ERROR 3
SCAN FOR = IN STRING BODY
YES
FOUND ?
NO
1
YES
22

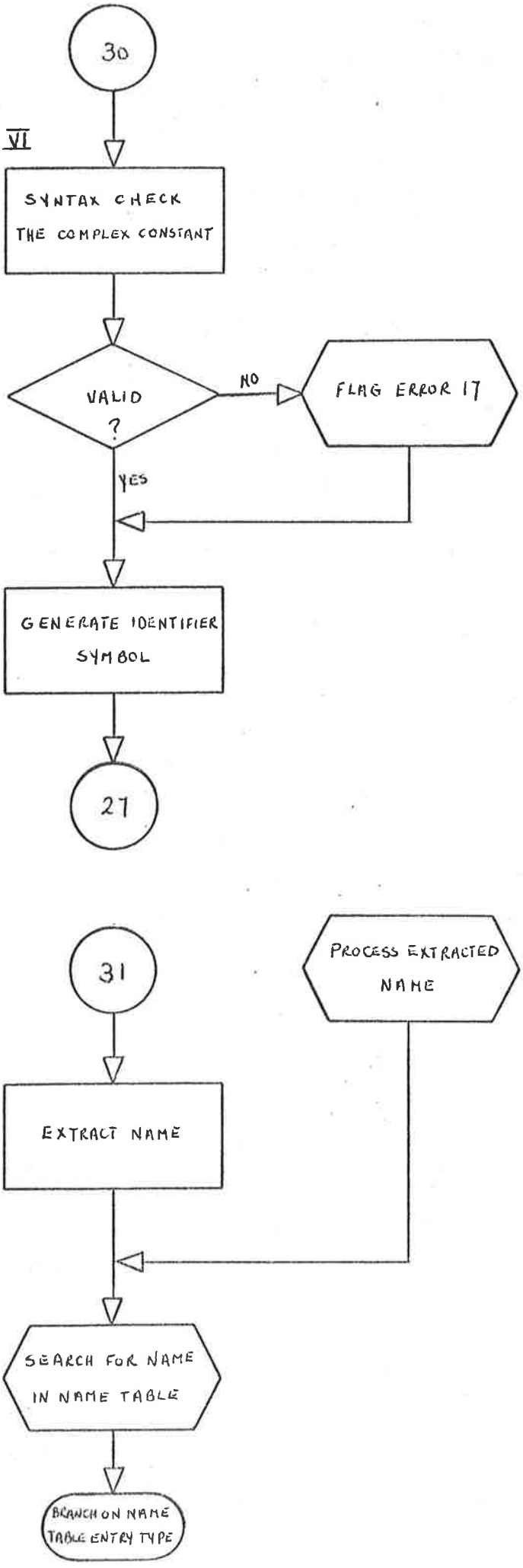

22
EXTRACT NEXT SYMBOL
PROCESS EXPRESSION
1

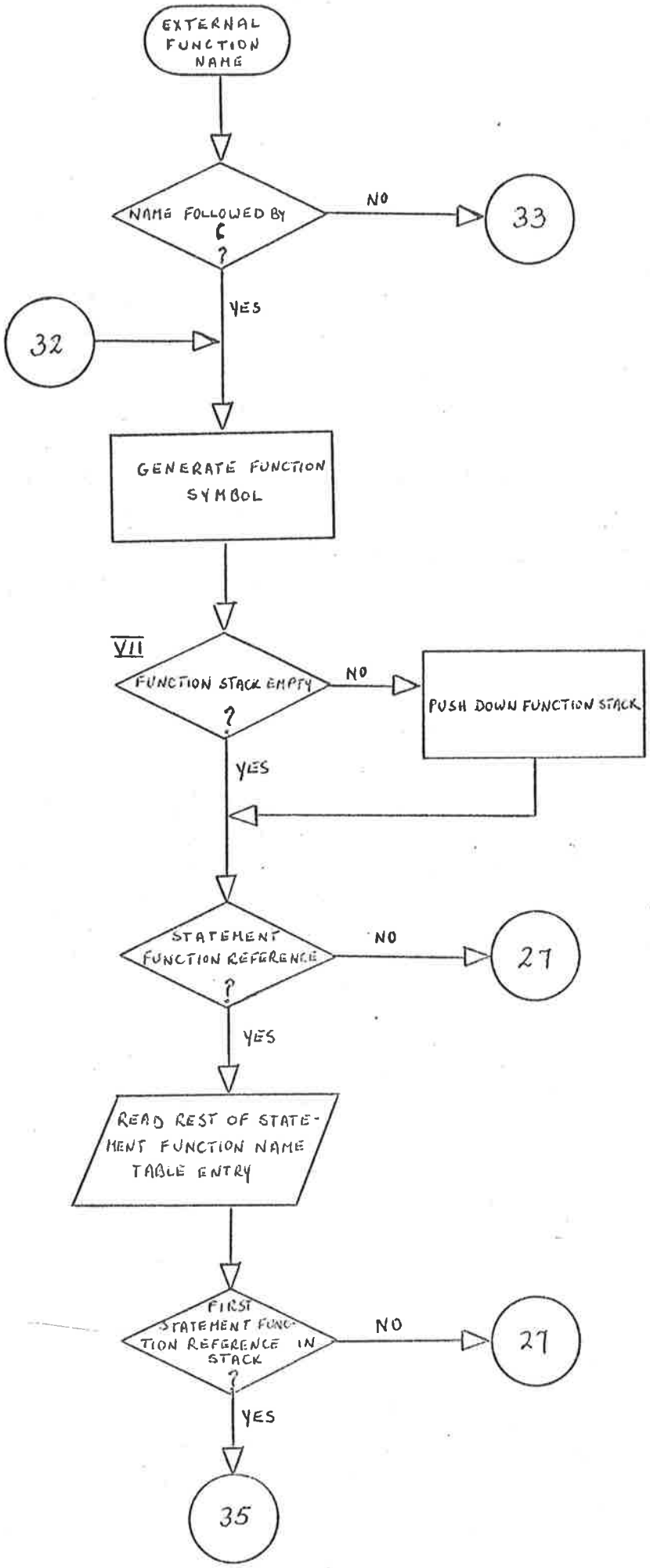

EXTRACT NEXT SYMBOL
END OF STRING BODY ALREADY FOUND ?
YES
11
NO
IS NEXT CHARACTER ALPHABETIC ?
YES
31
NO
III
IS NEXT CHARACTER NUMERIC ?
YES
SYNTAX CHECK NUMBER
NO
IS NEXT CHARACTER NON-ALPHANUMERIC ?
NO
34
YES
23
VALID ?
YES
27
NO
FLAG ERROR 17
27

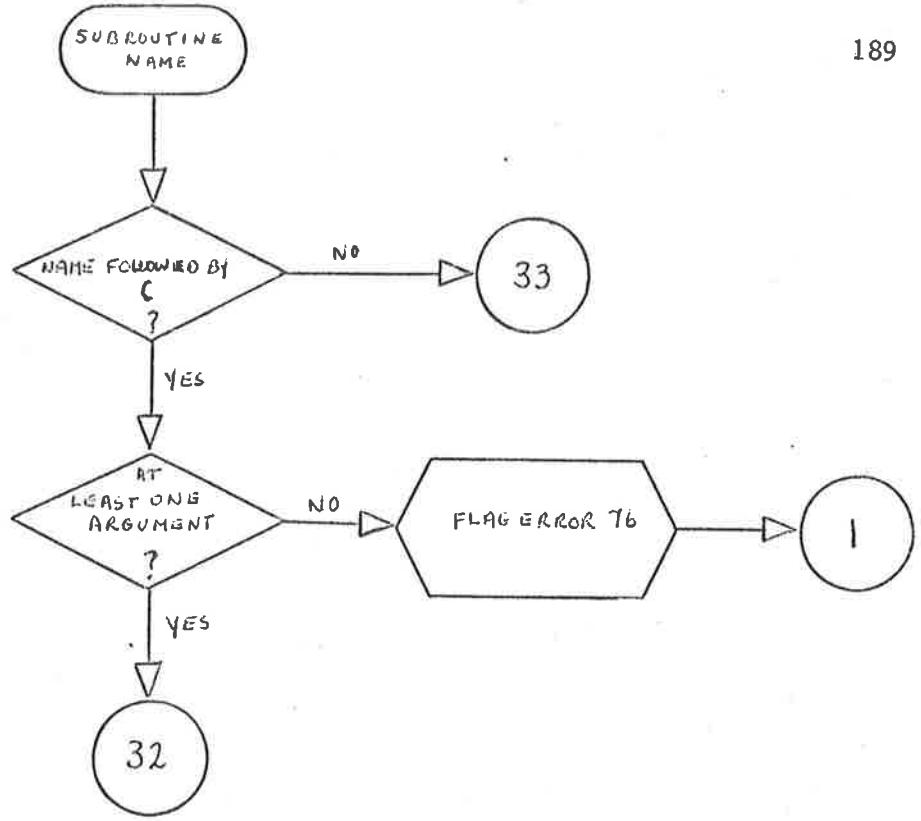

23
IS NEXT CHARACTER (?
NO
IS IT • ?
YES
YES
26
II
IS IT START OF AN ARGUMENT LIST ?
NO
28
NO
YES
24
INSIDE A FUNCTION REFERENCE ?
YES
25
NO
INSIDE A SUBROUTINE ARGUMENT ?
NO
SET SYMBOL TO CURRENT CHARACTER
EXIT
YES
IS NEXT CHARACTER) ?
YES
NO
IS IT ' ?
YES
GENERATE "END OF STRING" SYMBOL
EXIT
NO
25

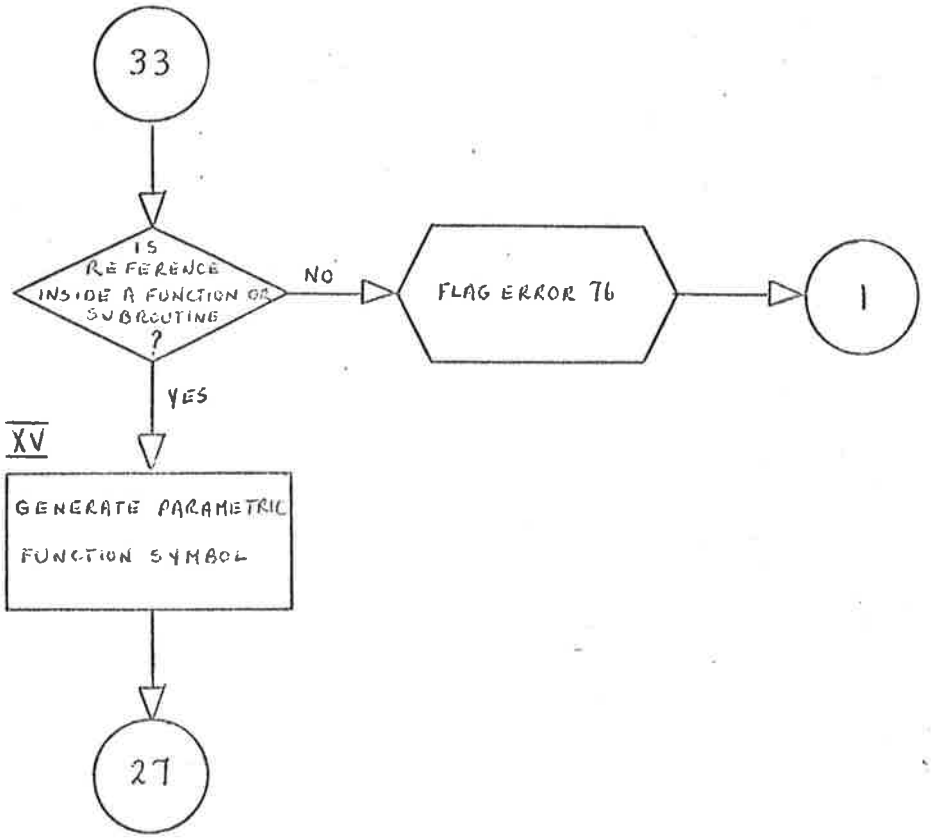

26
IS NEXT CHARACTER NUMERIC ?
NO
FLAG ERROR 17
1
YES
III
SYNTAX CHECK NUMBER
27
IV
IS FPL FACILITY ON ?
YES
STORE TYPE OF CURRENT ITEM AT TOP OF THE TYPE STACK
NO
EXIT

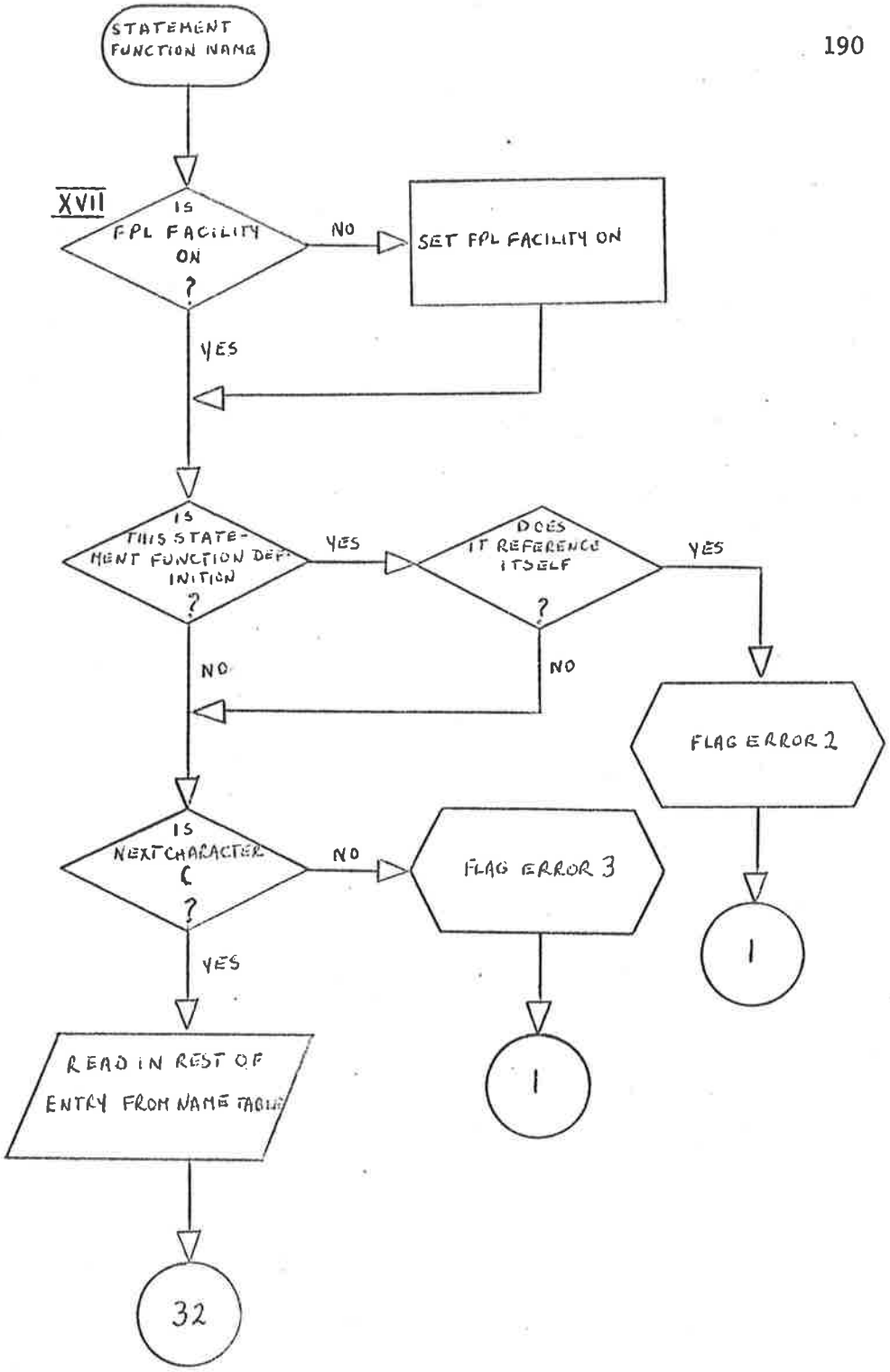

28
V
SAVE POINTERS TO STRING BODY
IS NEXT CHARACTER NUMERIC ?
YES
29
NO
IS IT MINUS ?
YES
NEGATIVE REAL PART ?
YES
29
NO
NO
IS IT . ?
NO
UNSAVE SAVED POINTERS TO STRING BODY
24
YES
FRACTIONAL REAL PART ?
NO
YES
29
SCAN FOR COMMA BETWEEN REAL AND IMAGINARY PARTS
FOUND ?
NO
YES
30

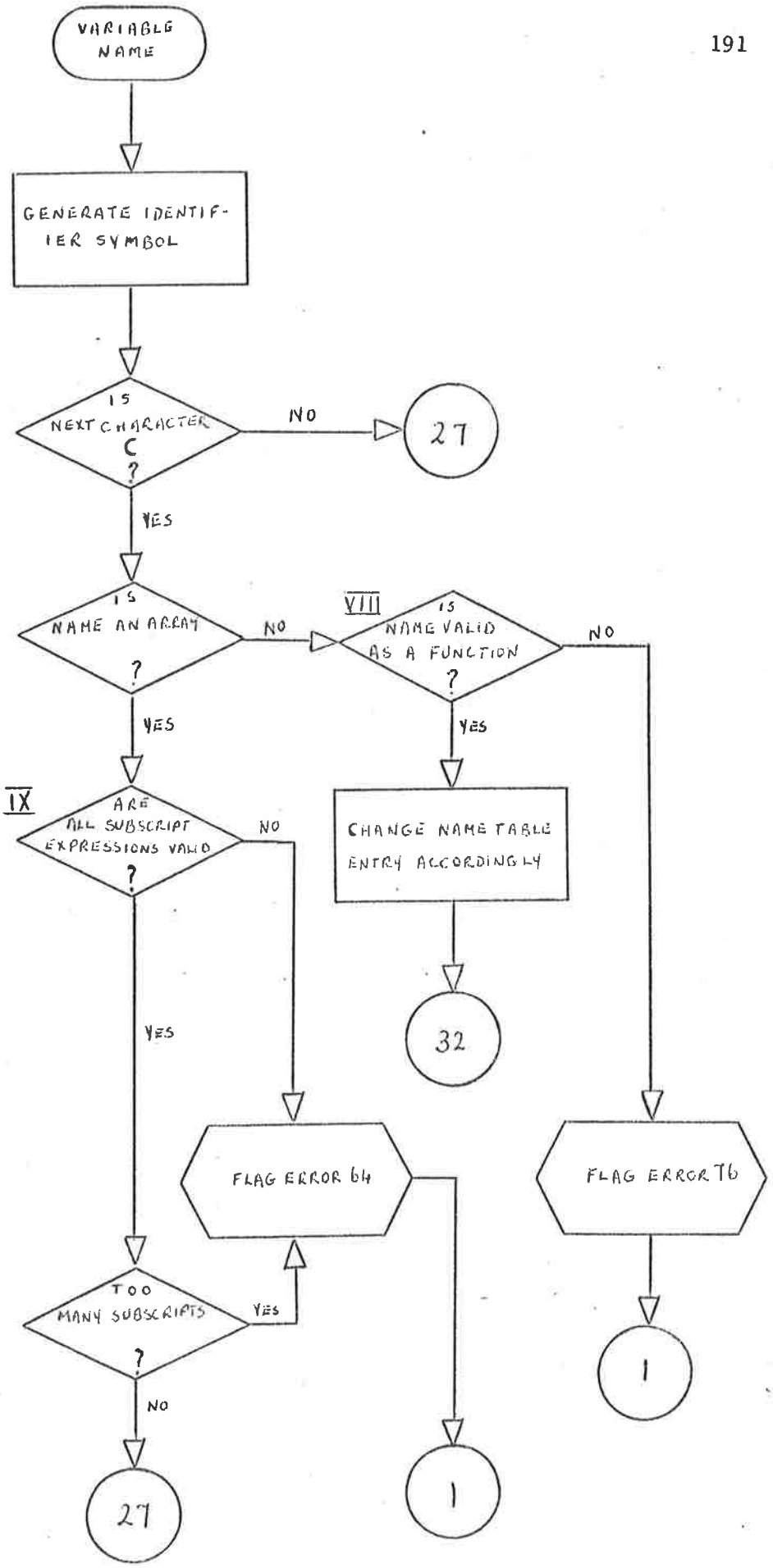

30
VI
SYNTAX CHECK THE COMPLEX CONSTANT
VALID ?
NO
FLAG ERROR 17
YES
GENERATE IDENTIFIER SYMBOL
27
31
PROCESS EXTRACTED NAME
EXTRACT NAME
SEARCH FOR NAME IN NAME TABLE
BRANCH ON NAME TABLE ENTRY TYPE

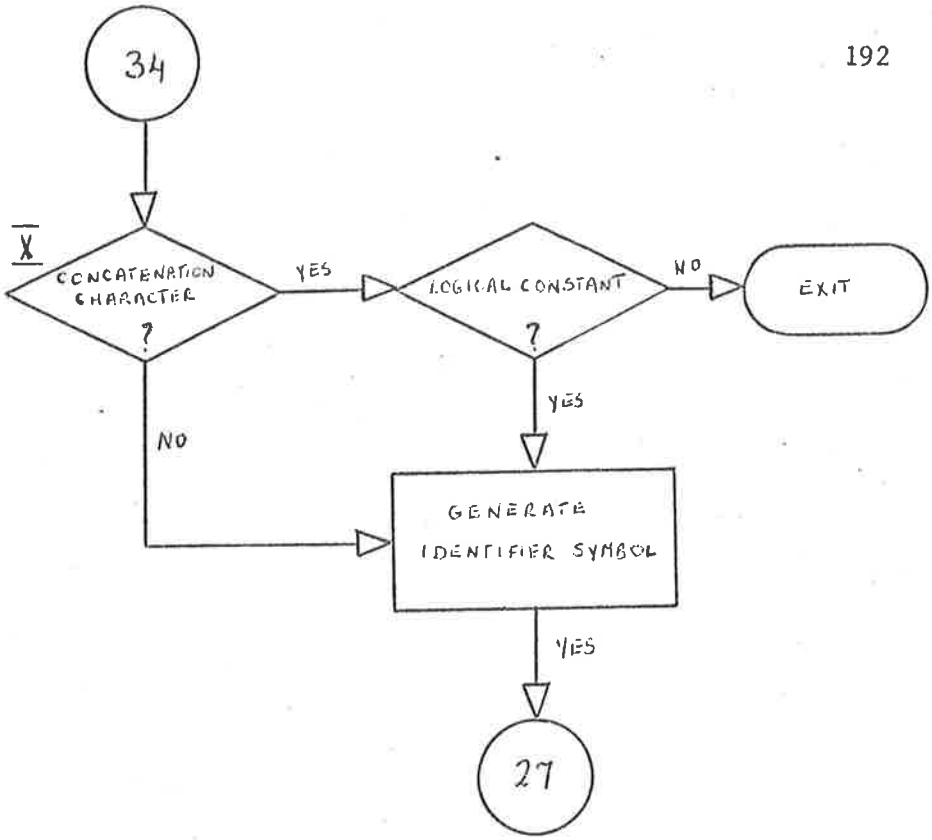

EXTERNAL FUNCTION NAME
NAME FOLLOWED BY (?
NO
33
YES
32
GENERATE FUNCTION SYMBOL
VII
FUNCTION STACK EMPTY ?
NO
PUSH DOWN FUNCTION STACK
YES
STATEMENT FUNCTION REFERENCE ?
NO
27
YES
READ REST OF STATEMENT FUNCTION NAME TABLE ENTRY
FIRST STATEMENT FUNCTION REFERENCE IN STACK ?
NO
27
YES
35

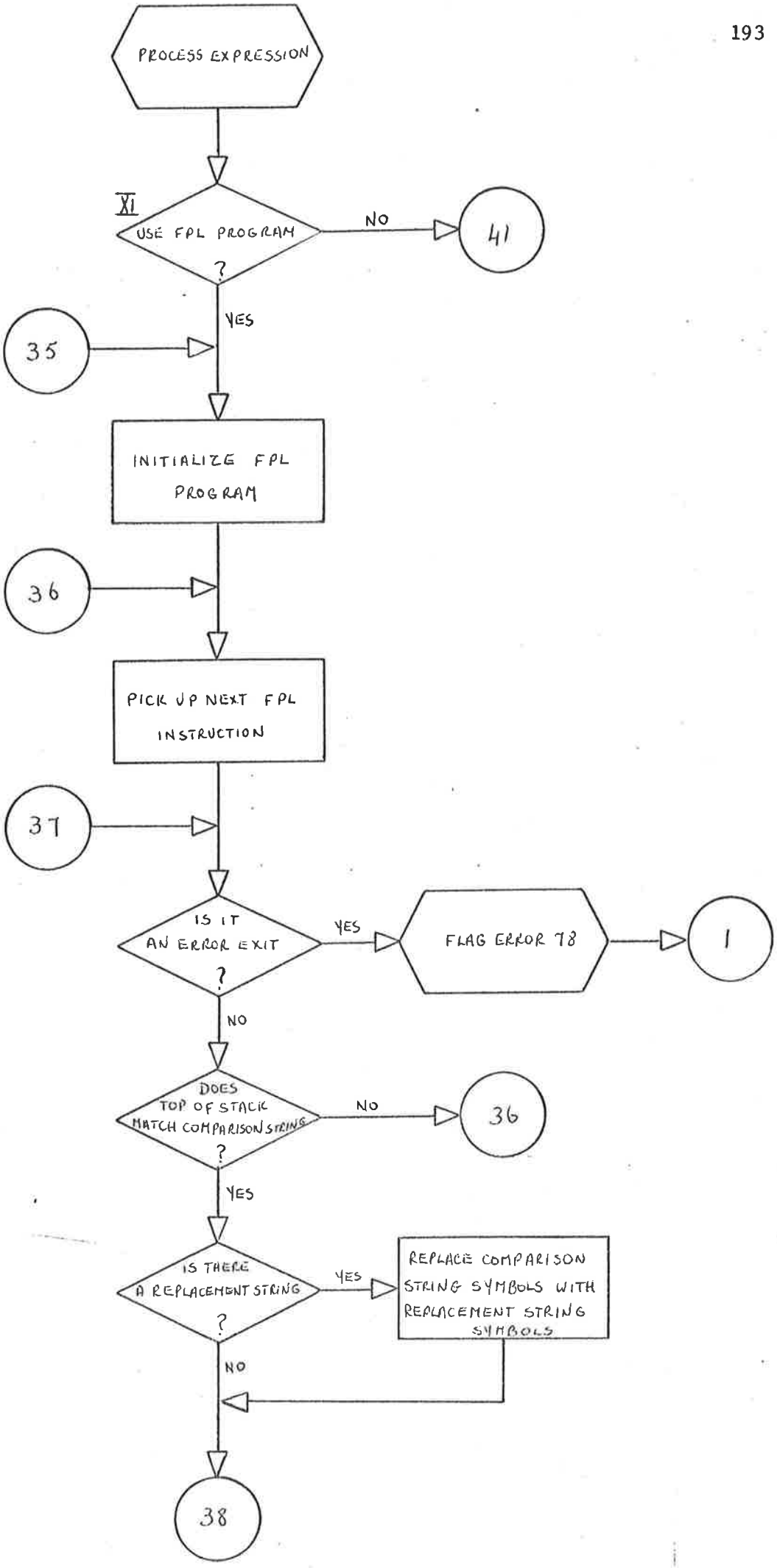

SUBROUTINE NAME
NAME FOLLOWED BY C ?
NO
33
YES
AT LEAST ONE ARGUMENT ?
NO
FLAG ERROR 76
1
YES
32

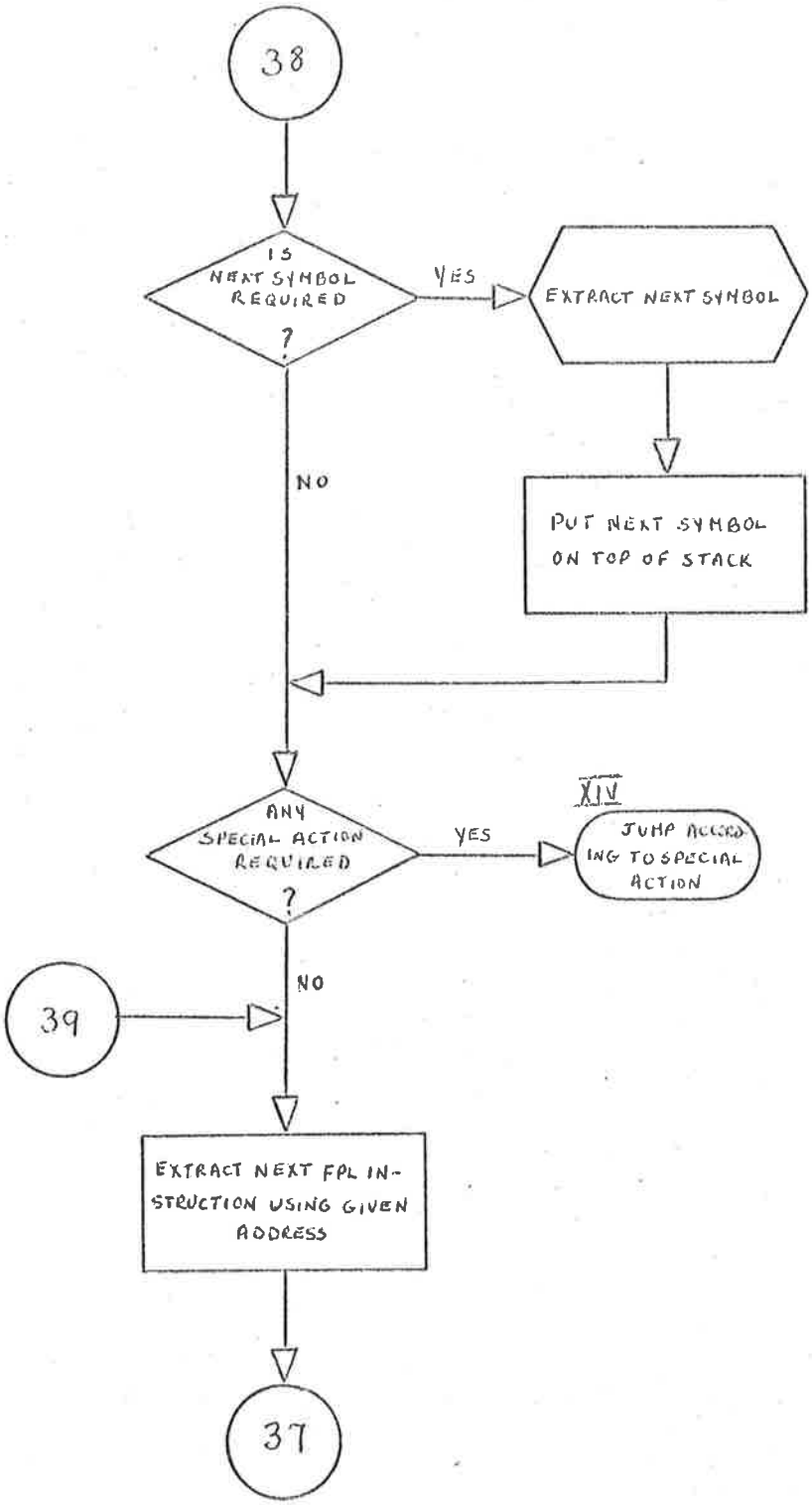

33
IS REFERENCE INSIDE A FUNCTION OR SUBROUTINE ?
NO
FLAG ERROR 76
1
YES
XV
GENERATE PARAMETRIC FUNCTION SYMBOL
27

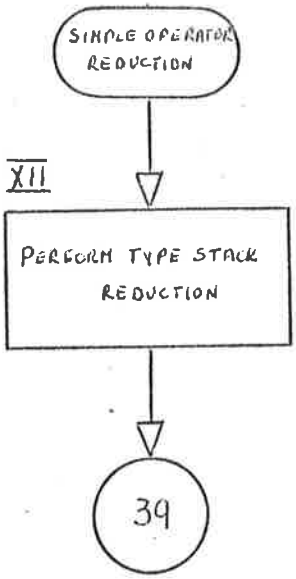

STATEMENT FUNCTION NAME
XVII
IS FPL FACILITY ON ?
NO
SET FPL FACILITY ON
YES
IS THIS STATEMENT FUNCTION DEFINITION ?
YES
DOES IT REFERENCE ITSELF ?
YES
NO
NO
FLAG ERROR 2
IS NEXT CHARACTER (?
NO
FLAG ERROR 3
YES
1
1
READ IN REST OF ENTRY FROM NAME TABLE
32

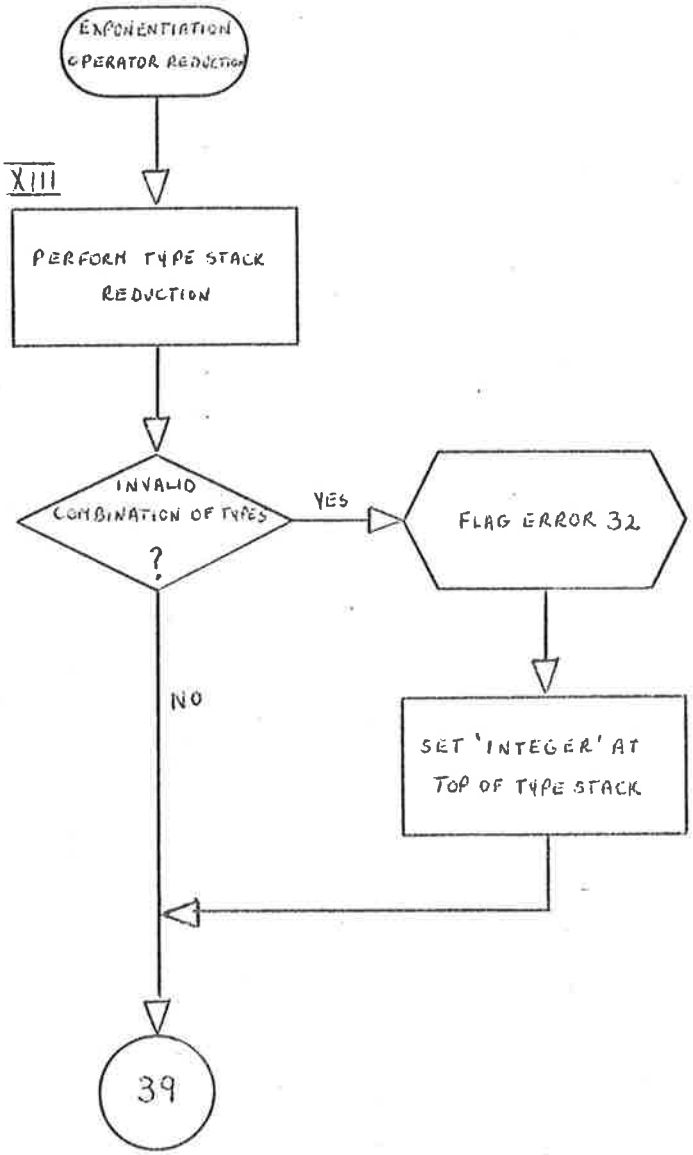

VARIABLE NAME
GENERATE IDENTIFIER SYMBOL
IS NEXT CHARACTER C ?
NO
27
YES
IS NAME AN ARRAY ?
NO
VIII
IS NAME VALID AS A FUNCTION ?
NO
YES
YES
IX
ARE ALL SUBSCRIPT EXPRESSIONS VALID ?
NO
CHANGE NAME TABLE ENTRY ACCORDINGLY
32
YES
FLAG ERROR 64
FLAG ERROR 76
TOO MANY SUBSCRIPTS ?
YES
NO
1
1
27

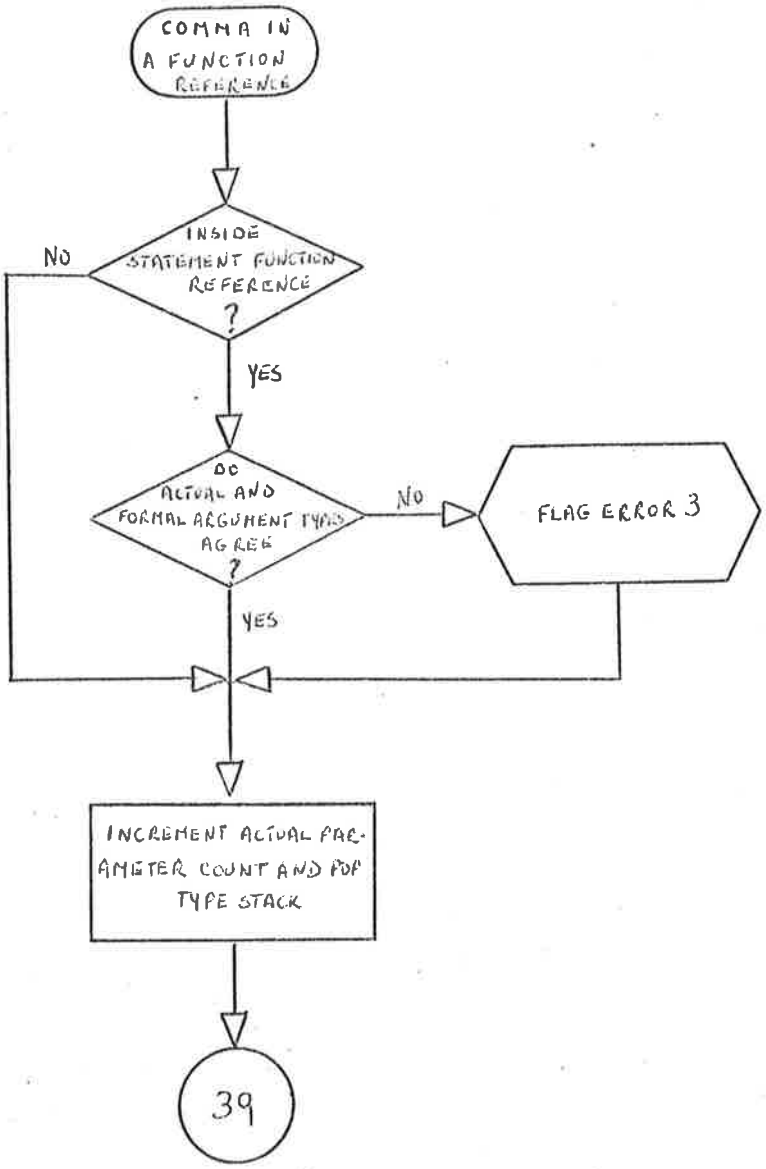

34
X
CONCATENATION CHARACTER ?
YES
LOGICAL CONSTANT ?
NO
EXIT
YES
NO
GENERATE IDENTIFIER SYMBOL
YES
27

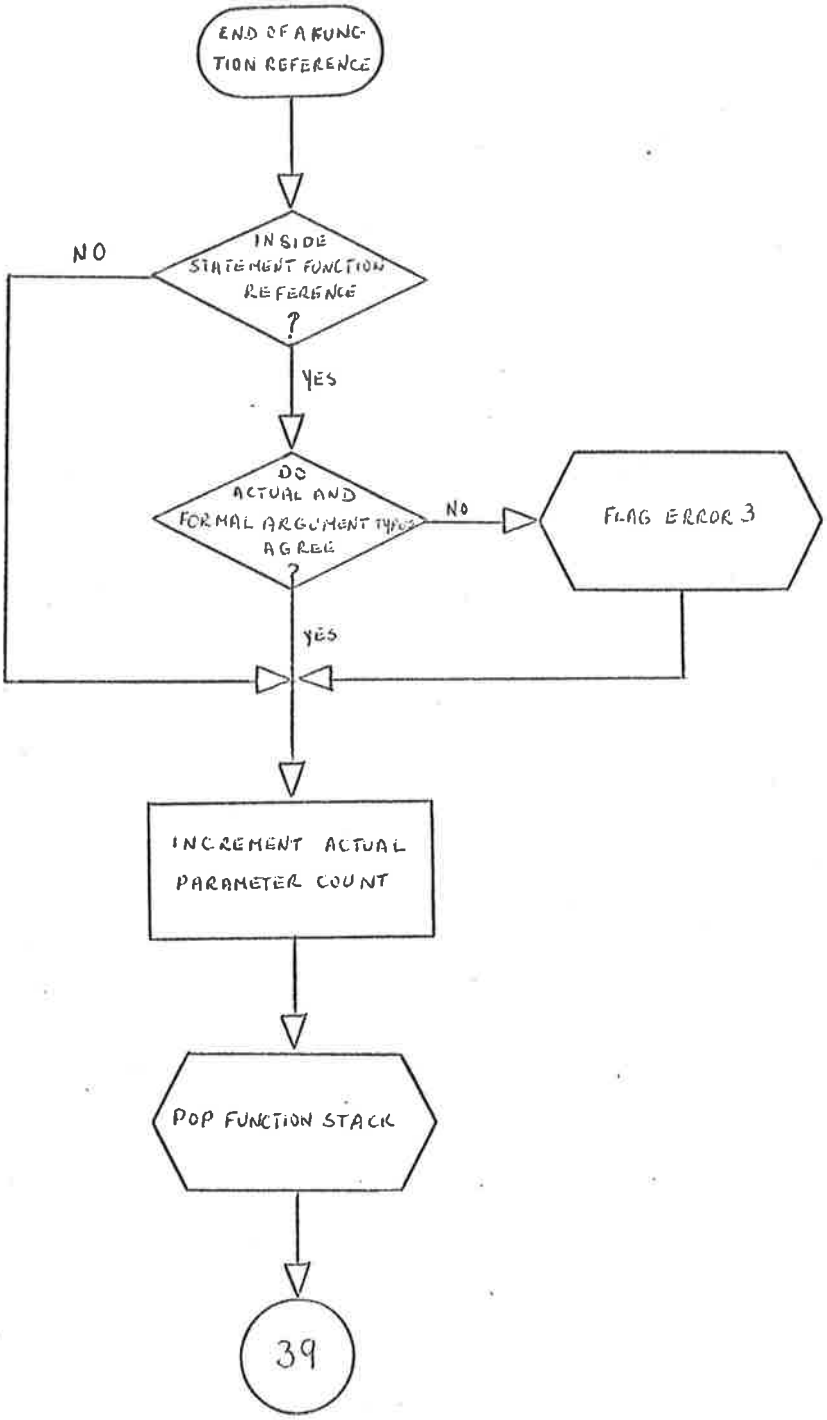

PROCESS EXPRESSION
XI
USE FPL PROGRAM ?
NO
41
YES
35
INITIALIZE FPL PROGRAM
36
PICK UP NEXT FPL INSTRUCTION
37
IS IT AN ERROR EXIT ?
YES
FLAG ERROR 78
1
NO
DOES TOP OF STACK MATCH COMPARISON STRING ?
NO
36
YES
IS THERE A REPLACEMENT STRING ?
YES
REPLACE COMPARISON STRING SYMBOLS WITH REPLACEMENT STRING SYMBOLS
NO
38

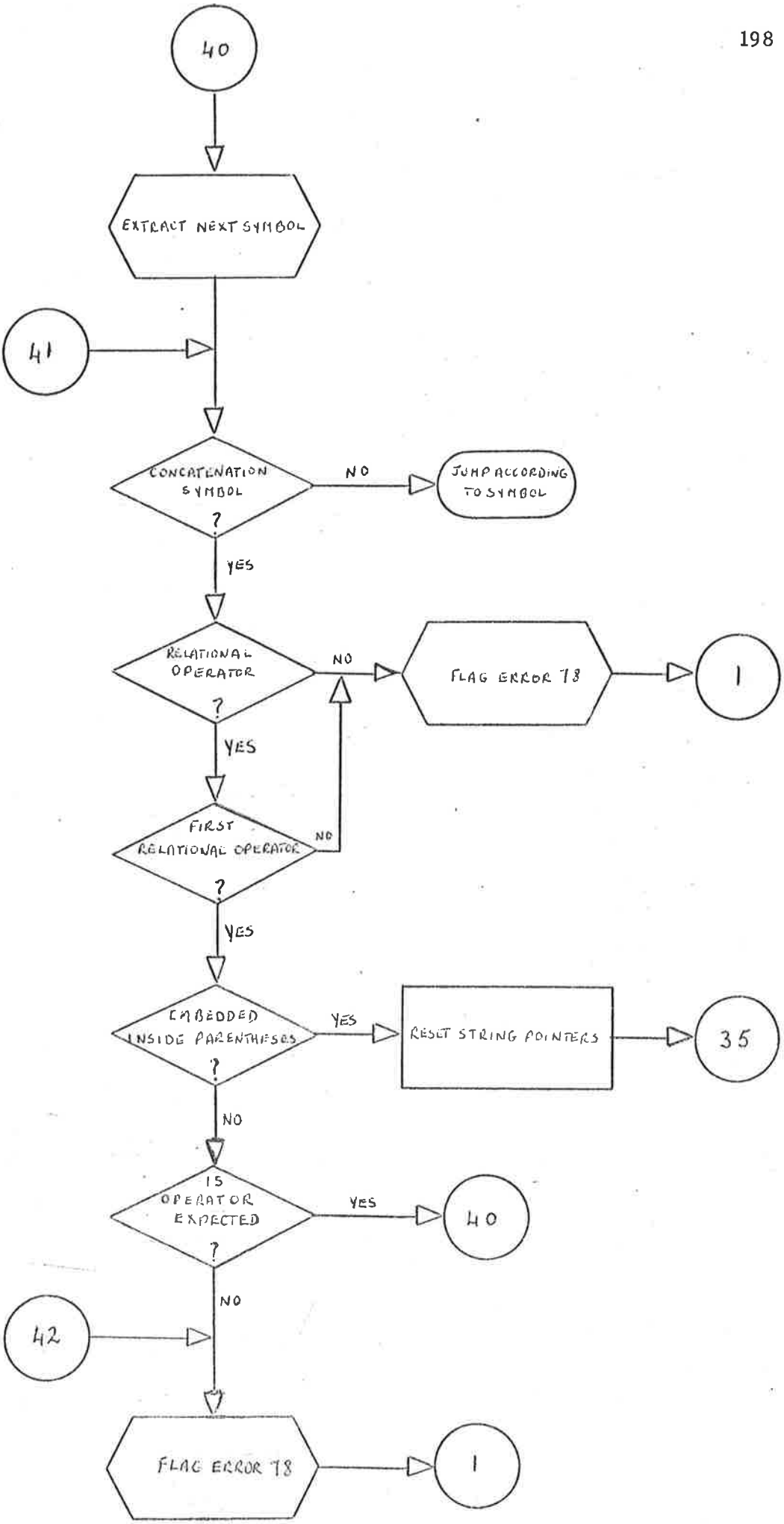

38
IS NEXT SYMBOL REQUIRED ?
YES
EXTRACT NEXT SYMBOL
NO
PUT NEXT SYMBOL ON TOP OF STACK
ANY SPECIAL ACTION REQUIRED ?
YES
XIV
JUMP ACCORDING TO SPECIAL ACTION
NO
39
EXTRACT NEXT FPL INSTRUCTION USING GIVEN ADDRESS
37

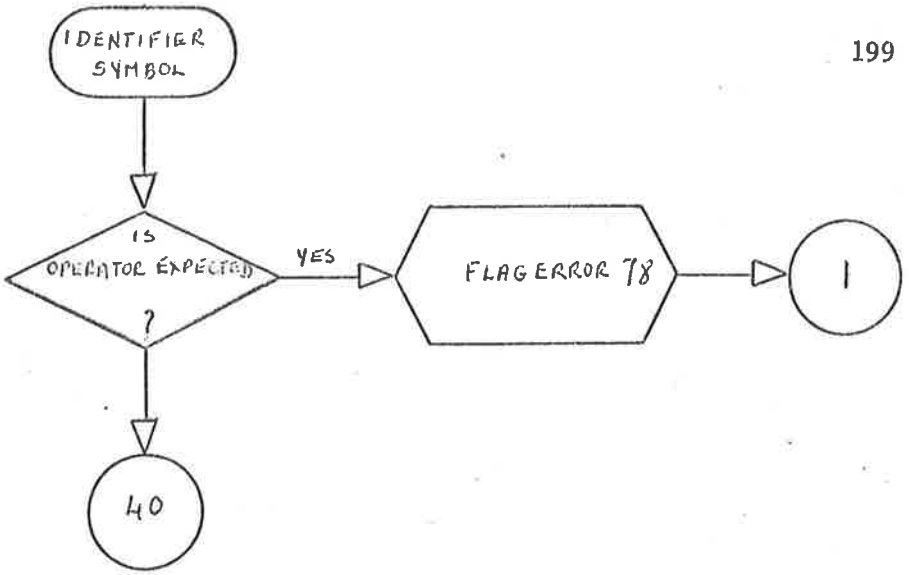

SIMPLE OPERATOR REDUCTION
XII
PERFORM TYPE STACK REDUCTION
39

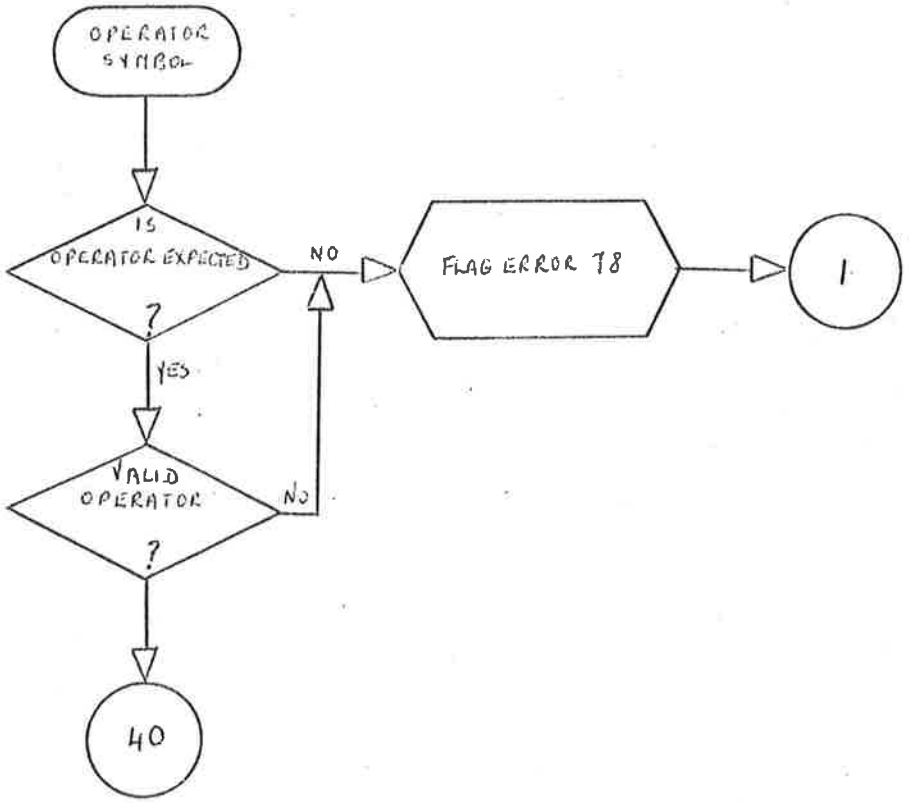

EXPONENTIATION OPERATOR REDUCTION
XIII
PERFORM TYPE STACK REDUCTION
INVALID COMBINATION OF TYPES ?
YES
FLAG ERROR 32
NO
SET 'INTEGER' AT TOP OF TYPE STACK
39

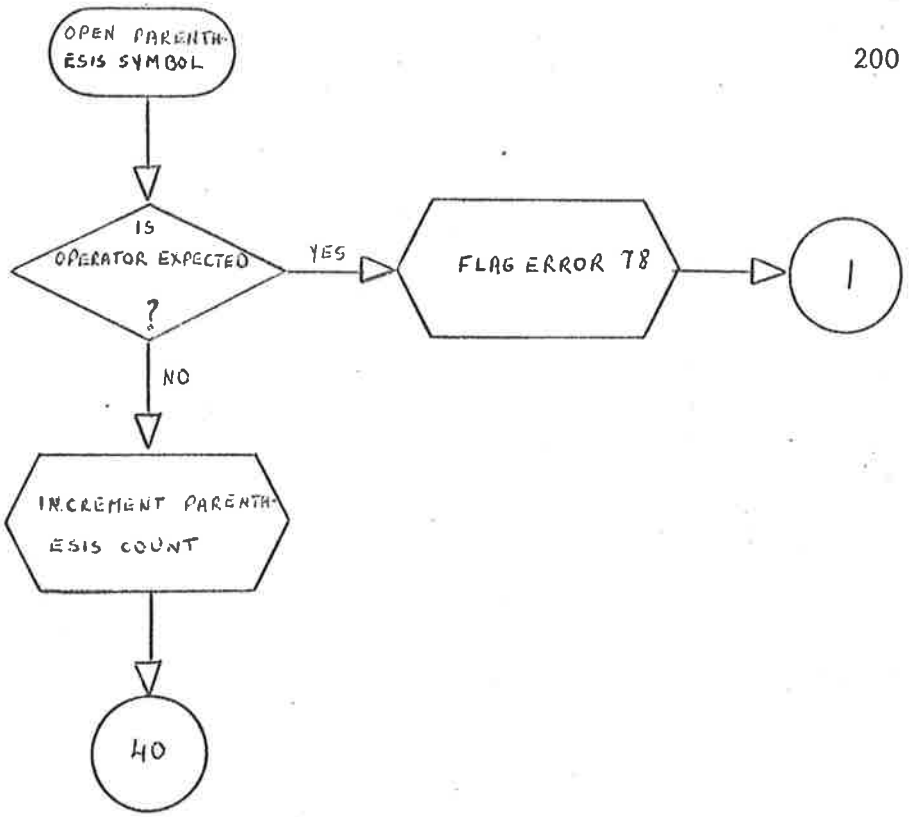

COMMA IN A FUNCTION REFERENCE
INSIDE STATEMENT FUNCTION REFERENCE ?
NO
YES
DO ACTUAL AND FORMAL ARGUMENT TYPES AGREE ?
NO
FLAG ERROR 3
YES
INCREMENT ACTUAL PARAMETER COUNT AND POP TYPE STACK
39

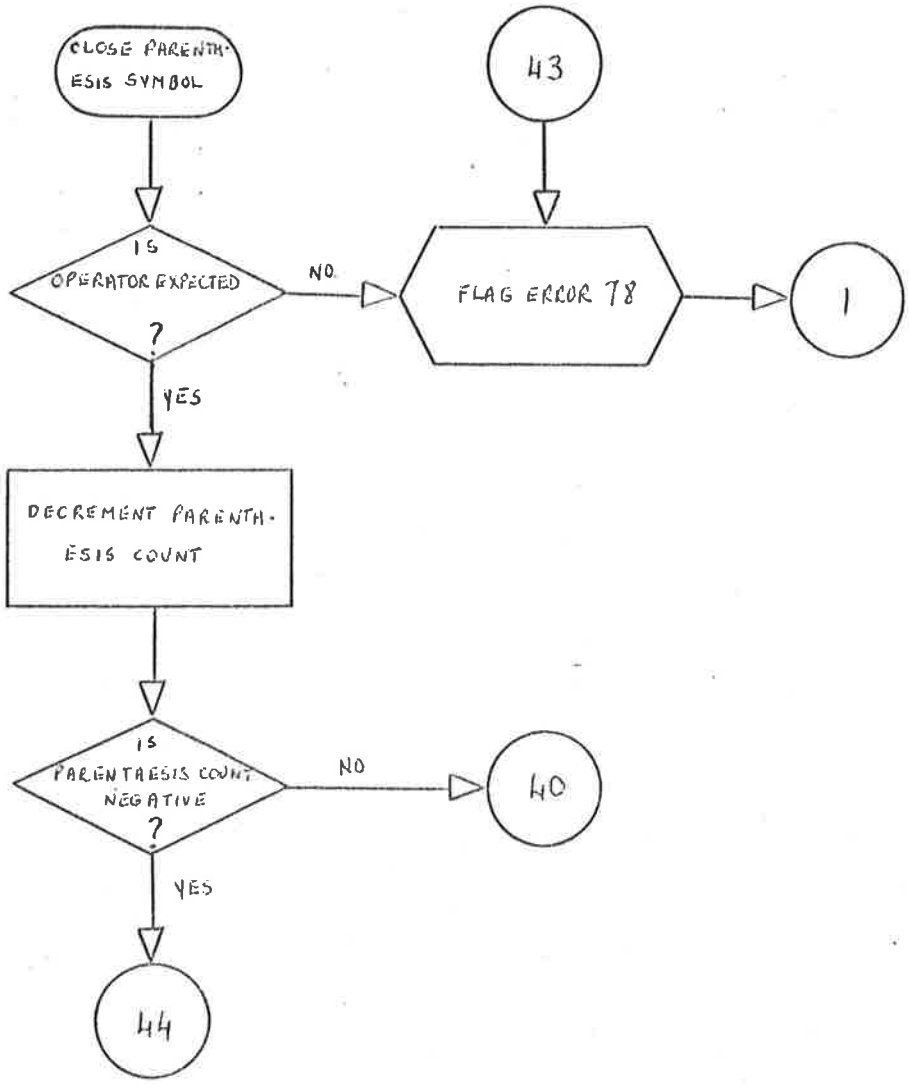

END OF A FUNC-
TION REFERENCE
INSIDE
STATEMENT FUNCTION
REFERENCE
?
NO
YES
DO
ACTUAL AND
FORMAL ARGUMENT TYPES
AGREE
?
NO
FLAG ERROR 3
YES
INCREMENT ACTUAL
PARAMETER COUNT
POP FUNCTION STACK
39

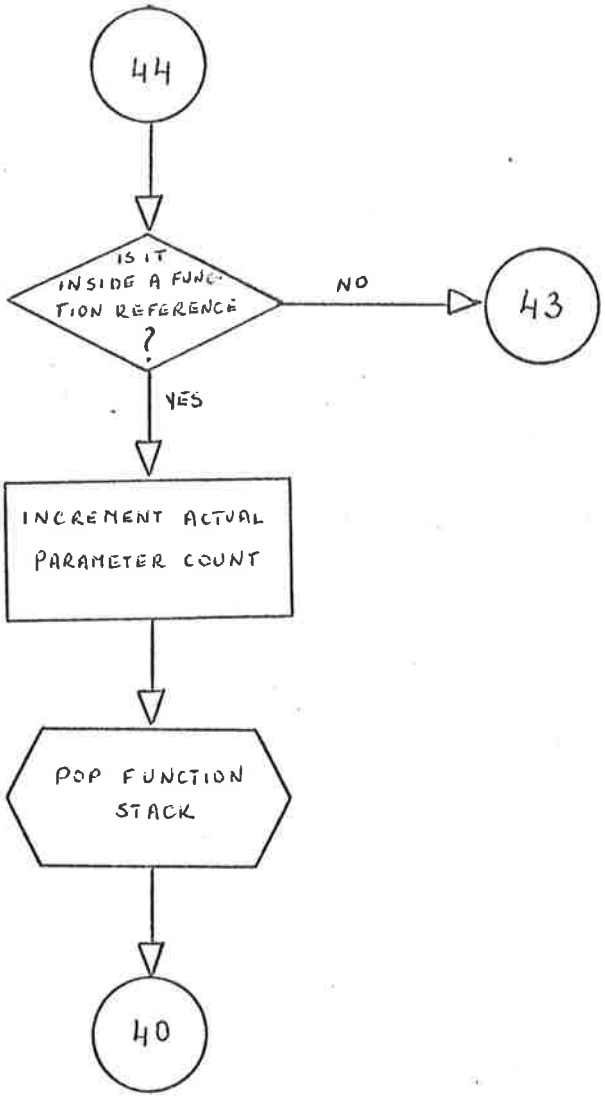

40
EXTRACT NEXT SYMBOL
41
CONCATENATION SYMBOL ?
NO
JUMP ACCORDING TO SYMBOL
YES
RELATIONAL OPERATOR ?
NO
FLAG ERROR 78
1
YES
FIRST RELATIONAL OPERATOR ?
NO
YES
EMBEDDED INSIDE PARENTHESES ?
YES
RESET STRING POINTERS
35
NO
IS OPERATOR EXPECTED ?
YES
40
NO
42
FLAG ERROR 78
1

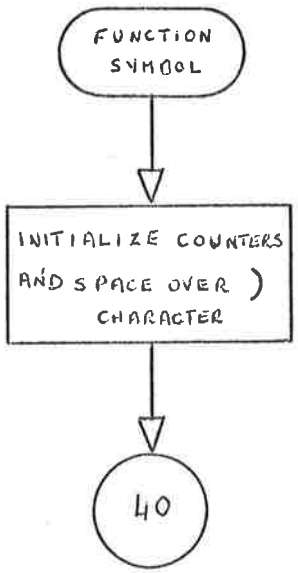

IDENTIFIER SYMBOL
IS OPERATOR EXPECTED ?
YES
FLAG ERROR 78
1
40

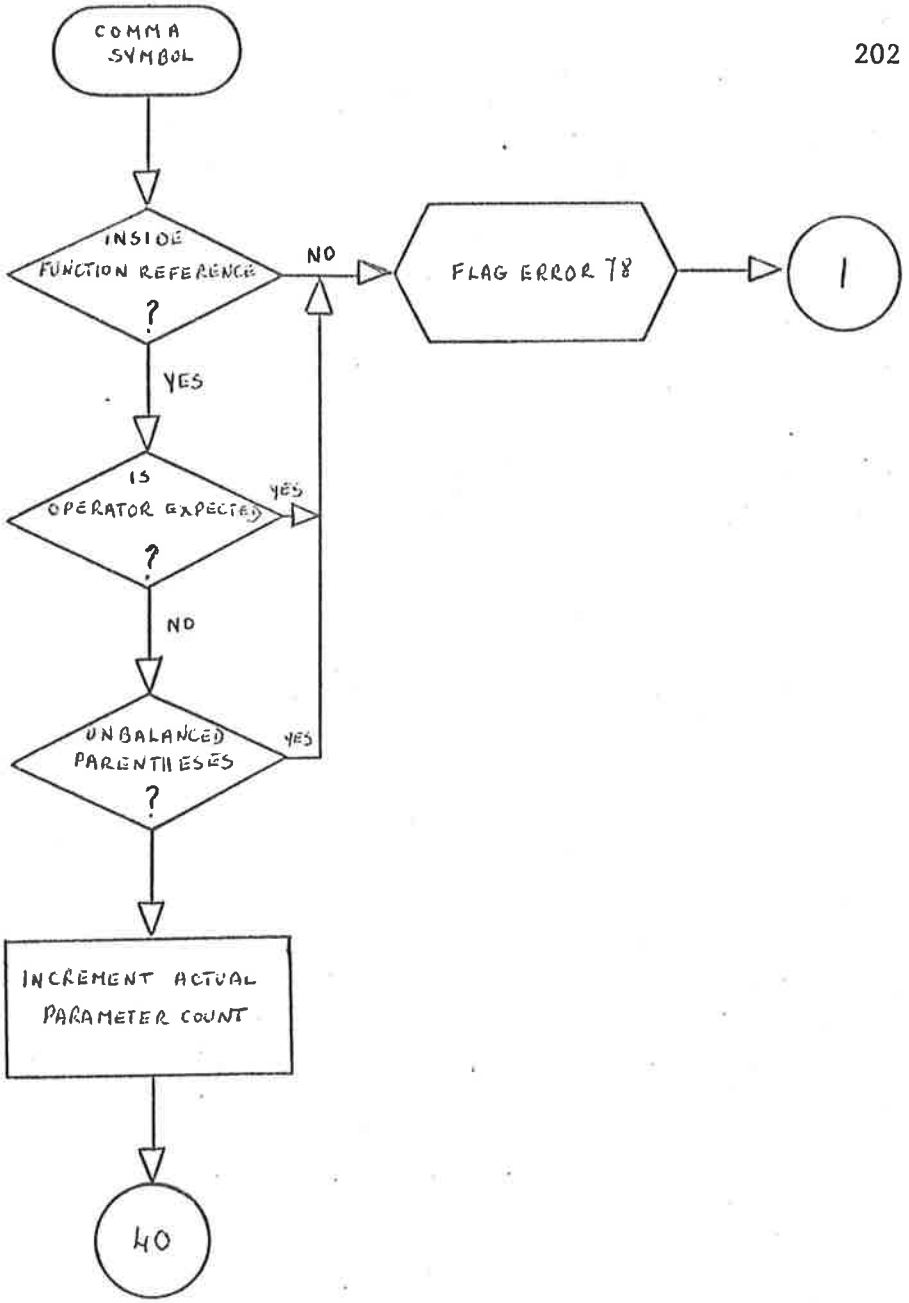

OPERATOR SYMBOL
IS OPERATOR EXPECTED ?
NO
FLAG ERROR 78
1
YES
VALID OPERATOR ?
NO
40

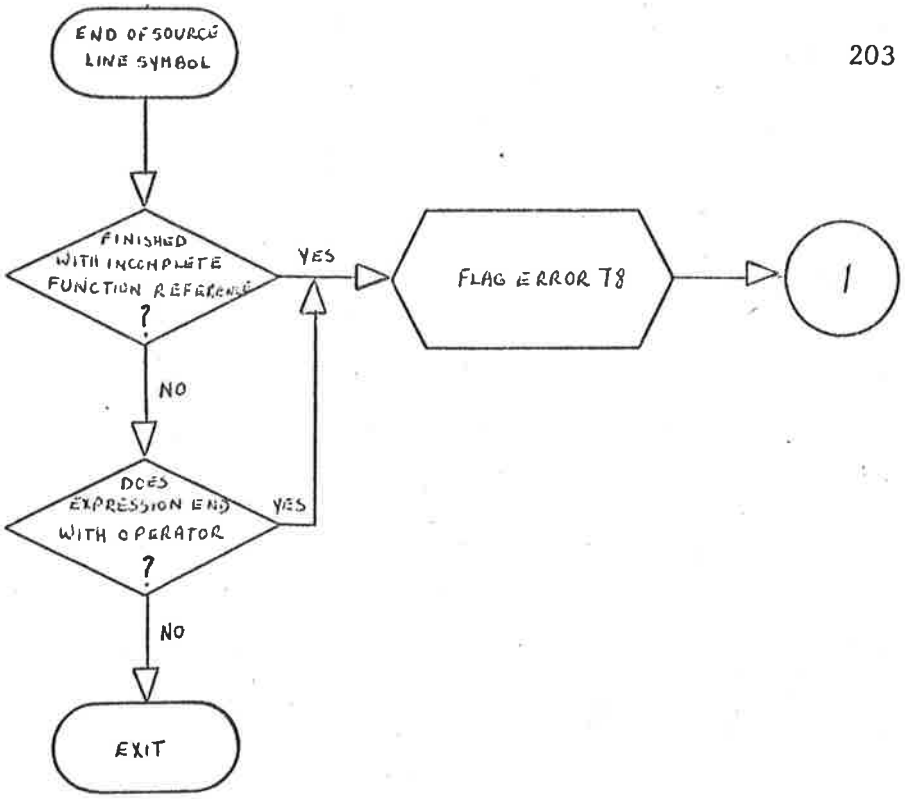

OPEN PARENTH-
ESIS SYMBOL
IS
OPERATOR EXPECTED
?
YES
FLAG ERROR 78
1
NO
INCREMENT PARENTH-
ESIS COUNT
40

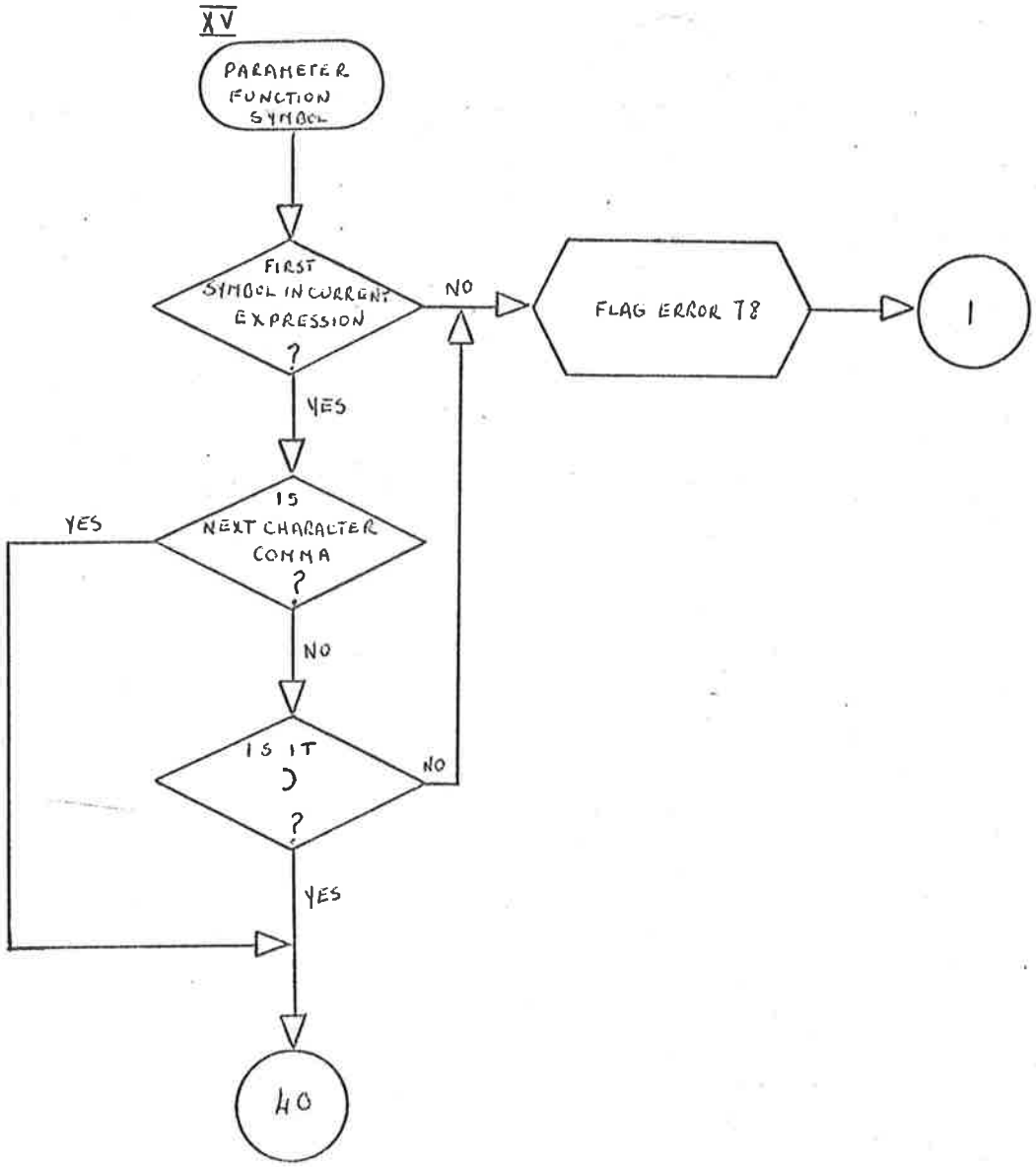

CLOSE PARENTH-
ESIS SYMBOL
43
IS
OPERATOR EXPECTED
?
NO
FLAG ERROR 78
1
YES
DECREMENT PARENTH-
ESIS COUNT
IS
PARENTHESIS COUNT
NEGATIVE
?
NO
40
YES
44

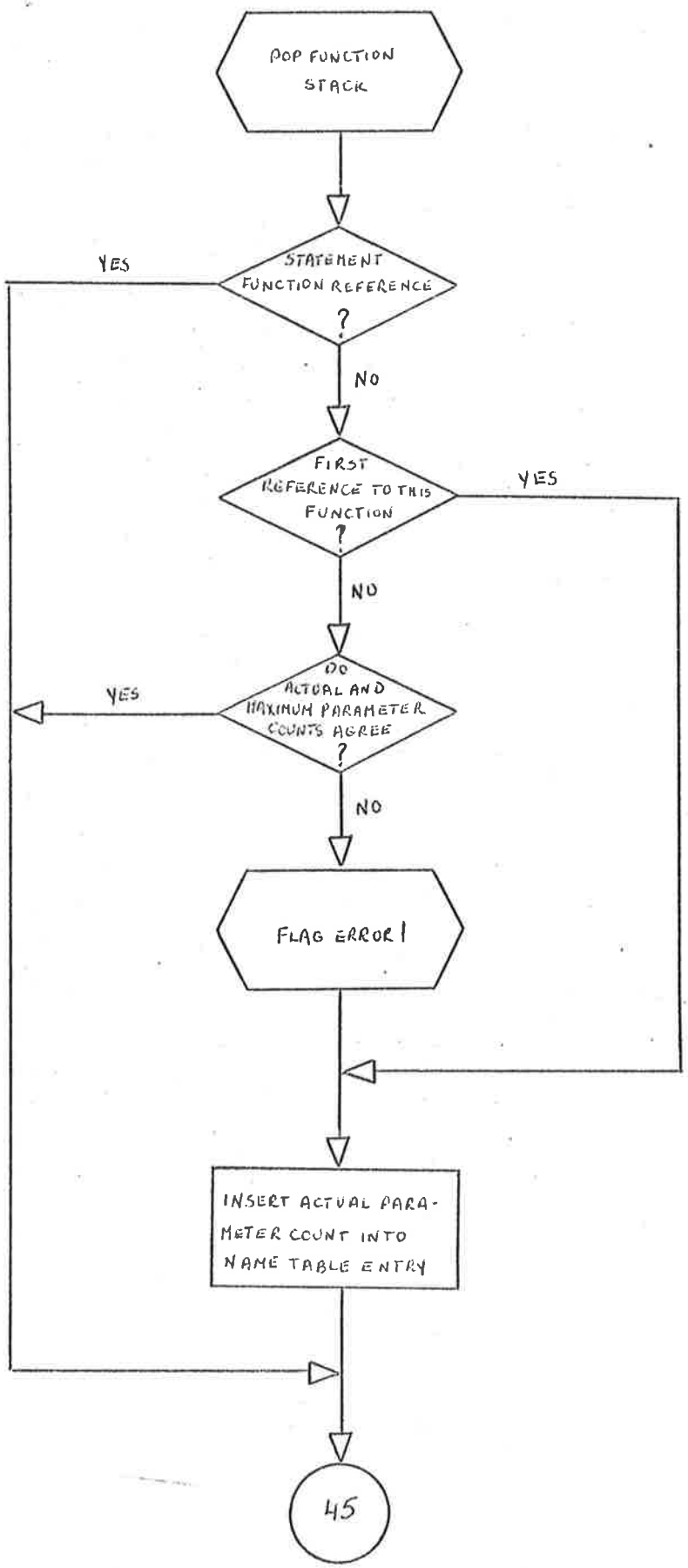

44
IS IT INSIDE A FUNCTION REFERENCE ?
NO
43
YES
INCREMENT ACTUAL PARAMETER COUNT
POP FUNCTION STACK
40

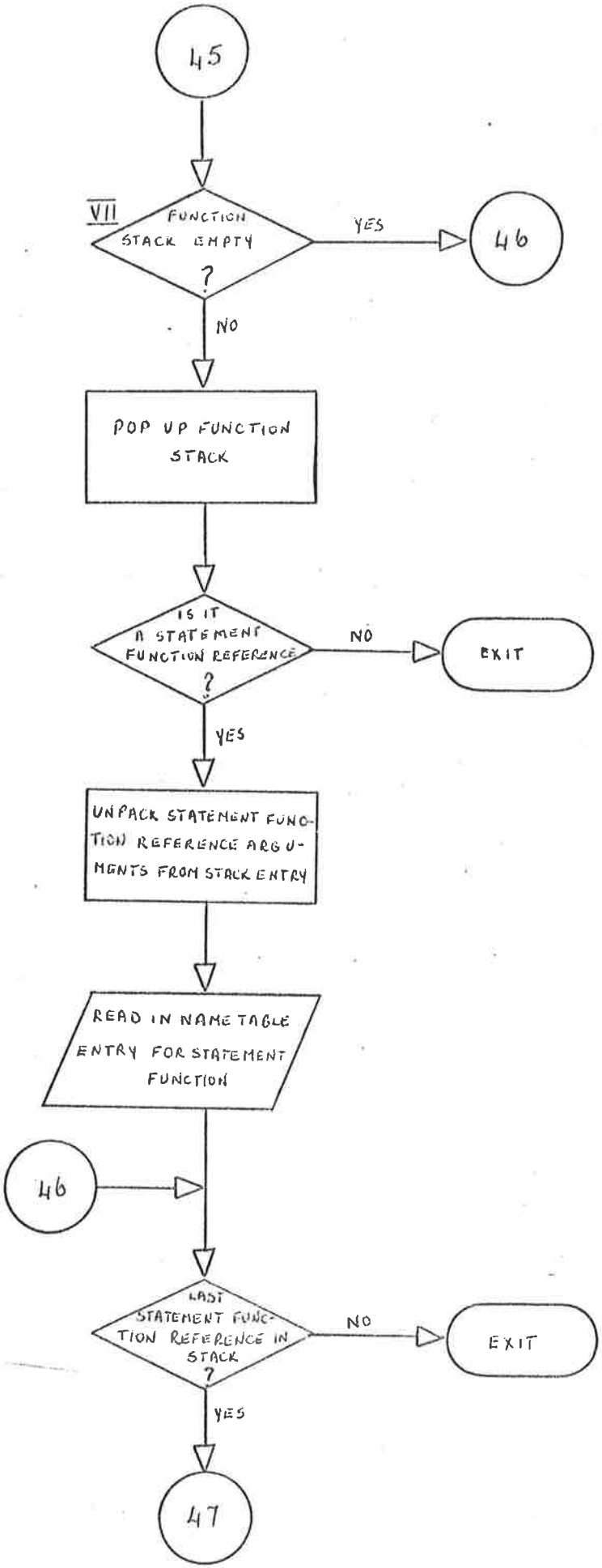

FUNCTION SYMBOL
INITIALIZE COUNTERS AND SPACE OVER) CHARACTER
40

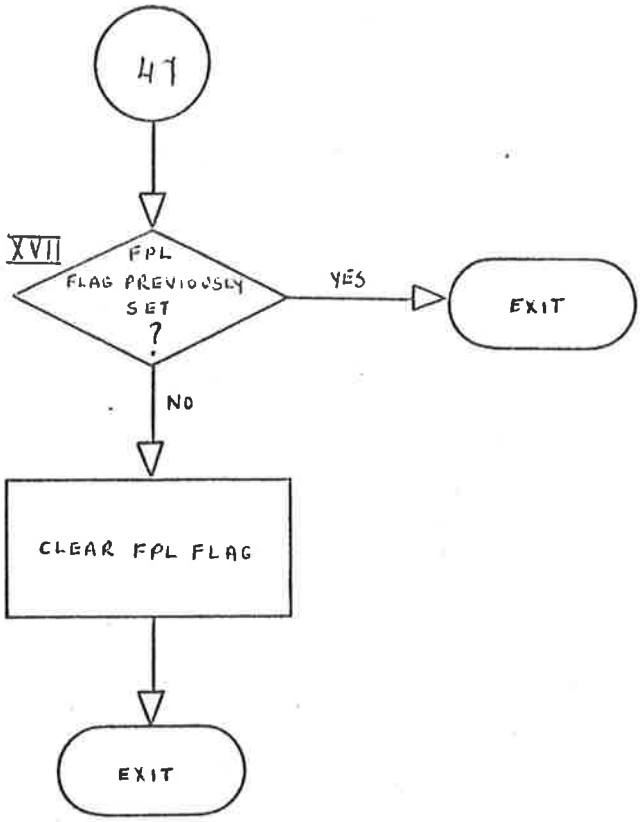

COMMA SYMBOL
INSIDE FUNCTION REFERENCE ?
NO
FLAG ERROR 78
1
YES
IS OPERATOR EXPECTED ?
YES
NO
UNBALANCED PARENTHESES ?
YES
INCREMENT ACTUAL PARAMETER COUNT
40

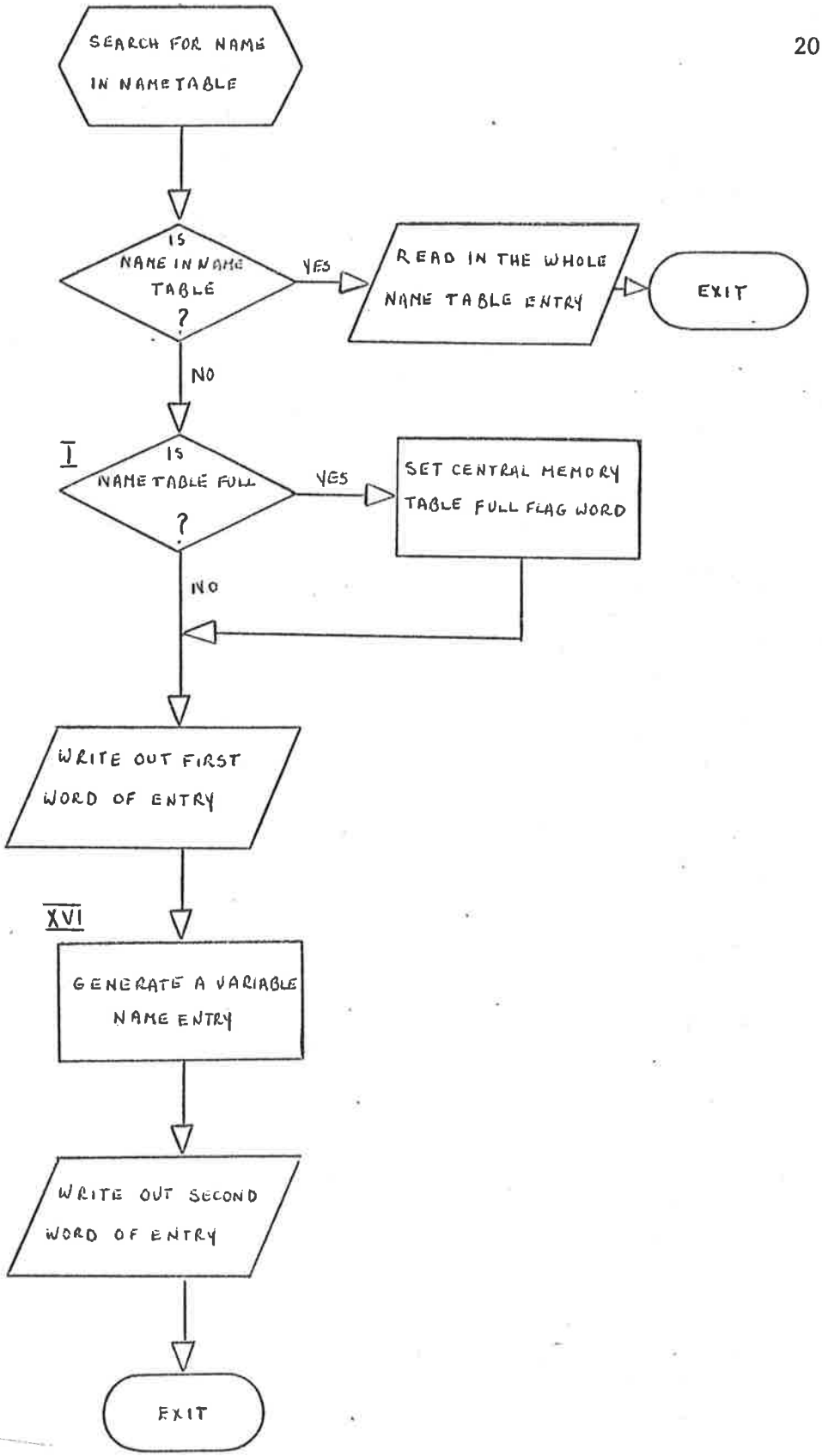

END OF SOURCE LINE SYMBOL
FINISHED WITH INCOMPLETE FUNCTION REFERENCE ?
YES
FLAG ERROR T8
1
NO
DOES EXPRESSION END WITH OPERATOR ?
YES
NO
EXIT

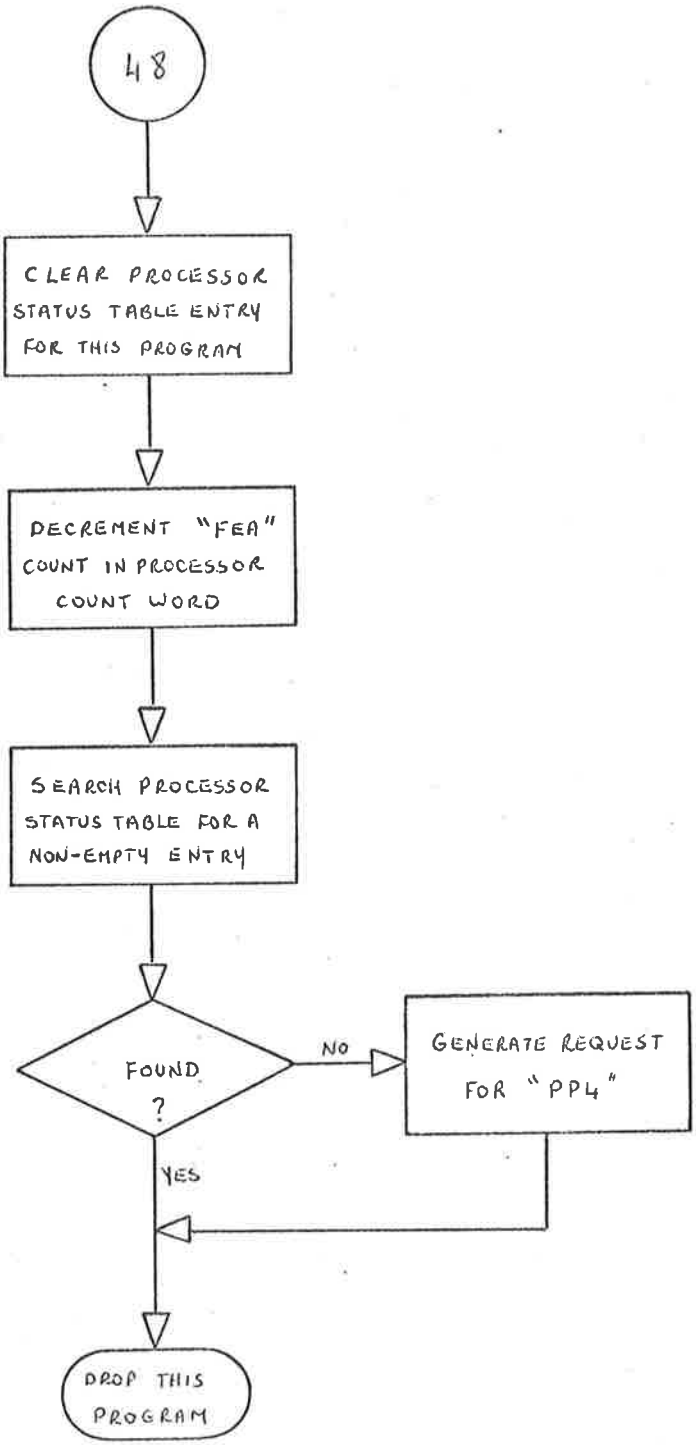

XV
PARAMETER FUNCTION SYMBOL
FIRST SYMBOL IN CURRENT EXPRESSION ?
NO
FLAG ERROR T8
1
YES
IS NEXT CHARACTER COMMA ?
YES
NO
IS IT) ?
NO
YES
40


POP FUNCTION STACK
STATEMENT FUNCTION REFERENCE ?
YES
NO
FIRST REFERENCE TO THIS FUNCTION ?
YES
NO
DO ACTUAL AND MAXIMUM PARAMETER COUNTS AGREE ?
YES
NO
FLAG ERROR!
INSERT ACTUAL PARAMETER COUNT INTO NAME TABLE ENTRY
45


45
VII
FUNCTION STACK EMPTY ?
YES
46
NO
POP UP FUNCTION STACK
IS IT A STATEMENT FUNCTION REFERENCE ?
NO
EXIT
YES
UNPACK STATEMENT FUNCTION REFERENCE ARGUMENTS FROM STACK ENTRY
READ IN NAME TABLE ENTRY FOR STATEMENT FUNCTION
46
LAST STATEMENT FUNCTION REFERENCE IN STACK ?
NO
EXIT
YES
47


47
XVII
FPL FLAG PREVIOUSLY SET ?
YES
EXIT
NO
CLEAR FPL FLAG
EXIT


SEARCH FOR NAME IN NAMETABLE
IS NAME IN NAME TABLE ?
YES
READ IN THE WHOLE NAME TABLE ENTRY
EXIT
NO
I
IS NAME TABLE FULL ?
YES
SET CENTRAL MEMORY TABLE FULL FLAG WORD
NO
WRITE OUT FIRST WORD OF ENTRY
XVI
GENERATE A VARIABLE NAME ENTRY
WRITE OUT SECOND WORD OF ENTRY
EXIT


48
CLEAR PROCESSOR STATUS TABLE ENTRY FOR THIS PROGRAM
DECREMENT "FEA" COUNT IN PROCESSOR COUNT WORD
SEARCH PROCESSOR STATUS TABLE FOR A NON-EMPTY ENTRY
FOUND ?
NO
GENERATE REQUEST FOR "PP4"
YES
DROP THIS PROGRAM

10. PROGRAM MSA - MISCELLANEOUS STATEMENT ANALYSER

10.0 Introduction

This program further processes all those statements not handled by programs GSR, DSA and FEA. Like program FEA, it is loaded on request by programs DSA and PCR when either of these programs detects that the processor 3 communication buffer is non-empty. The program continuously extracts and processes entries from its communication buffer until the latter is empty. If a subsequent scan through the processor status table indicates that there are no other programs still running, a request is made to the monitor to load program PP4. Program MSA then drops itself.

Of the statement types processed by program MSA, the DO statement is of particular interest. Not only must this statement type be syntax checked, but the validity of the nesting of DO loops must also be examined. However, since the latter cannot be done by program MSA (see 4.2), an entry is made in the DO table (see 5.1.4). The information for this table entry is built up during the syntax check of the statement. Appropriate table entry values are generated if either the DO loop control variable has not previously been referenced or the terminating statement number definition statement has not yet been processed. Eventually when all source statements have been processed and program PP4 is loaded, the DO table will contain an entry for every DO statement in the current source subprogram. Therefore, program PP4 will be able to validate DO loop nesting.

Program MSA handles some communication buffer strings which have been already partially processed by program GSR. These are the second part of arithmetic and two-branch logical IF statements (see 4.1 and note XI of 10.1) and are further processed here because they both contain statement number references, which are only processed by program MSA. Similarly, the special IF statements, which are conditional on file status (see note X of 10.1), involve statement number references as well as logical unit numbers which are also only processed by this program.

The ENTRY statement is processed by this program rather than program DSA because this declarative statement does not generally appear before the first executable statement. However, program DSA is organized in such a way that it can only process declaratives occurring before the first executable statement.

10.1 Explanatory Notes to the Flow-Chart

Although not explicitly stated in the accompanying flow-chart, references to all tables are made only after access to the required table has been successfully requested, using the procedure described in 5.1.0.

The following notes refer to the flow-chart:

I. This check is to determined whether the name represents an unsubscripted integer variable. If it is not already defined in the name table, an entry will be created for it and its type will be determined implicitly.

II. Both GOTO (10,12,14) J

and GOTO (10,12,14), J

are acceptable.

III. A statement number cannot be a valid DO terminator if defined or referenced as a FORMAT statement number. It is also invalid if the statement associated with that statement number is an arithmetic IF, GOTO, two-branch logical IF, RETURN or another DO statement. This information is stored in the statement label table (see 5.4).

IV. At this stage, the source line number of the DO statement and a pointer to the statement label table entry for the terminating statement number are inserted into a buffer area which will be inserted into the DO table as soon as the pointer to the name table entry for the DO loop control variable is known. See also note XVI below.

V. STOP and PAUSE statements may be followed by up to five octal digits.

VI. This is the same routine as used in program FEA.

VII. A name cannot be a valid ENTRY name unless its only previous occurrence was in a type declarative statement which did not specify dimensions.

VIII. These statements must take one of the forms:

ENCODE (c,u,v)L

or DECODE (c,u,v)L

where c is an unsigned integer constant or simple integer variable

u is a FORMAT statement number

v is a variable or array name

L is an input/output list.

IX. These statements take one of the forms:

BUFFER IN (u,c) (A,B)

or BUFFER OUT (u,c) (A,B)

where u is a logical unit number

c is an unsigned integer constant

A,B are variable names and/or array elements.

X. These statements must take one of the forms:

IF (ENDFILE u) n_1, n_2

or IF (EOF, u) n_1, n_2

or IF (IOCHECK, u) n_1, n_2

or IF (UNIT, u) n_1, n_2, n_3, n_4

where u is a logical unit number

n_1, n_2, n_3, n_4 are statement numbers.

XI. These are the statement numbers extracted from arithmetic IF and two-branch logical IF statements by program GSR.

XII. It is not essential that field descriptors be separated by commas.

XIII. Field descriptors X and 1X are equally acceptable in a FORMAT specification.

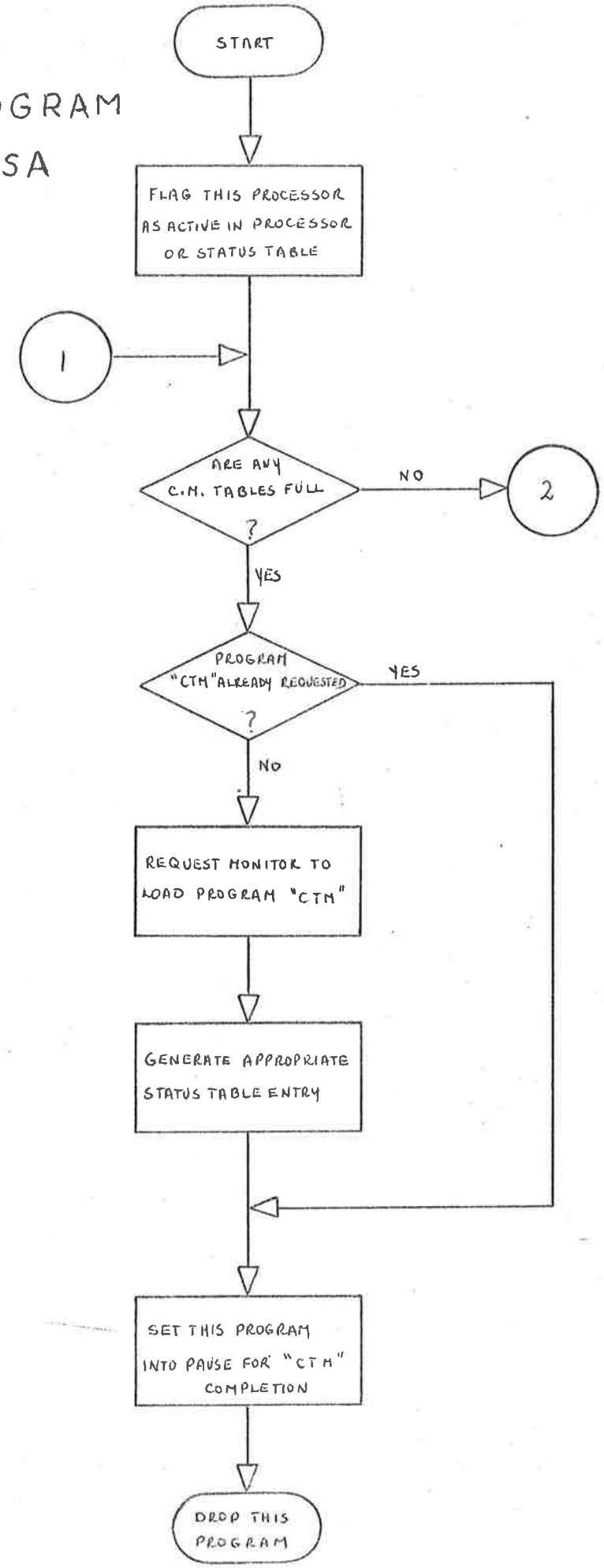
XIV. Numeric field descriptors F3 and F3.0 are equally acceptable. This is also true for D, E and G conversion codes.

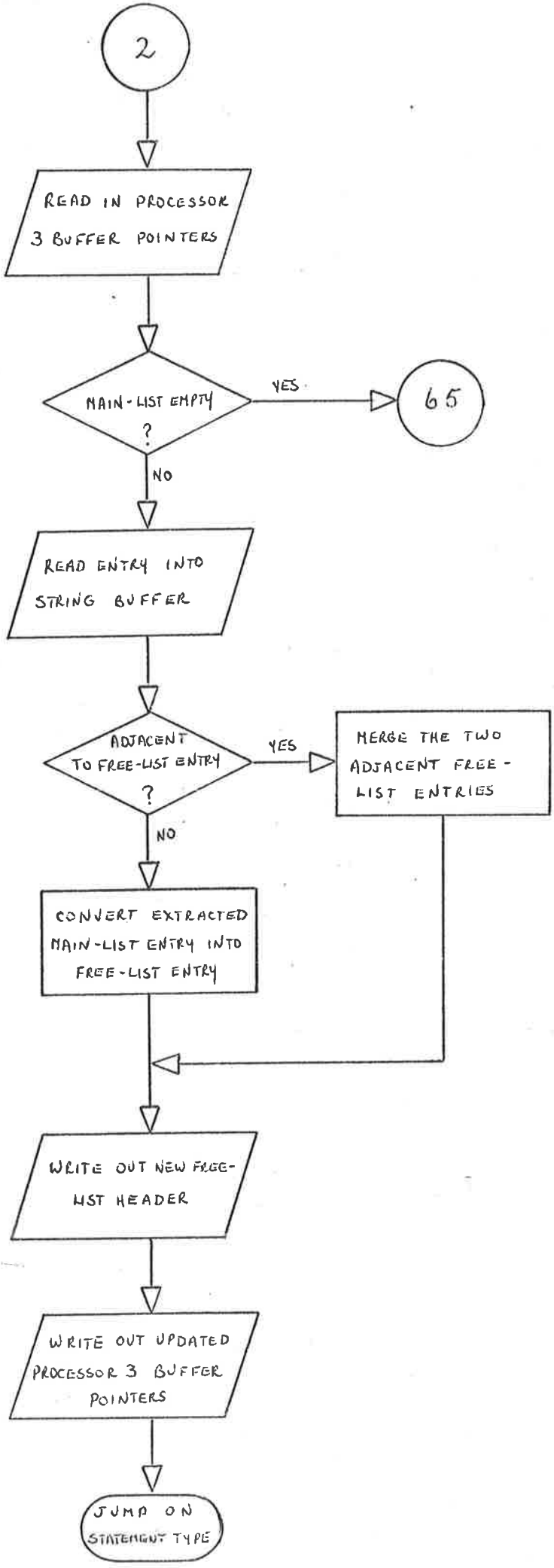
XV. Since this routine is called during the processing of DO statements as well as during the checking of indexed input/output lists, the error flagged is different in each case. For DO statement processing, it is error number 24 whilst for the other statement types it is error number 46.

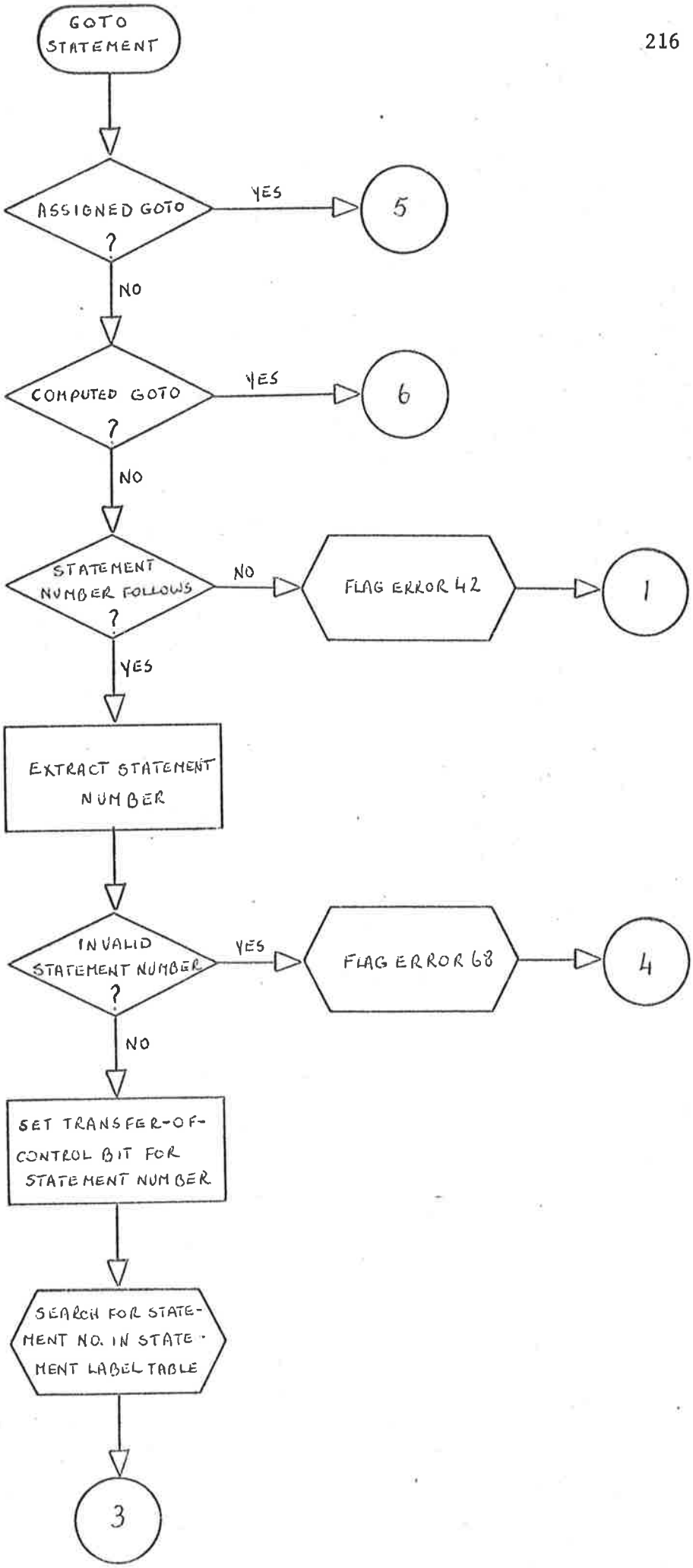
XVI. Since the pointer to the name table entry for the DO loop control variable is now known, the last part of the DO table entry can be generated. See also note IV above.

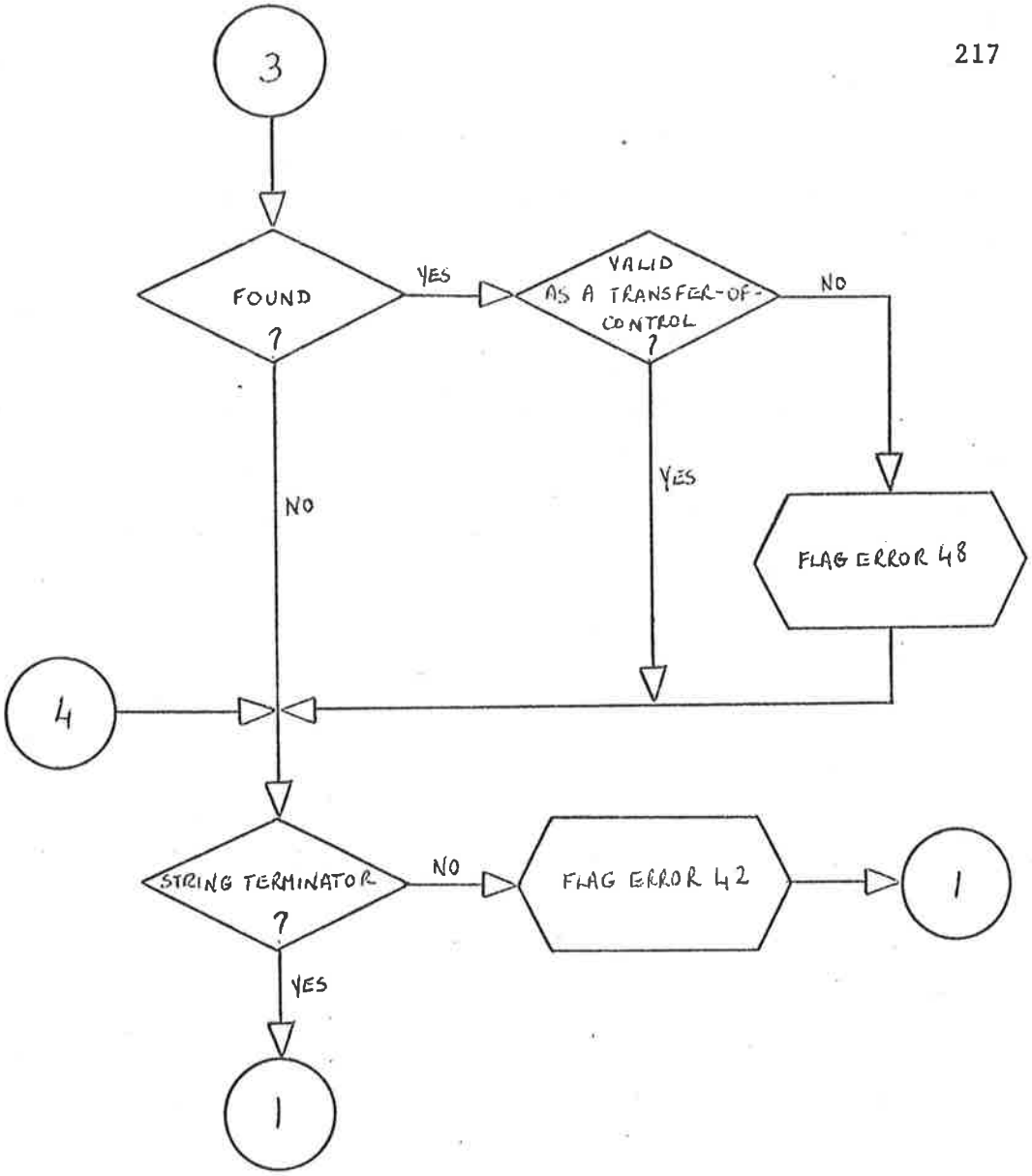
XVII. In this context, a FORMAT statement number is valid if it has been defined as such. However, if it has not yet been defined, it is valid if it has not previously been referenced by a transfer of control statement or a DO statement.

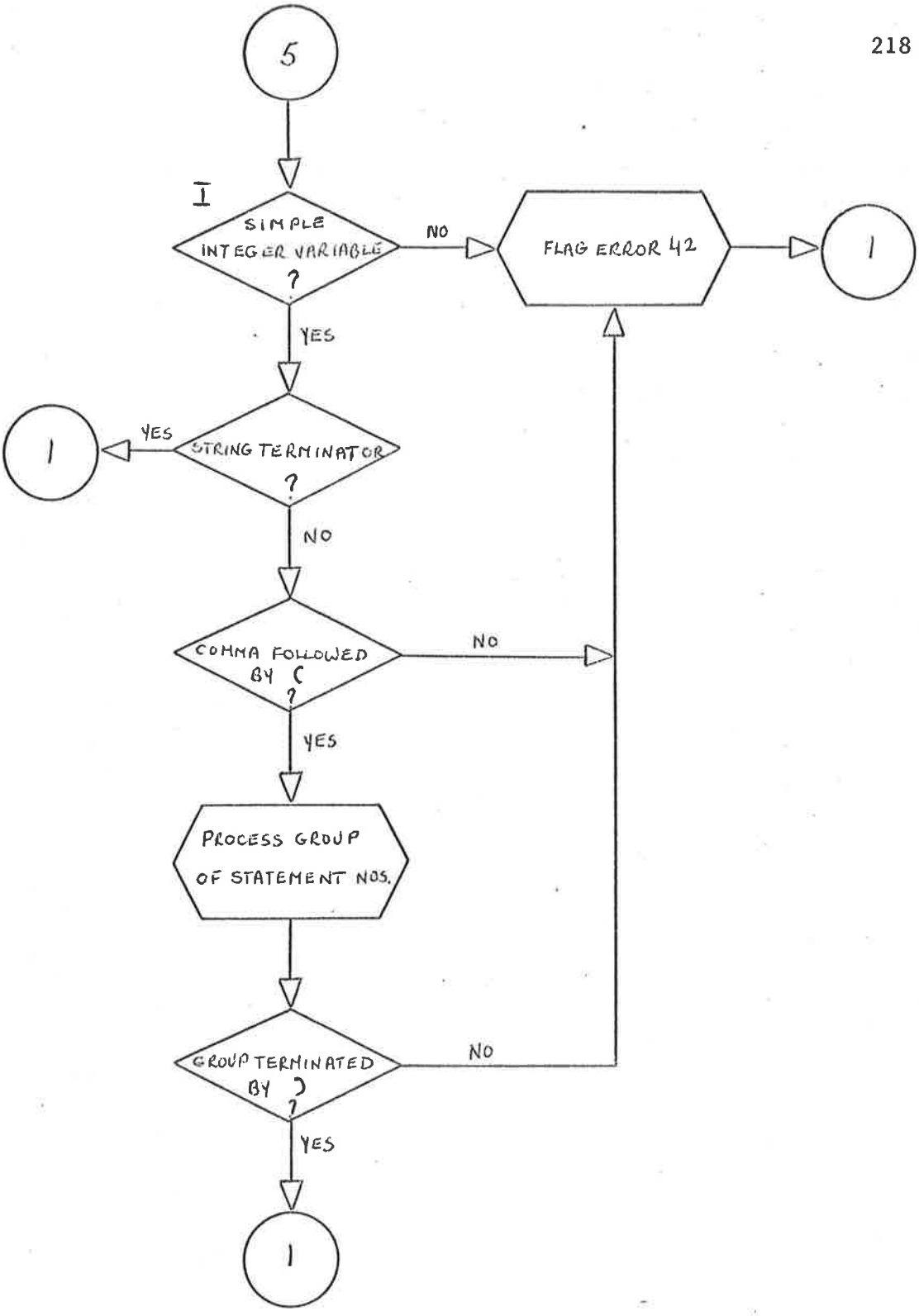
PROGRAM MSA

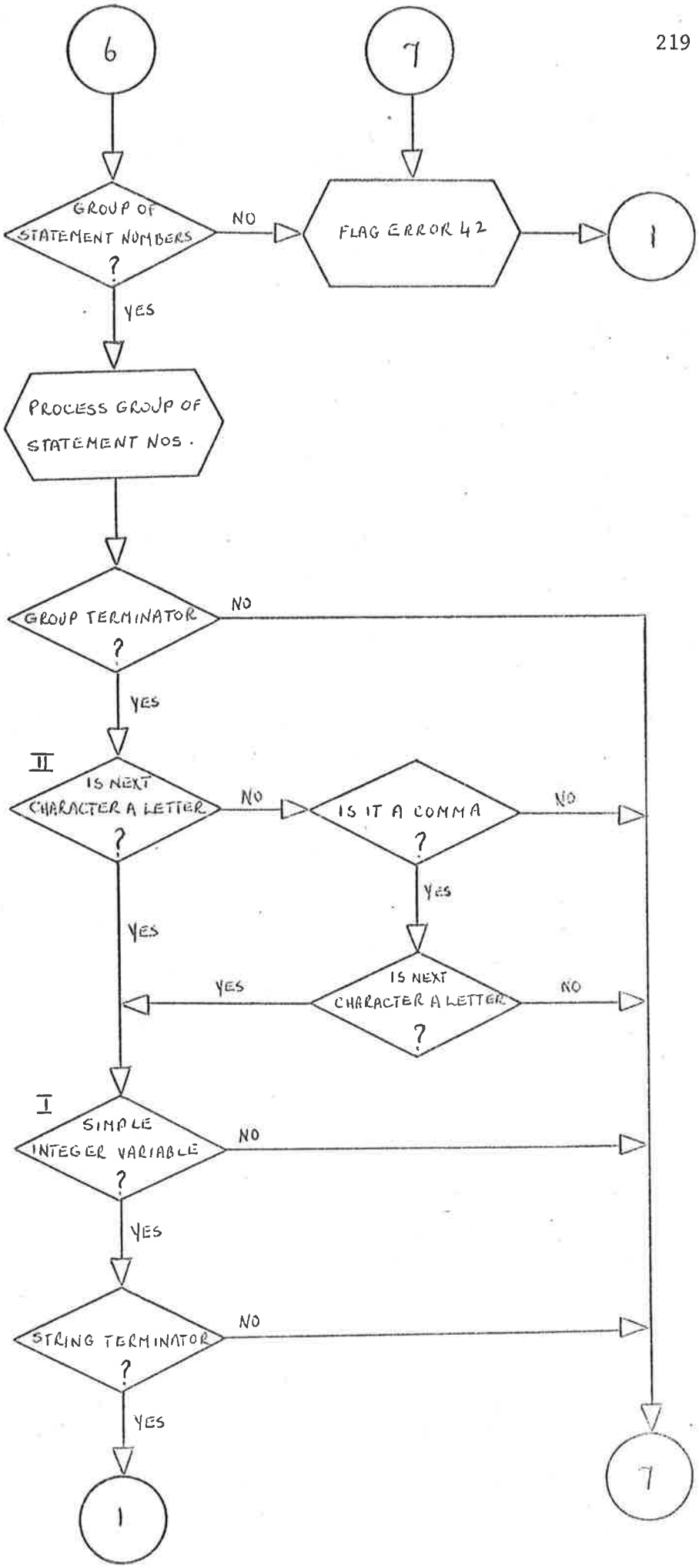

START
FLAG THIS PROCESSOR AS ACTIVE IN PROCESSOR OR STATUS TABLE
1
ARE ANY C.M. TABLES FULL ?
NO
2
YES
PROGRAM "CTM" ALREADY REQUESTED ?
YES
No
REQUEST MONITOR TO LOAD PROGRAM "CTM"
GENERATE APPROPRIATE STATUS TABLE ENTRY
SET THIS PROGRAM INTO PAUSE FOR "CTM" COMPLETION
DROP THIS PROGRAM

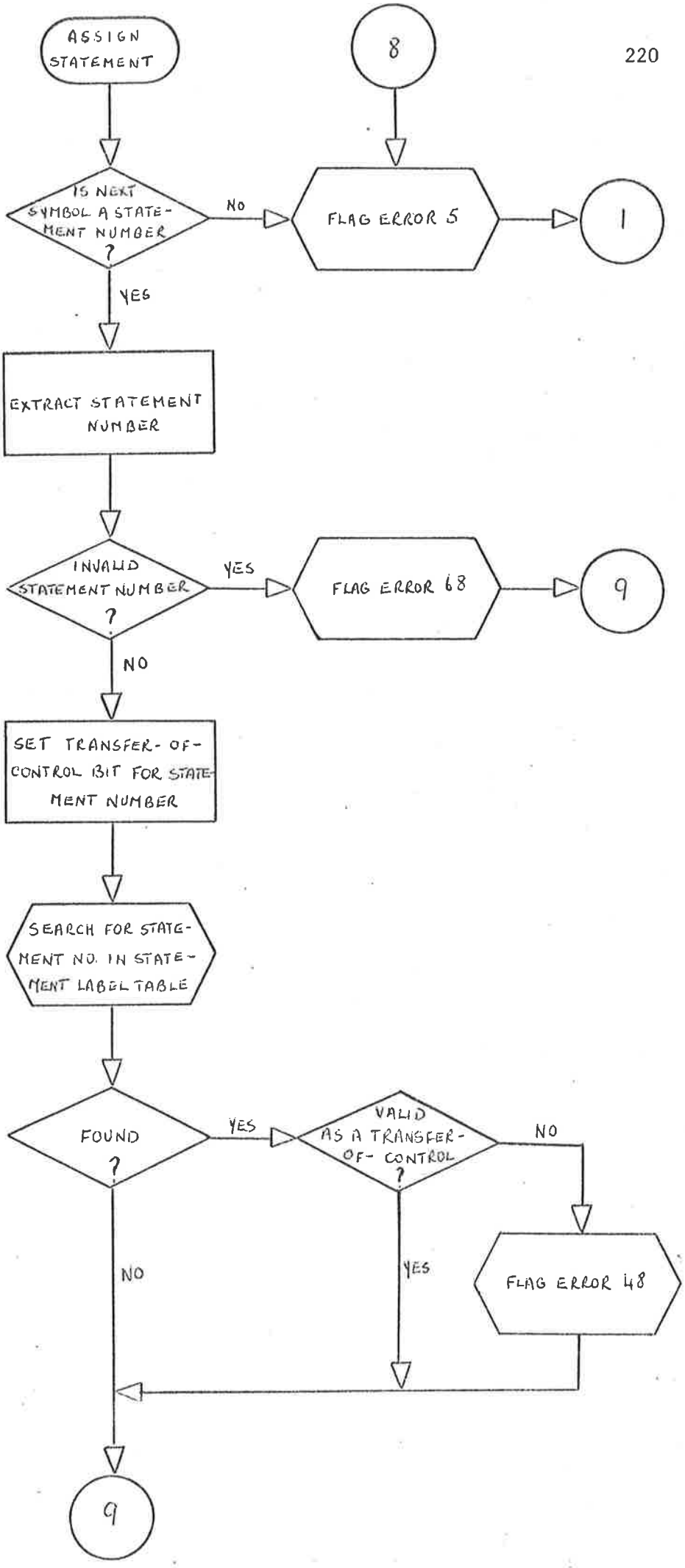

2
READ IN PROCESSOR 3 BUFFER POINTERS
MAIN-LIST EMPTY ?
YES
65
NO
READ ENTRY INTO STRING BUFFER
ADJACENT TO FREE-LIST ENTRY ?
YES
MERGE THE TWO ADJACENT FREE-LIST ENTRIES
NO
CONVERT EXTRACTED MAIN-LIST ENTRY INTO FREE-LIST ENTRY
WRITE OUT NEW FREE-LIST HEADER
WRITE OUT UPDATED PROCESSOR 3 BUFFER POINTERS
JUMP ON STATEMENT TYPE

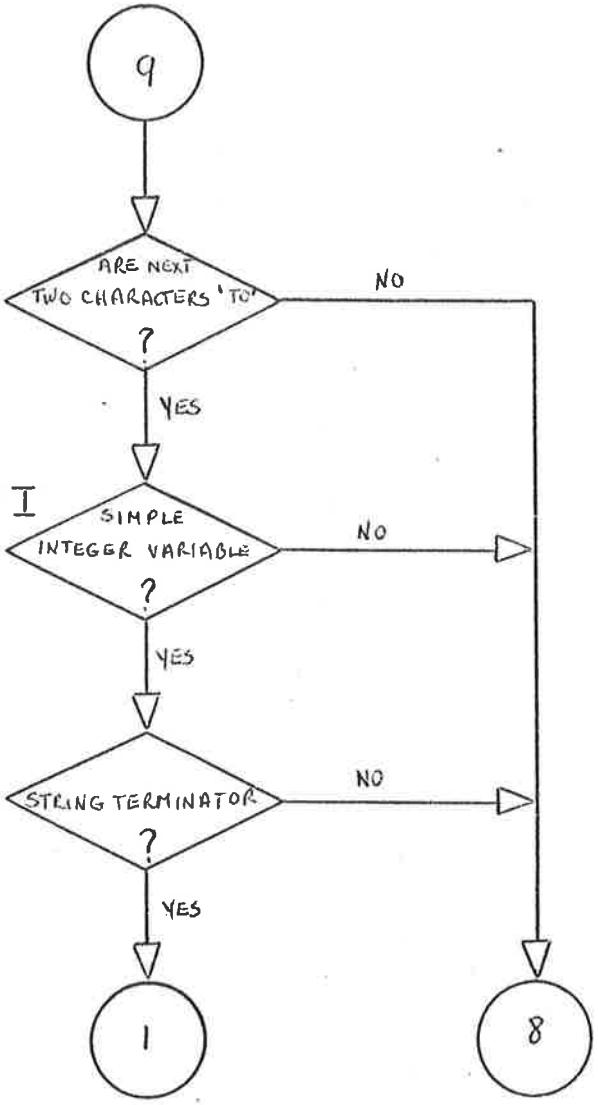

GOTO STATEMENT
ASSIGNED GOTO ?
YES
5
NO
COMPUTED GOTO ?
YES
6
NO
STATEMENT NUMBER FOLLOWS ?
NO
FLAG ERROR 42
1
YES
EXTRACT STATEMENT NUMBER
INVALID STATEMENT NUMBER ?
YES
FLAG ERROR 68
4
NO
SET TRANSFER-OF-CONTROL BIT FOR STATEMENT NUMBER
SEARCH FOR STATEMENT NO. IN STATEMENT LABEL TABLE
3

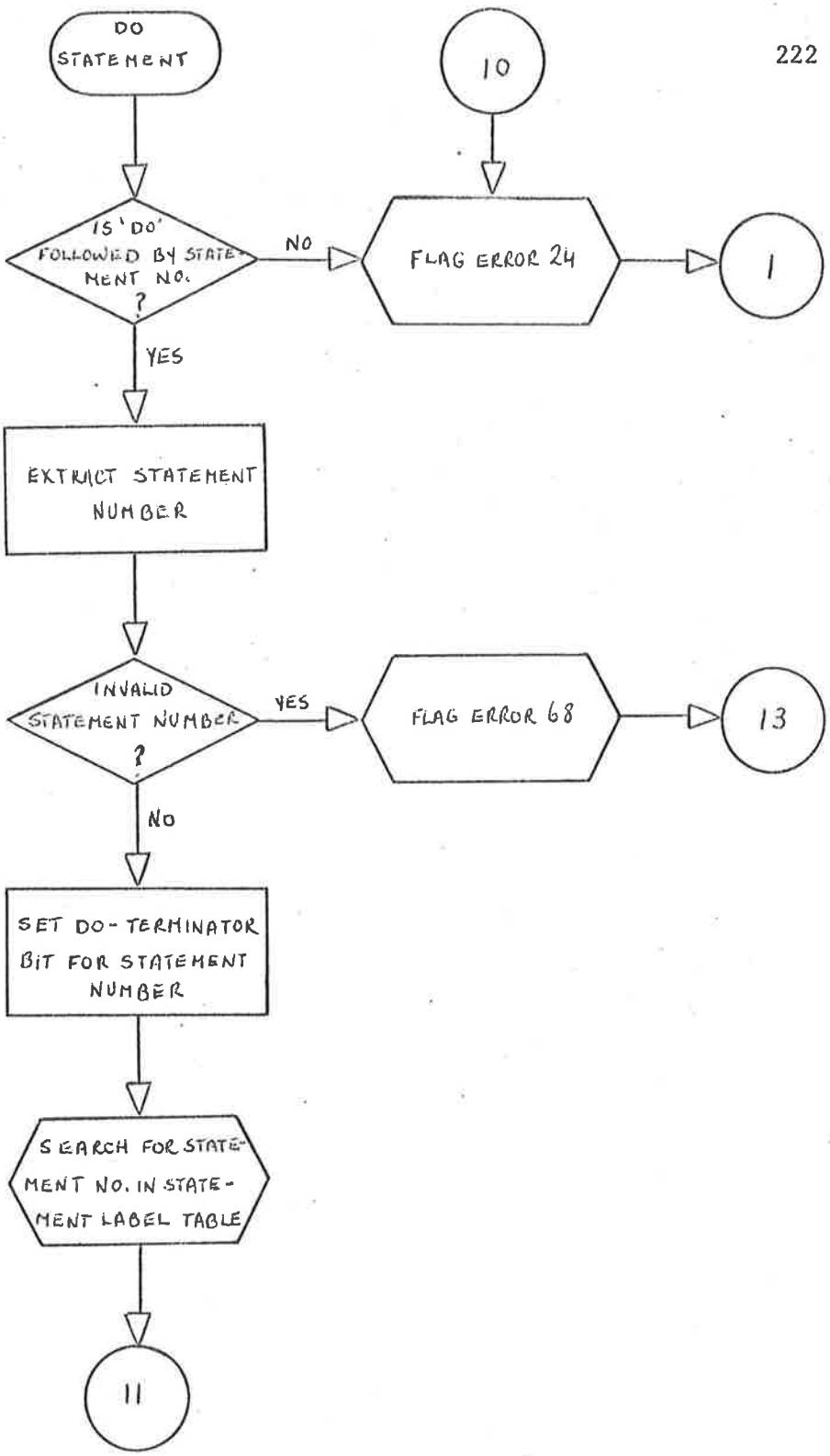

3
FOUND ?
YES
VALID AS A TRANSFER-OF-CONTROL ?
NO
YES
NO
FLAG ERROR 48
4
STRING TERMINATOR ?
NO
FLAG ERROR 42
1
YES
1

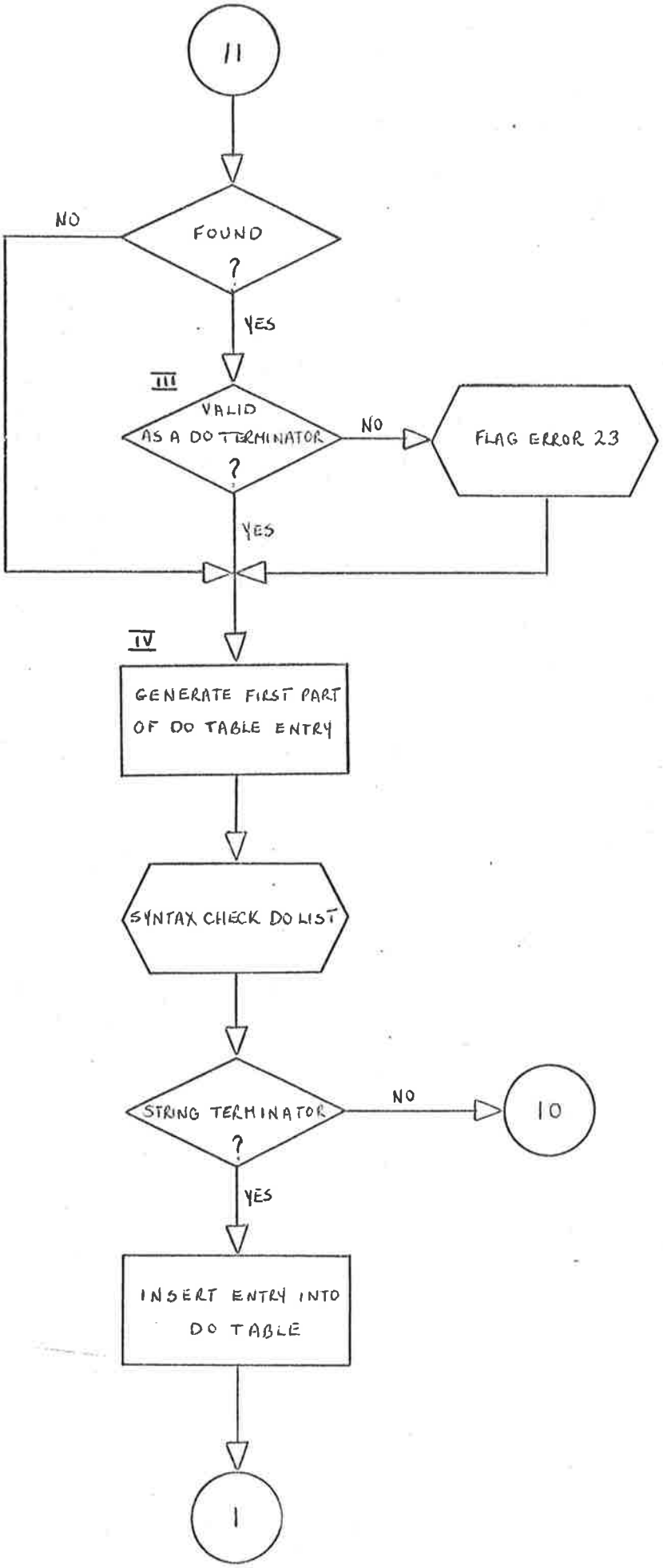

5
I
SIMPLE INTEGER VARIABLE ?
NO
FLAG ERROR 42
1
YES
1
YES
STRING TERMINATOR ?
NO
COMMA FOLLOWED BY (?
NO
YES
PROCESS GROUP OF STATEMENT NOS.
GROUP TERMINATED BY) ?
NO
YES
1

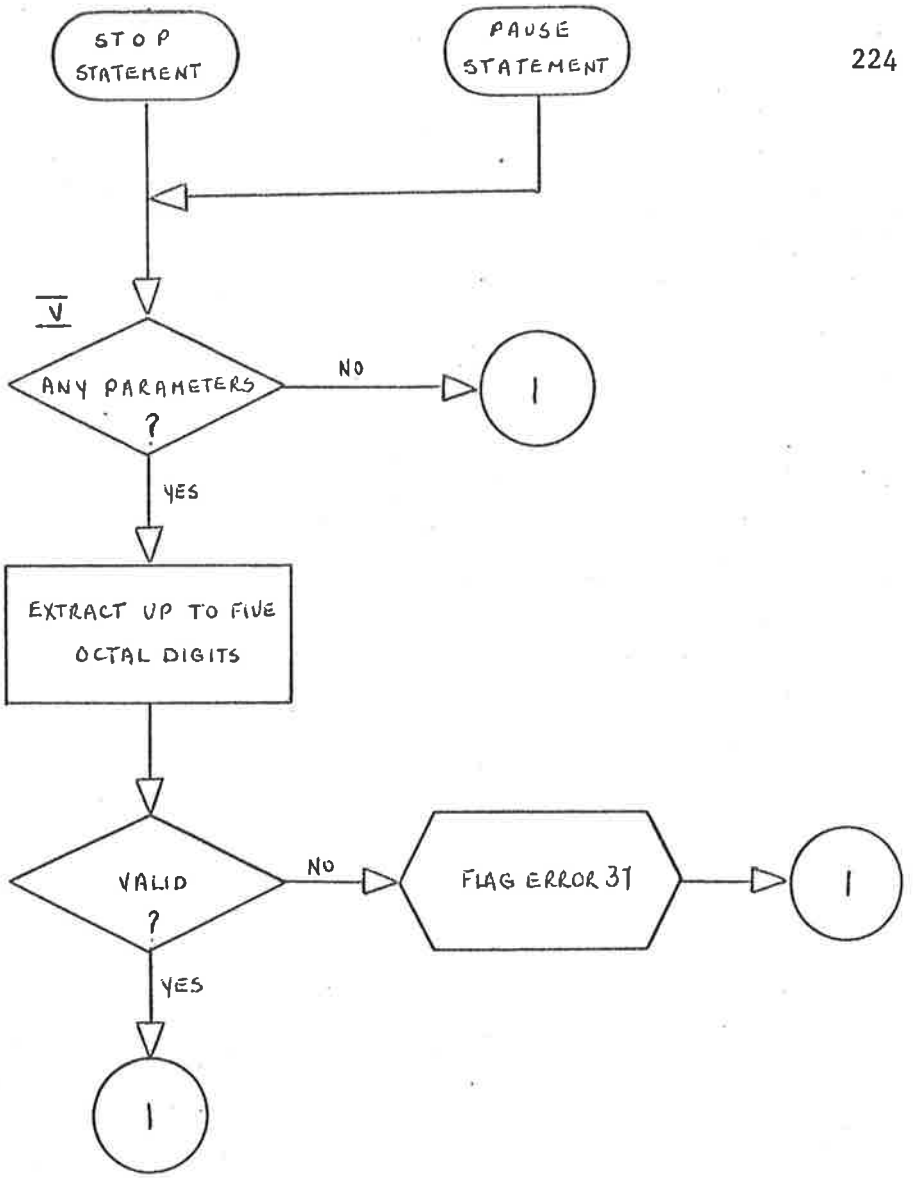

6
7
GROUP OF STATEMENT NUMBERS ?
NO
FLAG ERROR 42
1
YES
PROCESS GROUP OF STATEMENT NOS.
GROUP TERMINATOR ?
NO
YES
II
IS NEXT CHARACTER A LETTER ?
NO
IS IT A COMMA ?
NO
YES
YES
IS NEXT CHARACTER A LETTER ?
YES
NO
I
SIMPLE INTEGER VARIABLE ?
NO
YES
STRING TERMINATOR ?
NO
YES
1
7

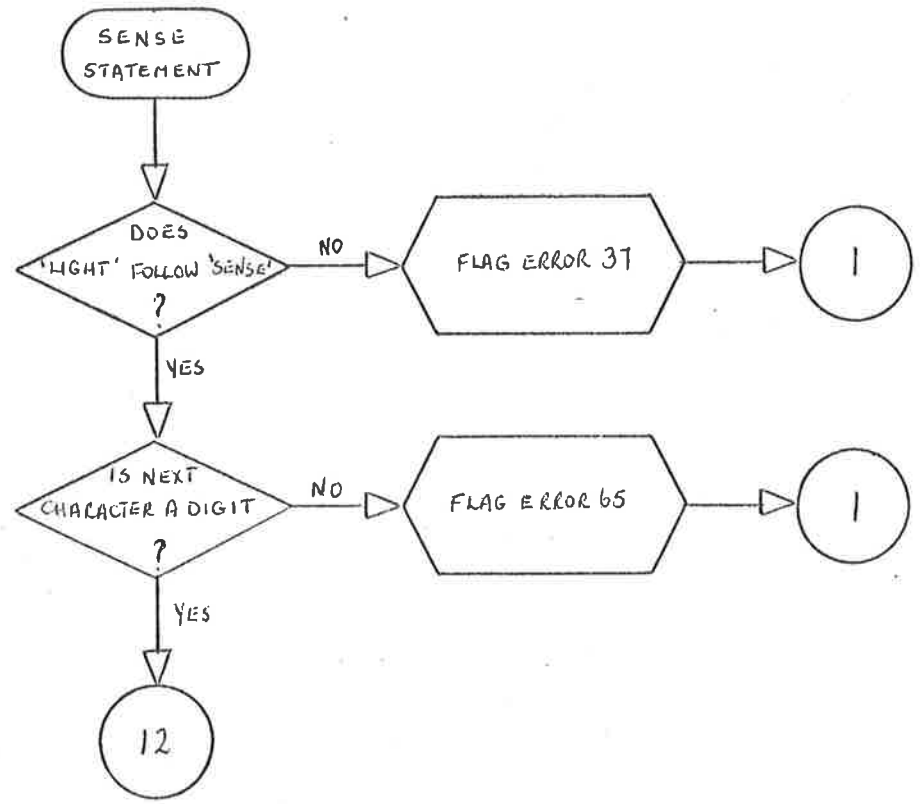

ASSIGN STATEMENT
8
IS NEXT SYMBOL A STATEMENT NUMBER ?
NO
FLAG ERROR 5
1
YES
EXTRACT STATEMENT NUMBER
INVALID STATEMENT NUMBER ?
YES
FLAG ERROR 68
9
NO
SET TRANSFER-OF-CONTROL BIT FOR STATEMENT NUMBER
SEARCH FOR STATEMENT NO. IN STATEMENT LABEL TABLE
FOUND ?
YES
VALID AS A TRANSFER-OF-CONTROL ?
NO
FLAG ERROR 48
NO
YES
9

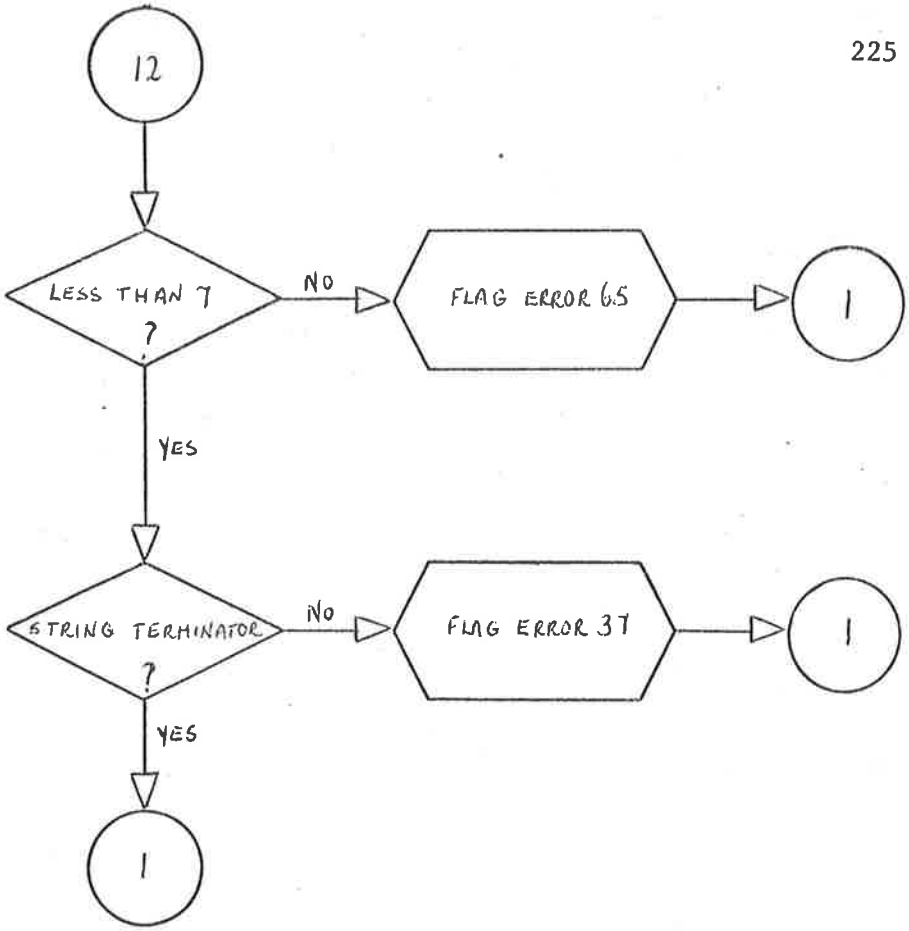

9
ARE NEXT TWO CHARACTERS 'TO' ?
NO
YES
I
SIMPLE INTEGER VARIABLE ?
NO
YES
STRING TERMINATOR ?
NO
YES
1
8

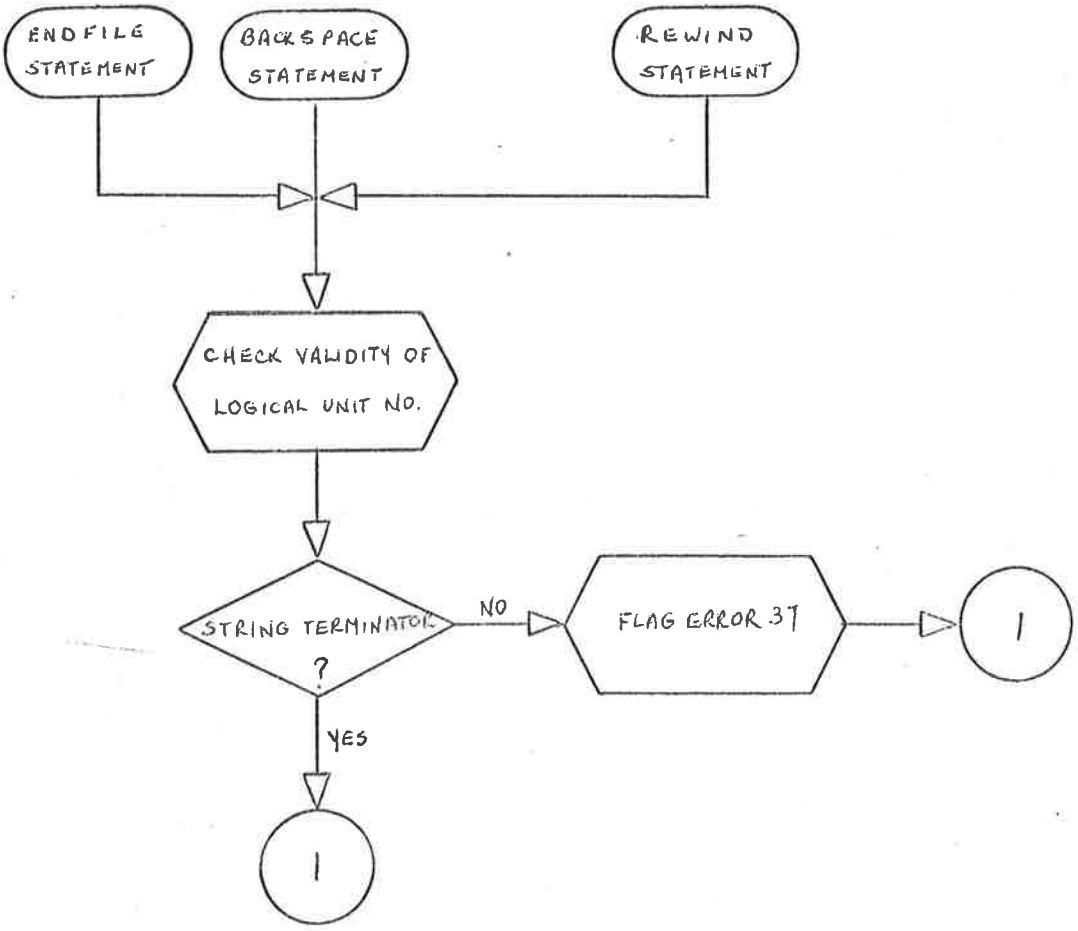

DO STATEMENT
10
IS 'DO' FOLLOWED BY STATEMENT NO. ?
NO
FLAG ERROR 24
1
YES
EXTRACT STATEMENT NUMBER
INVALID STATEMENT NUMBER ?
YES
FLAG ERROR 68
13
No
SET DO-TERMINATOR BIT FOR STATEMENT NUMBER
SEARCH FOR STATEMENT NO. IN STATEMENT LABEL TABLE
11

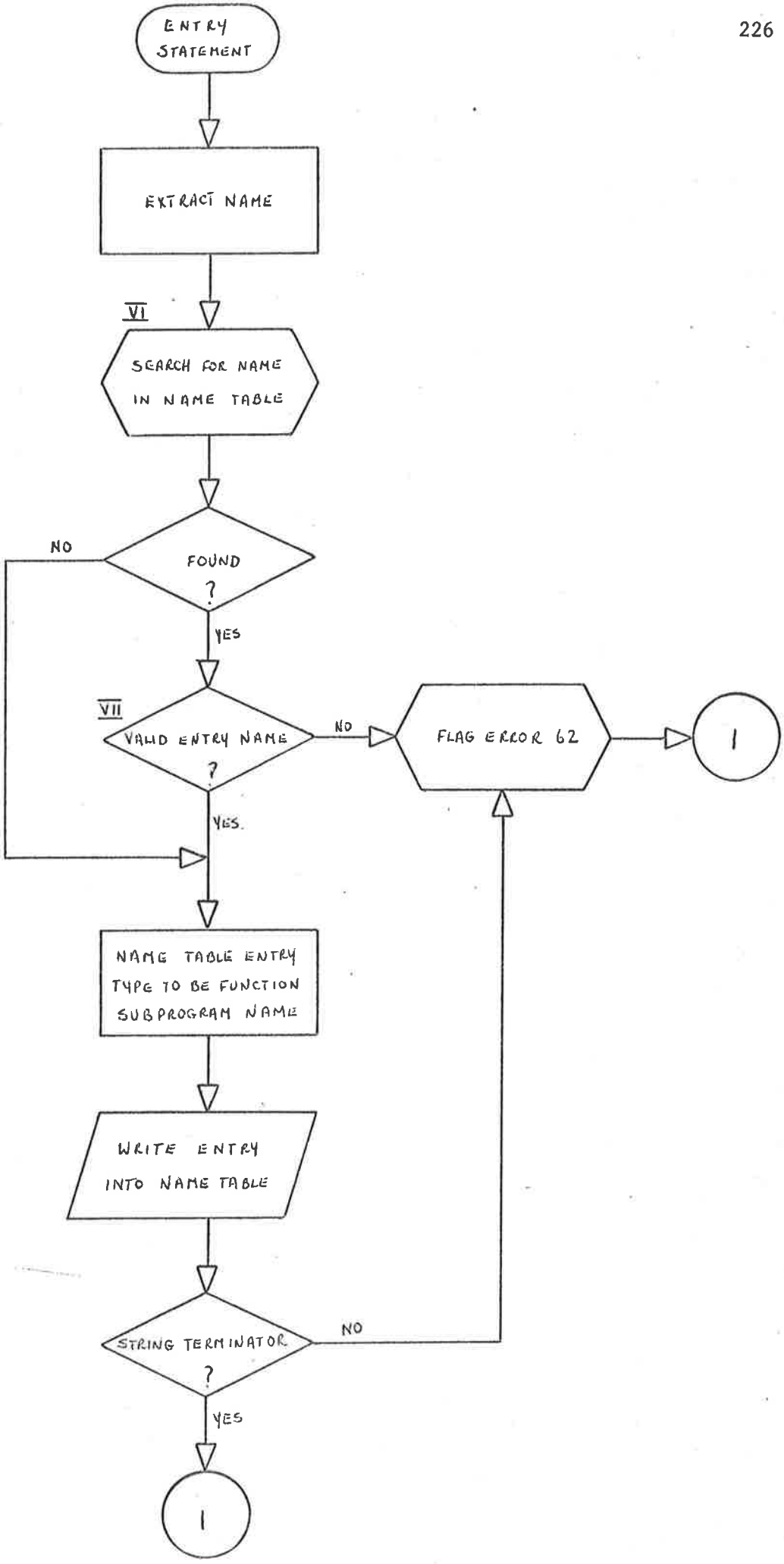

11
NO
FOUND
?
YES
III
VALID
AS A DO TERMINATOR
?
NO
FLAG ERROR 23
YES
IV
GENERATE FIRST PART
OF DO TABLE ENTRY
SYNTAX CHECK DO LIST
STRING TERMINATOR
?
NO
10
YES
INSERT ENTRY INTO
DO TABLE
1

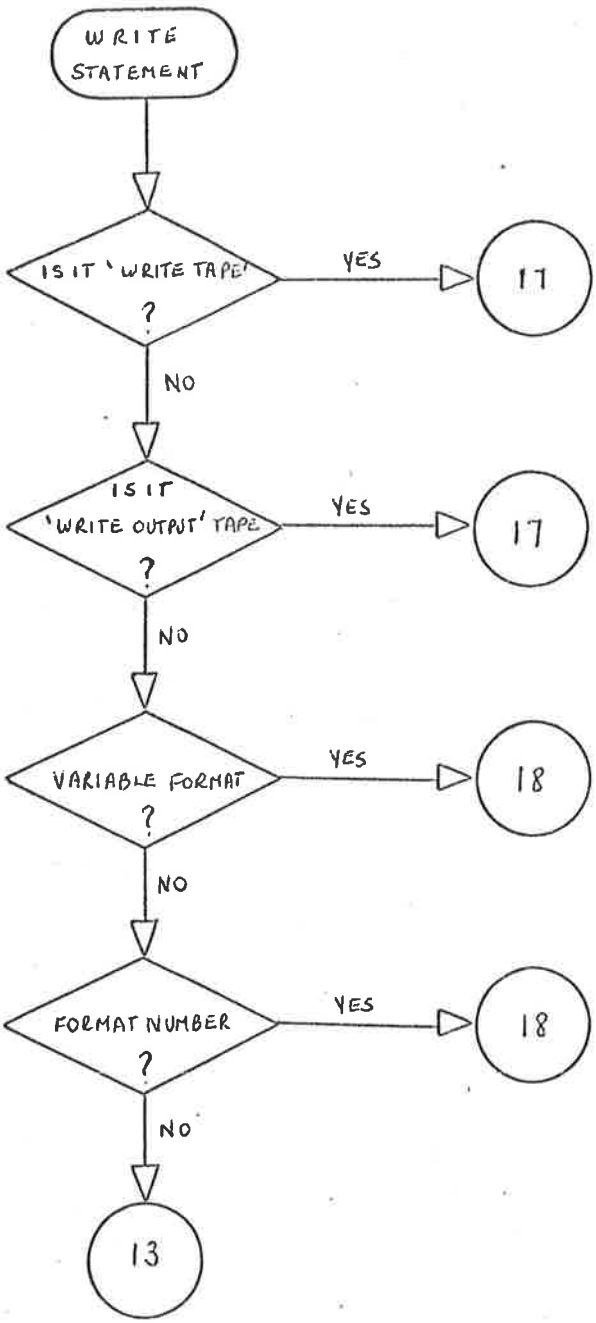

STOP STATEMENT
PAUSE STATEMENT
V
ANY PARAMETERS ?
NO
1
YES
EXTRACT UP TO FIVE OCTAL DIGITS
VALID ?
NO
FLAG ERROR 31
1
YES
1

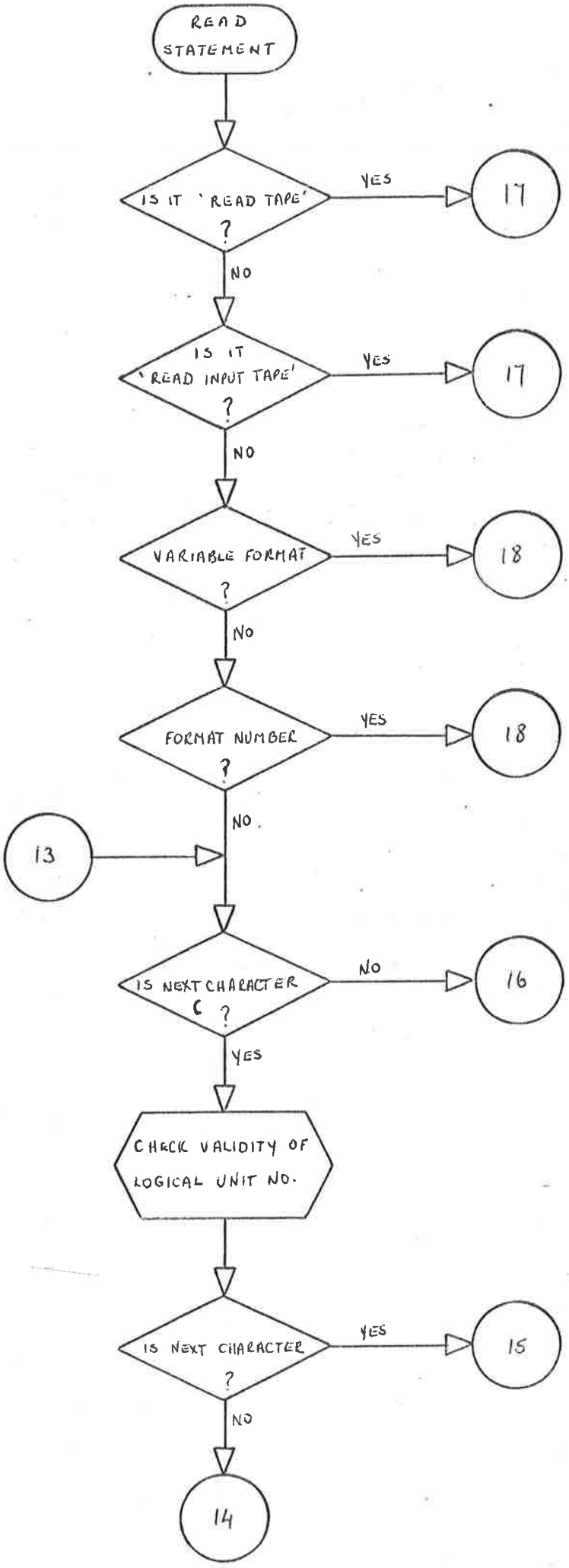

SENSE STATEMENT
DOES 'LIGHT' FOLLOW 'SENSE' ?
NO
FLAG ERROR 31
1
YES
IS NEXT CHARACTER A DIGIT ?
NO
FLAG ERROR 65
1
YES
12

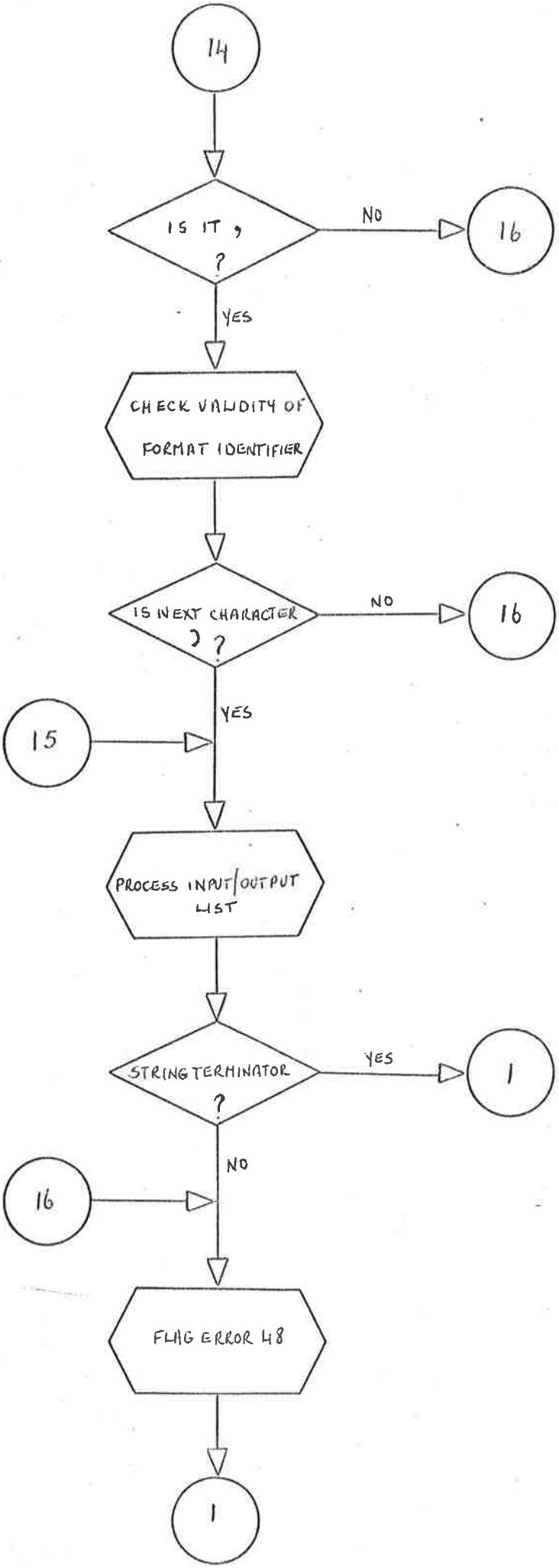

12
LESS THAN 7 ?
NO
FLAG ERROR 65
1
YES
STRING TERMINATOR ?
NO
FLAG ERROR 37
1
YES
1

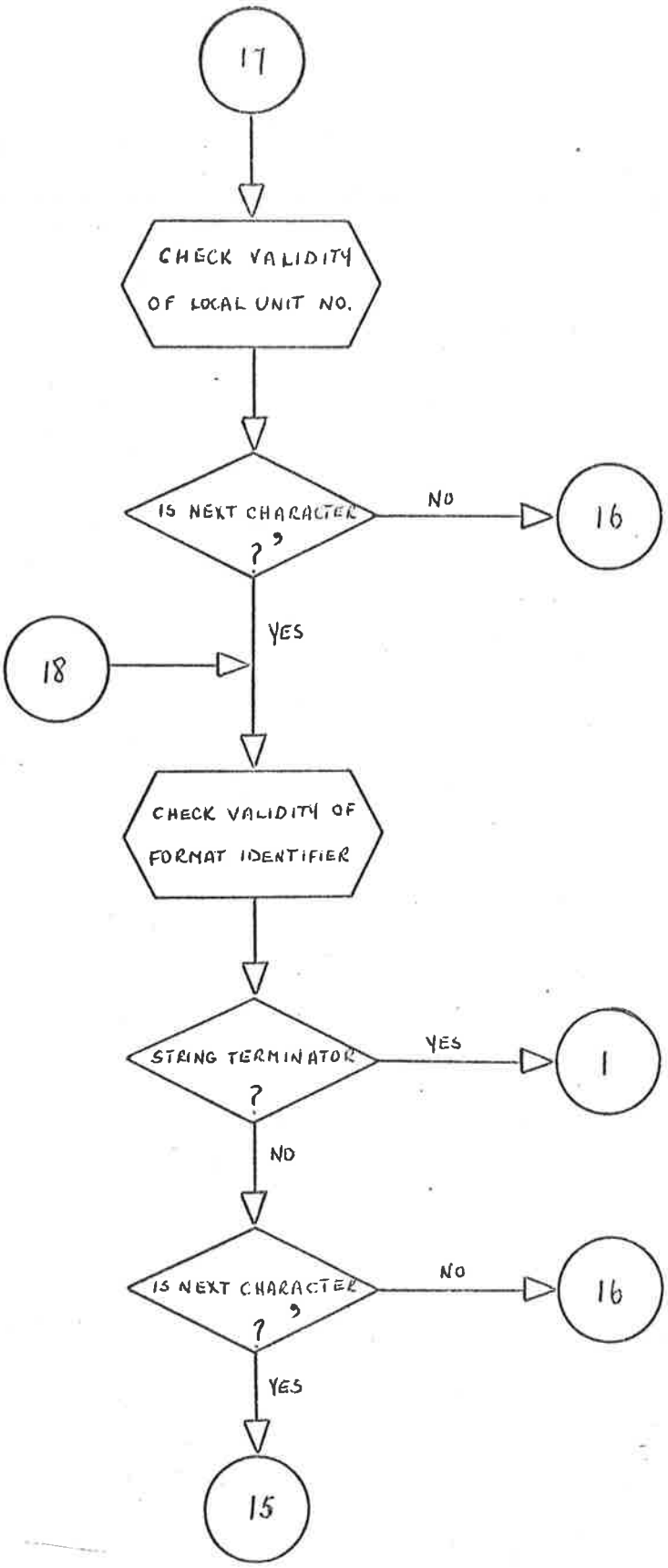

ENDFILE STATEMENT
BACKSPACE STATEMENT
REWIND STATEMENT
CHECK VALIDITY OF LOGICAL UNIT NO.
STRING TERMINATOR ?
NO
FLAG ERROR 37
1
YES
1


ENTRY STATEMENT
EXTRACT NAME
VI
SEARCH FOR NAME IN NAME TABLE
FOUND ?
NO
YES
VII
VALID ENTRY NAME ?
NO
FLAG ERROR 62
1
YES
NAME TABLE ENTRY TYPE TO BE FUNCTION SUBPROGRAM NAME
WRITE ENTRY INTO NAME TABLE
STRING TERMINATOR ?
NO
YES
1

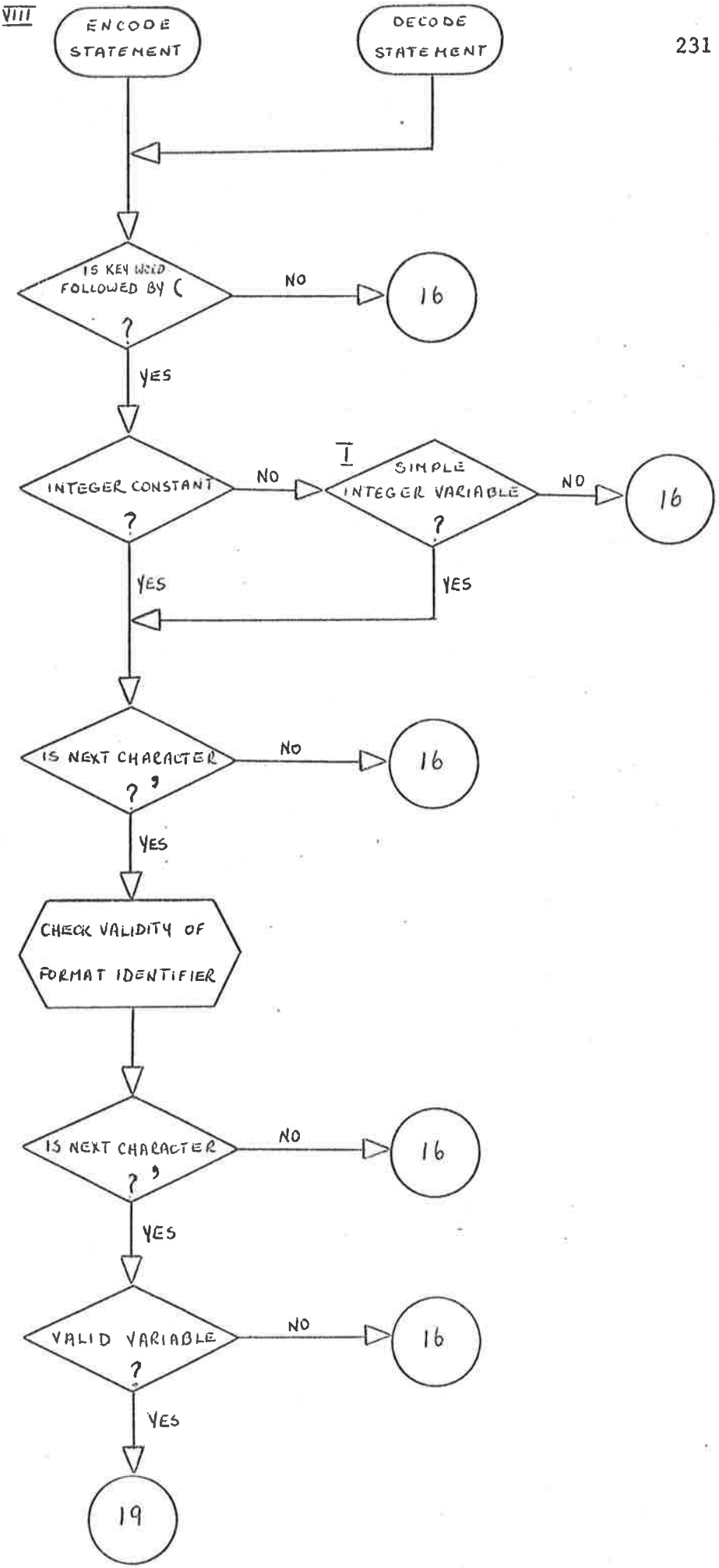

WRITE STATEMENT
IS IT 'WRITE TAPE' ?
YES
17
NO
IS IT 'WRITE OUTPUT' TAPE ?
YES
17
NO
VARIABLE FORMAT ?
YES
18
NO
FORMAT NUMBER ?
YES
18
NO
13

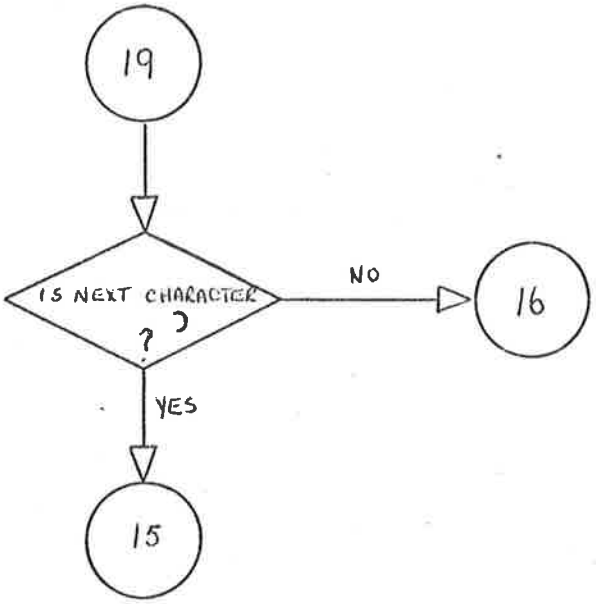

READ STATEMENT
IS IT 'READ TAPE' ?
YES
17
NO
IS IT 'READ INPUT TAPE' ?
YES
17
NO
VARIABLE FORMAT ?
YES
18
No
FORMAT NUMBER ?
YES
18
NO
13
IS NEXT CHARACTER C ?
No
16
YES
CHECK VALIDITY OF LOGICAL UNIT NO.
IS NEXT CHARACTER ?
YES
15
NO
14

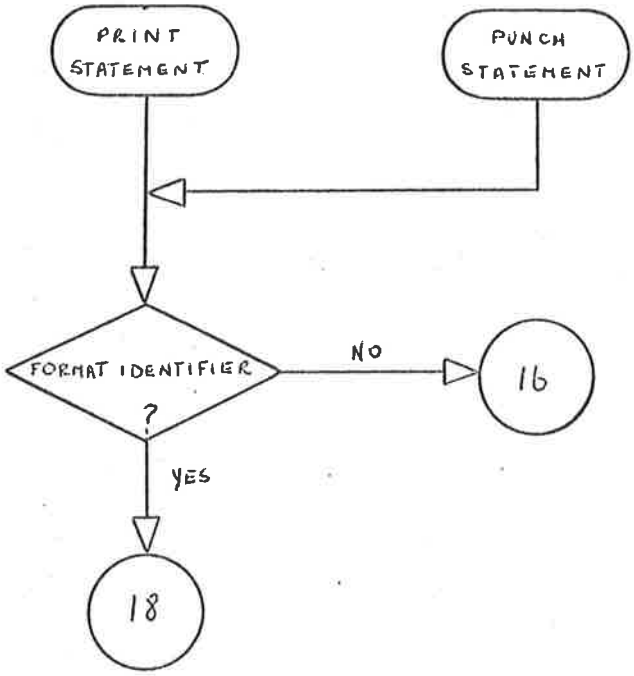

14
IS IT ,
?
NO
16
YES
CHECK VALIDITY OF
FORMAT IDENTIFIER
IS NEXT CHARACTER
) ?
NO
16
YES
15
PROCESS INPUT/OUTPUT
LIST
STRING TERMINATOR
?
YES
1
NO
16
FLAG ERROR 48
1

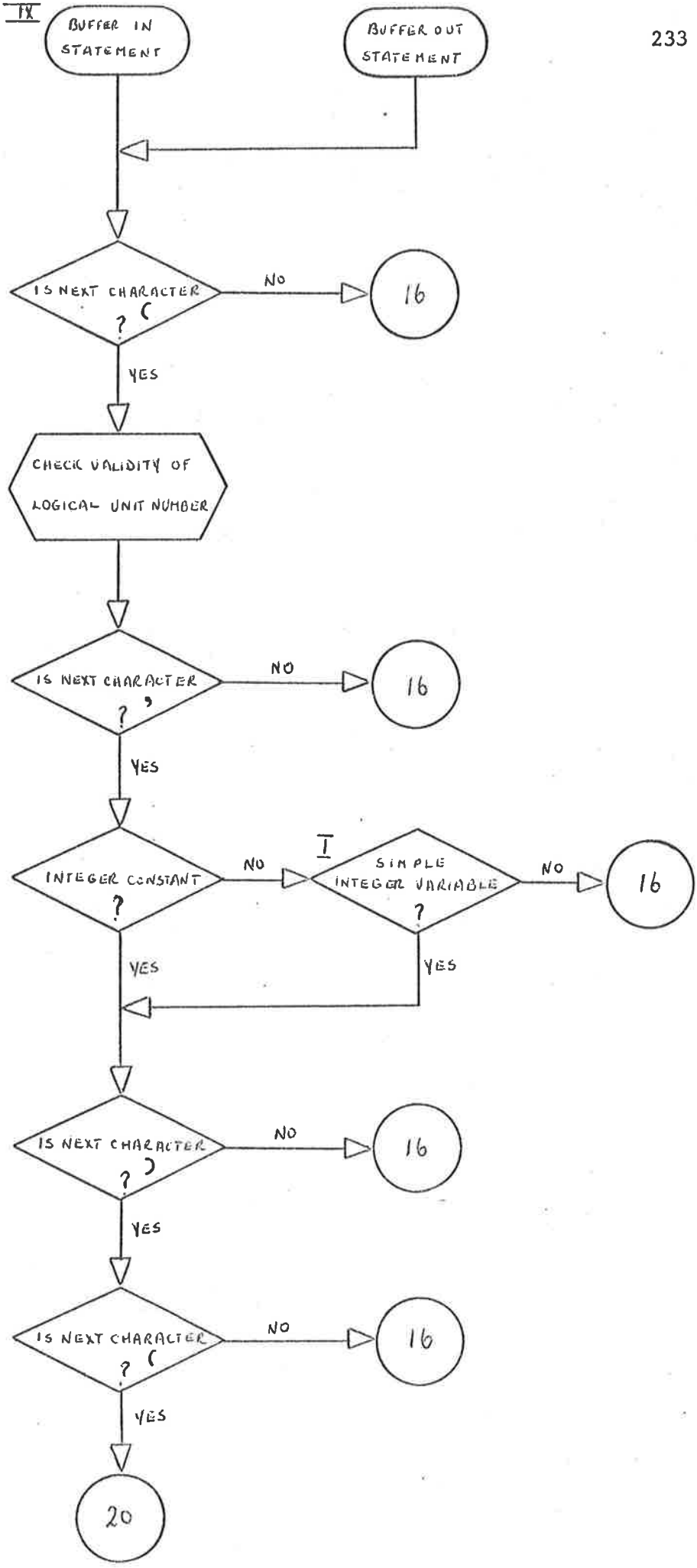

17
CHECK VALIDITY OF LOCAL UNIT NO.
IS NEXT CHARACTER ?'
NO
16
YES
18
CHECK VALIDITY OF FORMAT IDENTIFIER
STRING TERMINATOR ?
YES
1
NO
IS NEXT CHARACTER ?'
NO
16
YES
15

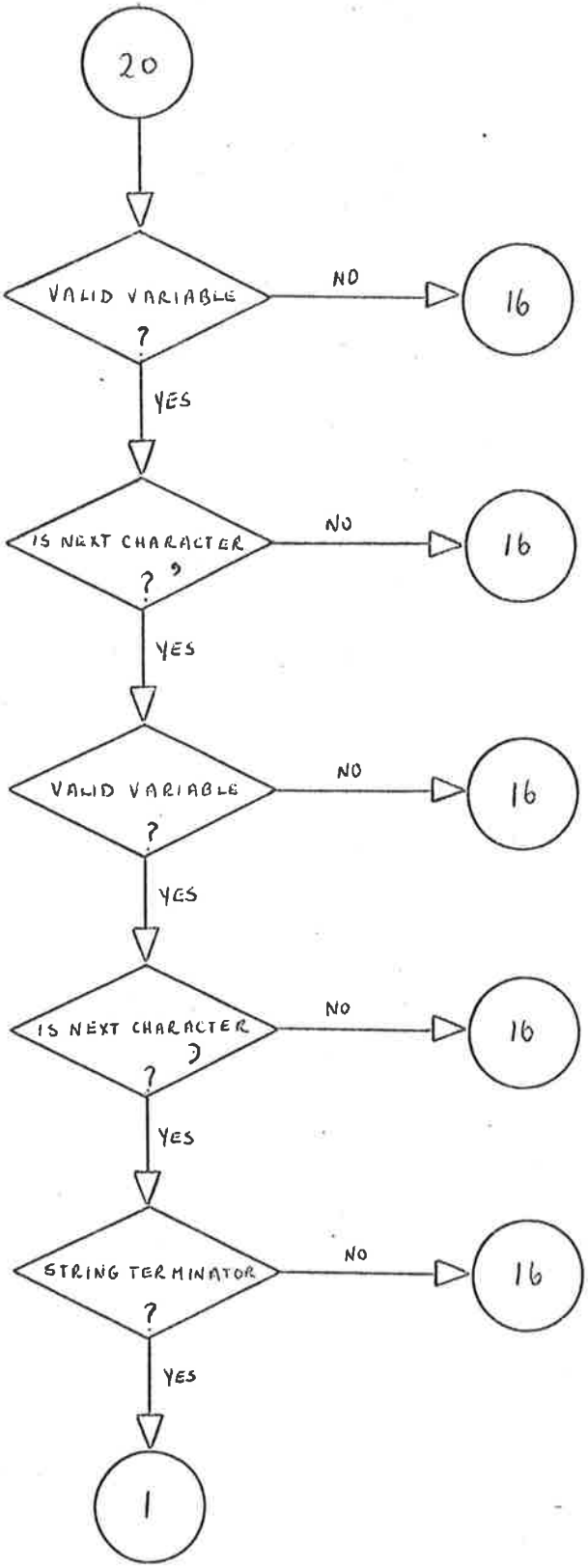
VIII

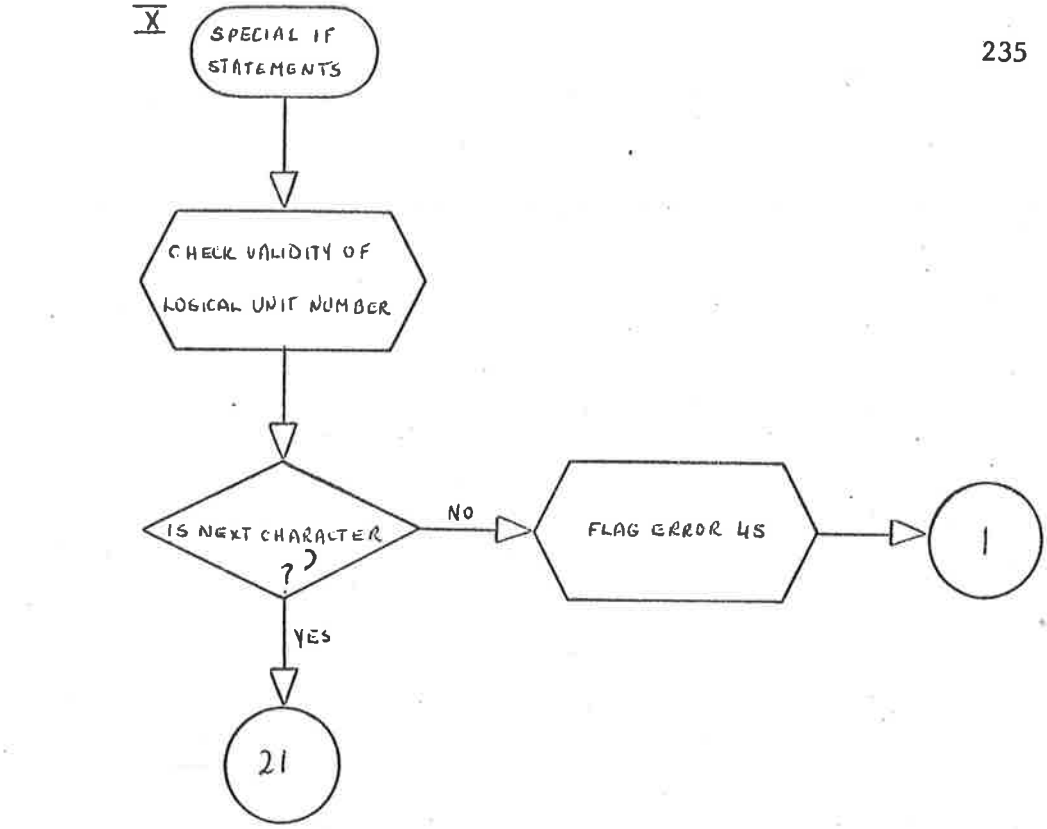

ENCODE STATEMENT
DECODE STATEMENT
IS KEY WORD FOLLOWED BY (?
NO
16
YES
INTEGER CONSTANT ?
NO
I
SIMPLE INTEGER VARIABLE ?
NO
16
YES
YES
IS NEXT CHARACTER ? '
NO
16
YES
CHECK VALIDITY OF FORMAT IDENTIFIER
IS NEXT CHARACTER ? '
NO
16
YES
VALID VARIABLE ?
NO
16
YES
19

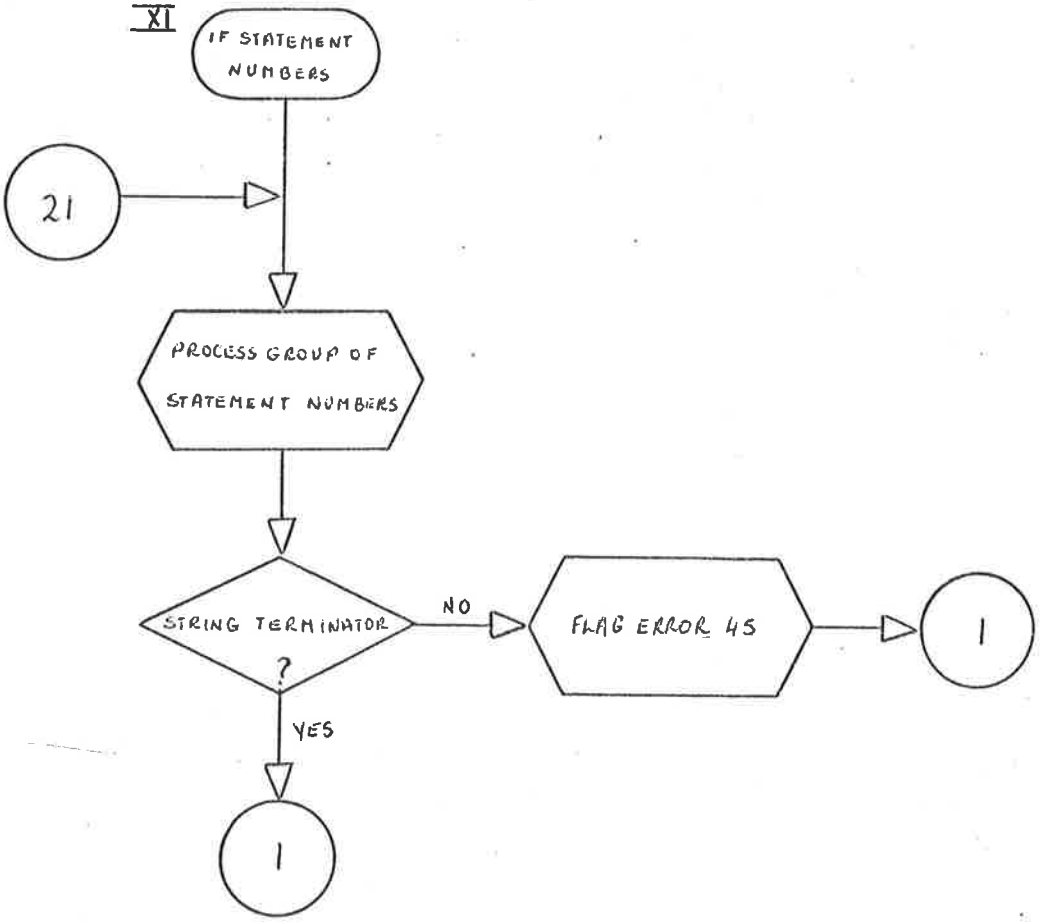

19
IS NEXT CHARACTER
?
NO
16
YES
15

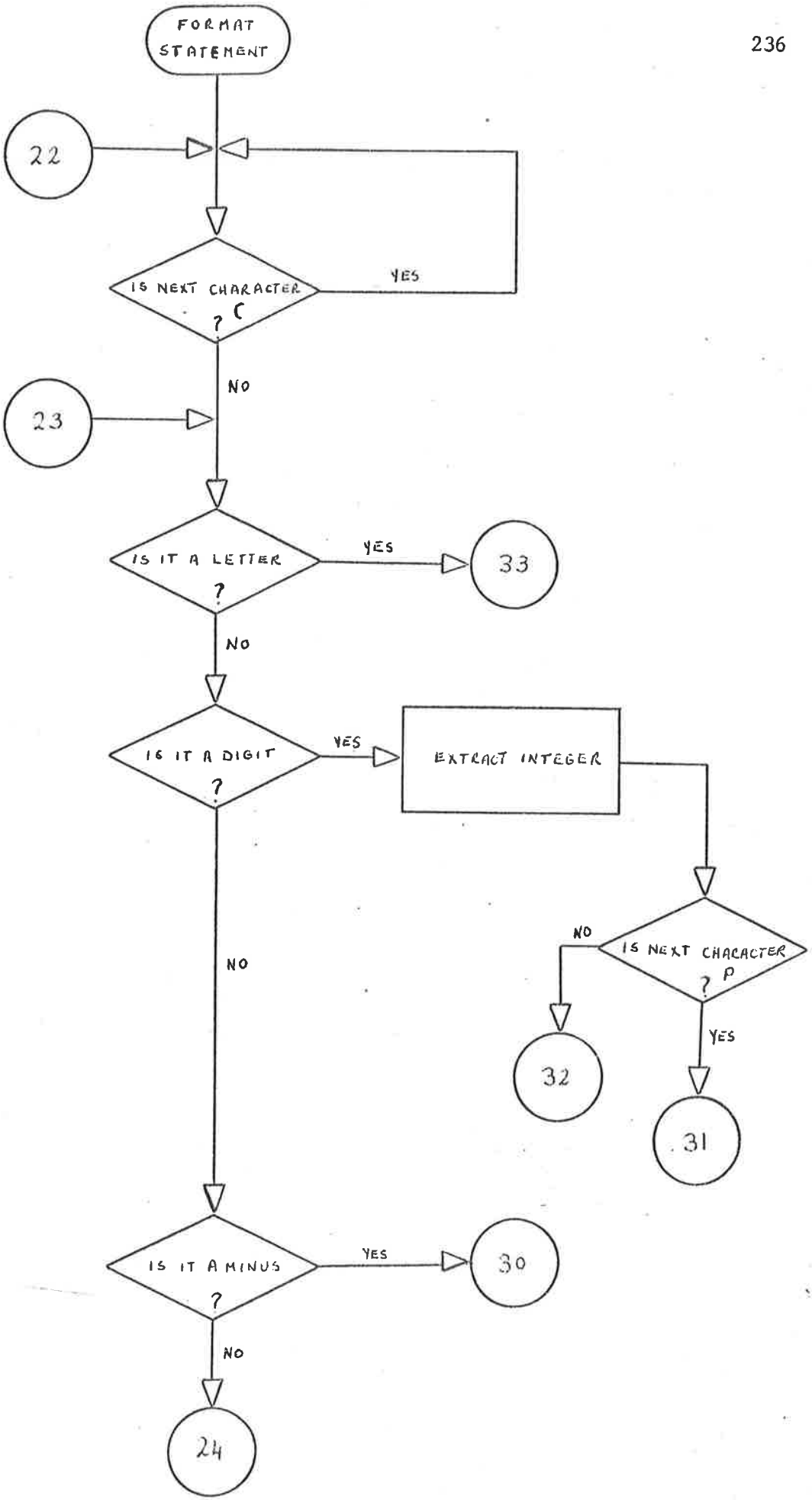

PRINT STATEMENT
PUNCH STATEMENT
FORMAT IDENTIFIER
?
NO
16
YES
18

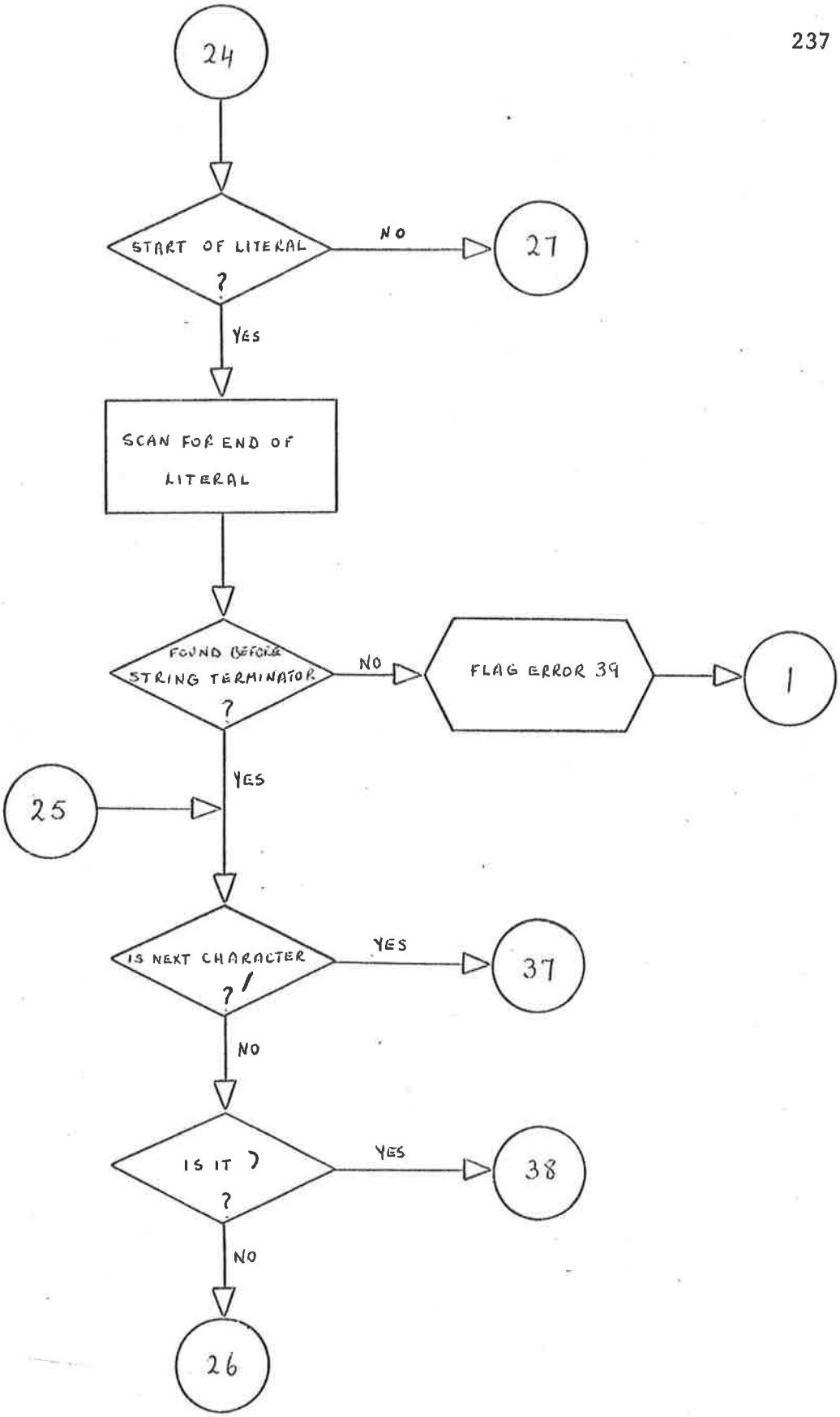

IX
BUFFER IN STATEMENT
BUFFER OUT STATEMENT
IS NEXT CHARACTER ? (
NO
16
YES
CHECK VALIDITY OF LOGICAL UNIT NUMBER
IS NEXT CHARACTER ? ,
NO
16
YES
INTEGER CONSTANT ?
NO
I
SIMPLE INTEGER VARIABLE ?
NO
16
YES
YES
IS NEXT CHARACTER ?)
NO
16
YES
IS NEXT CHARACTER ? (
NO
16
YES
20

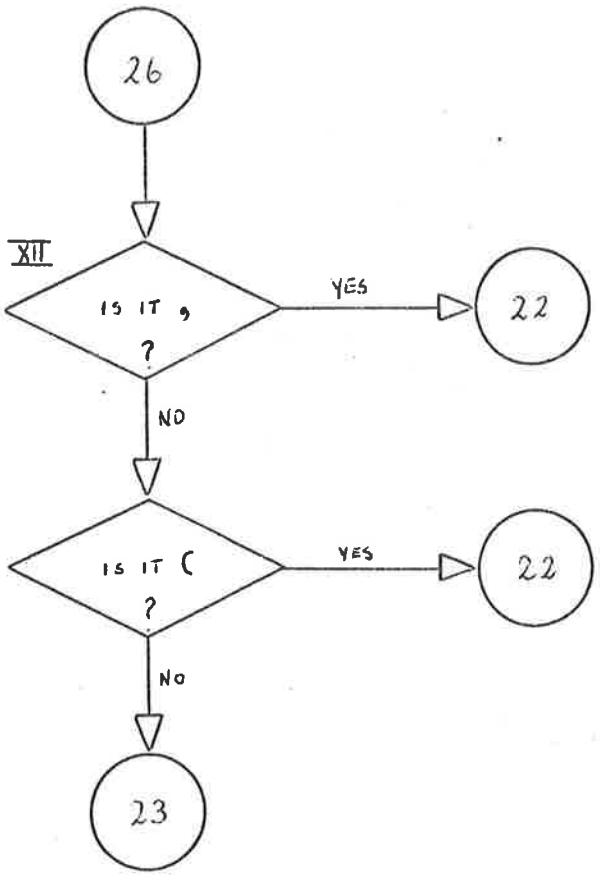

20
VALID VARIABLE ?
NO
16
YES
IS NEXT CHARACTER ? '
NO
16
YES
VALID VARIABLE ?
NO
16
YES
IS NEXT CHARACTER ?
NO
16
YES
STRING TERMINATOR ?
NO
16
YES
1

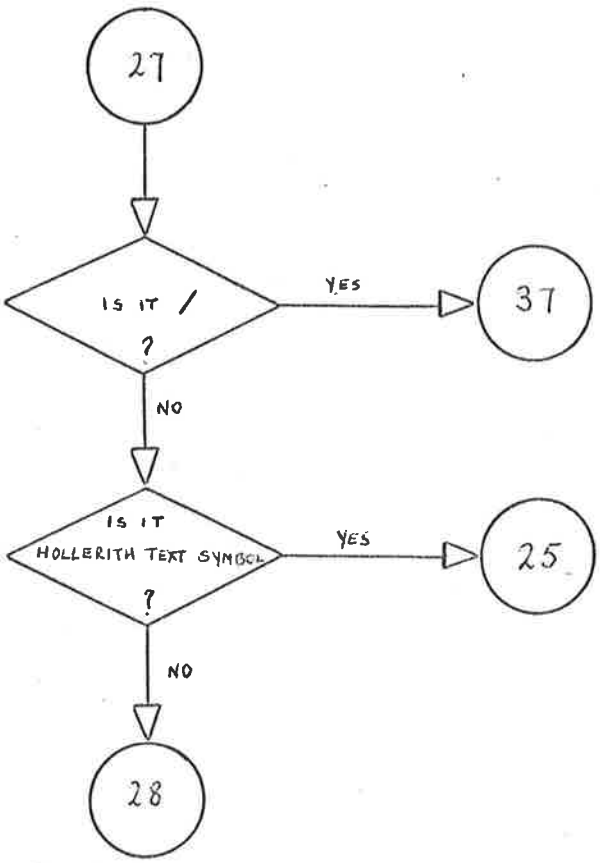

X
SPECIAL IF STATEMENTS
CHECK VALIDITY OF LOGICAL UNIT NUMBER
IS NEXT CHARACTER ?
NO
FLAG ERROR 45
1
YES
21

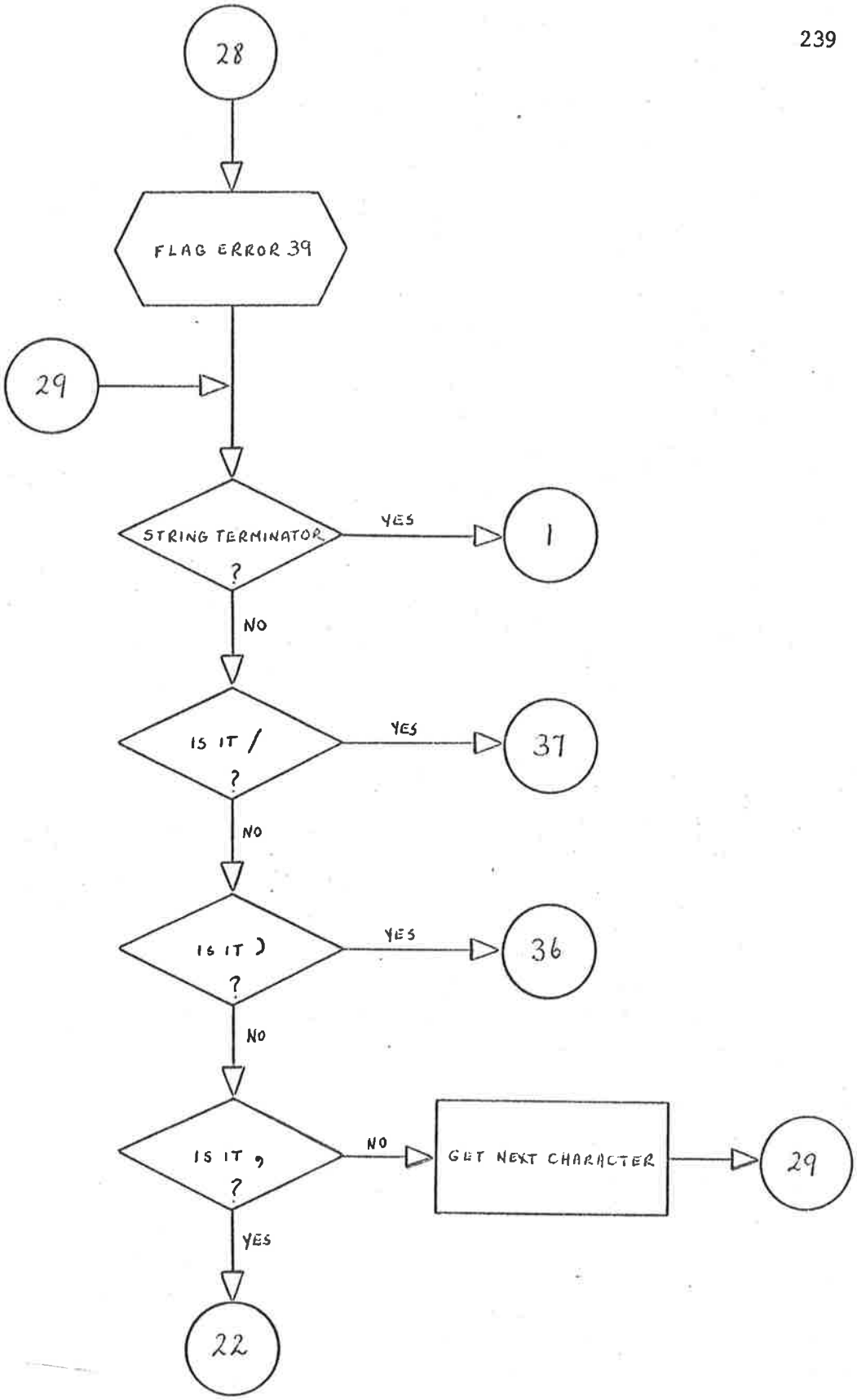

XI
IF STATEMENT NUMBERS
21
PROCESS GROUP OF STATEMENT NUMBERS
STRING TERMINATOR ?
NO
FLAG ERROR 45
1
YES
1

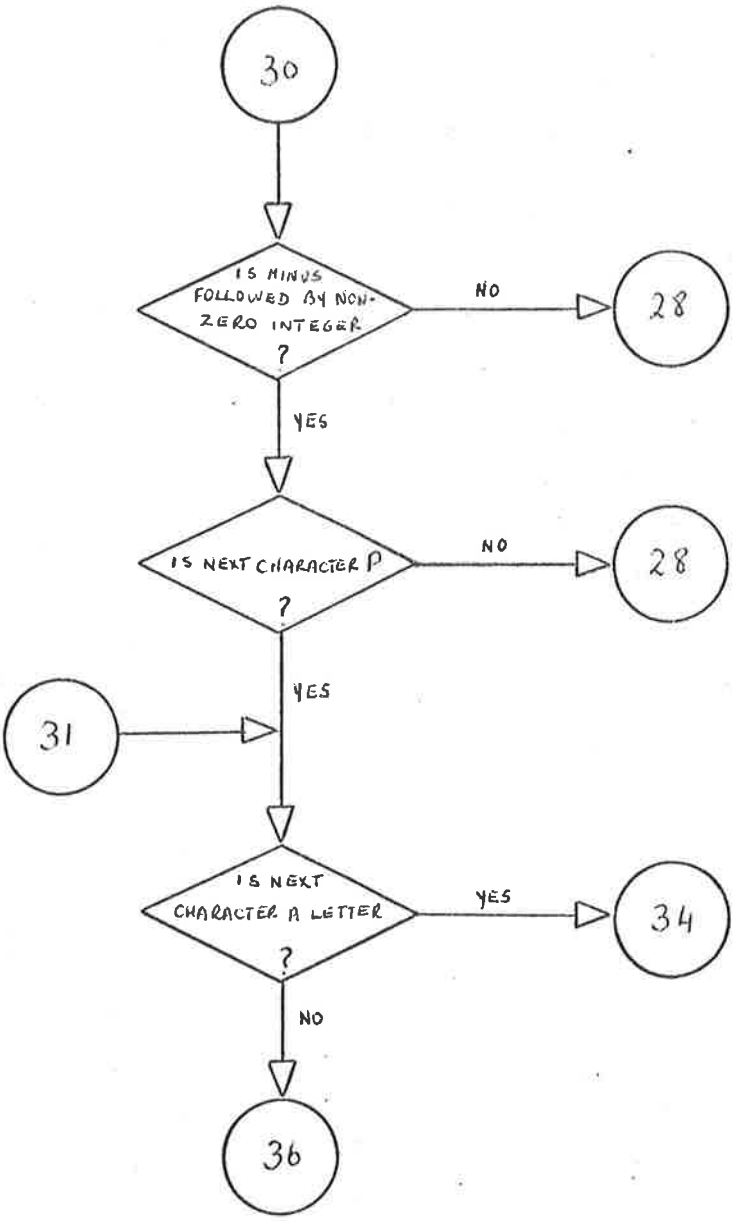

FORMAT STATEMENT
22
IS NEXT CHARACTER ? C
YES
NO
23
IS IT A LETTER ?
YES
33
NO
IS IT A DIGIT ?
YES
EXTRACT INTEGER
NO
IS NEXT CHARACTER ? P
NO
YES
32
31
NO
IS IT A MINUS ?
YES
30
NO
24

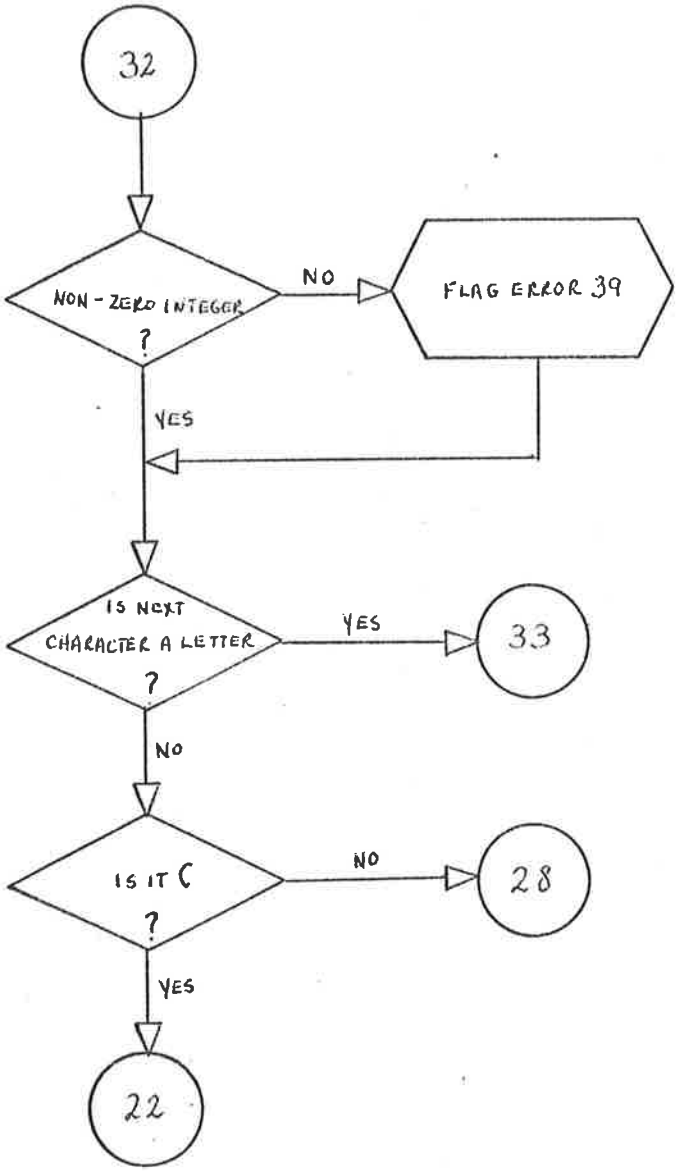

24
START OF LITERAL ?
NO
27
YES
SCAN FOR END OF LITERAL
FOUND BEFORE STRING TERMINATOR ?
NO
FLAG ERROR 39
1
YES
25
IS NEXT CHARACTER ' ?
YES
37
NO
IS IT) ?
YES
38
NO
26

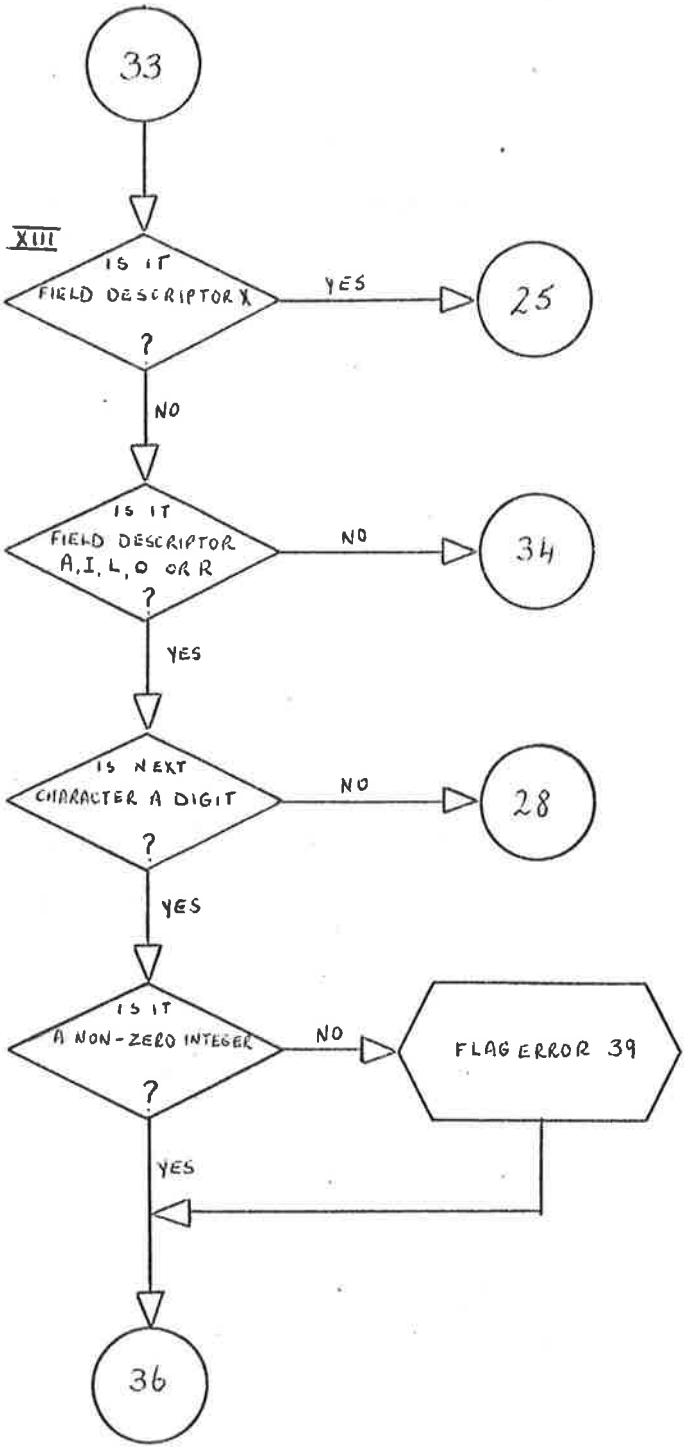

26
XII
IS IT ,
?
YES
22
NO
IS IT C
?
YES
22
NO
23

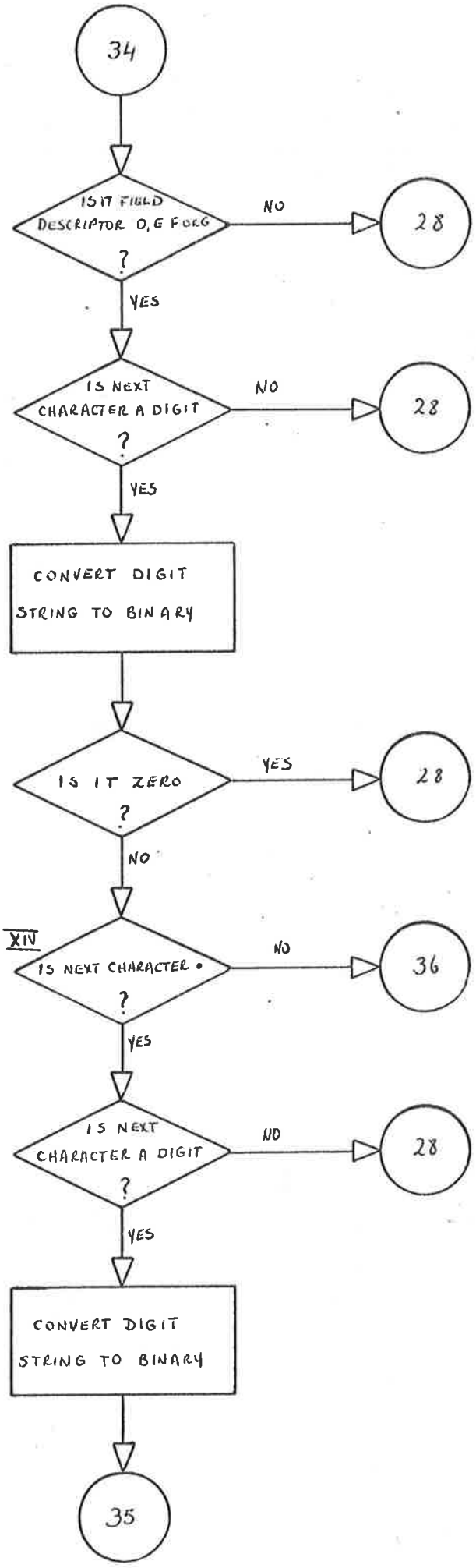

27
IS IT /
?
YES
37
NO
IS IT
HOLLERITH TEXT SYMBOL
?
YES
25
NO
28

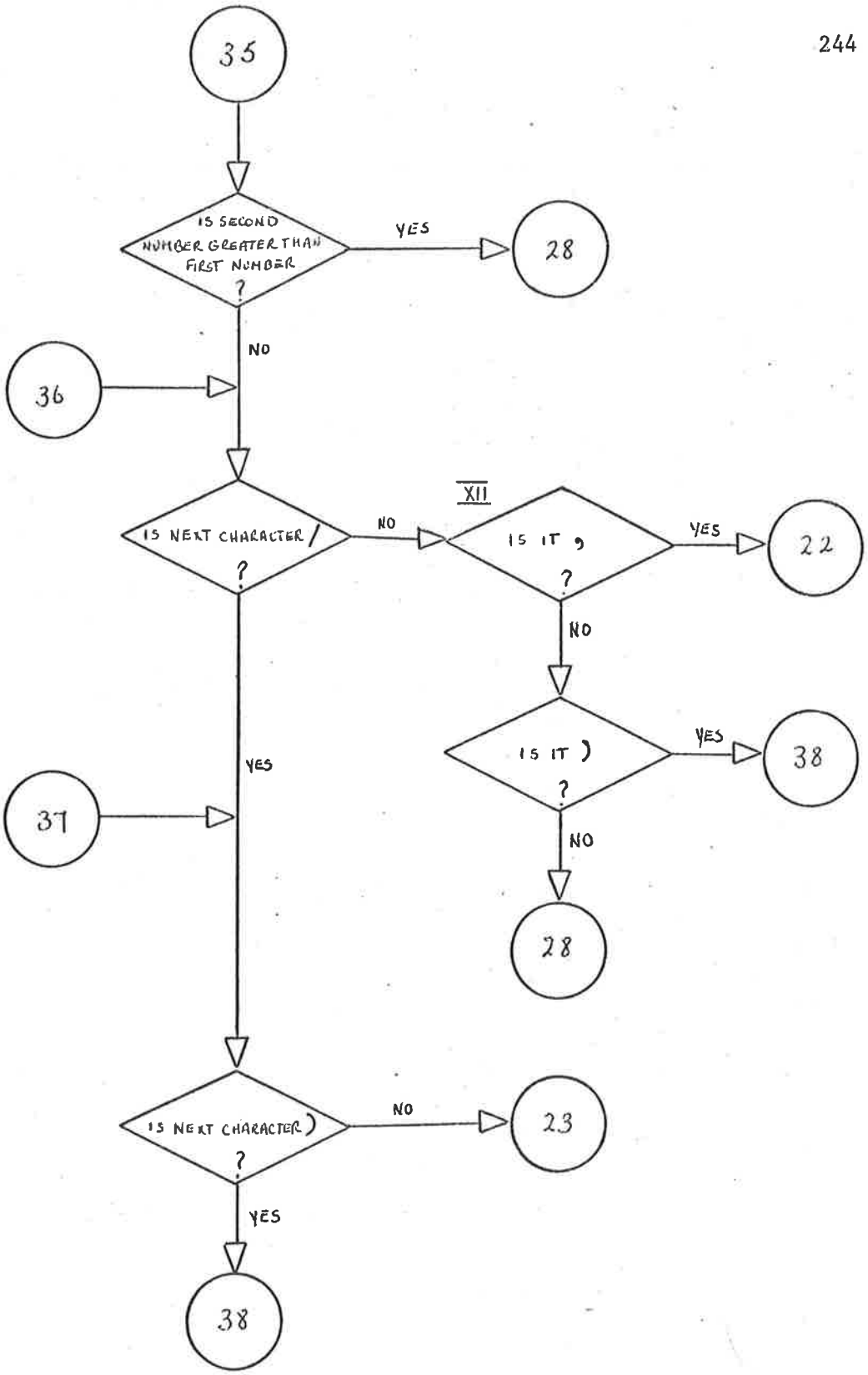

28
FLAG ERROR 39
29
STRING TERMINATOR ?
YES
1
NO
IS IT / ?
YES
37
NO
IS IT) ?
YES
36
NO
IS IT , ?
NO
GET NEXT CHARACTER
29
YES
22

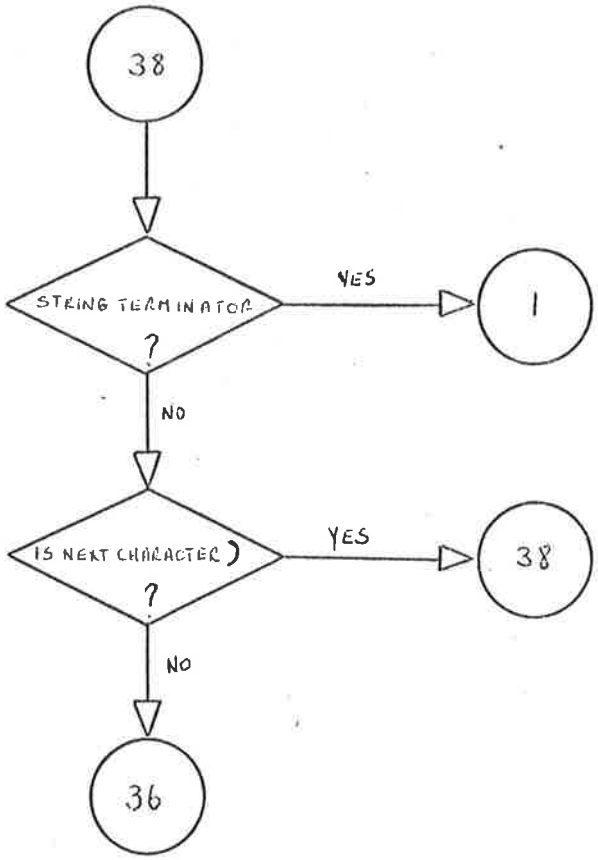

30
IS MINUS FOLLOWED BY NON-ZERO INTEGER ?
NO
28
YES
IS NEXT CHARACTER P ?
NO
28
YES
31
IS NEXT CHARACTER A LETTER ?
YES
34
NO
36

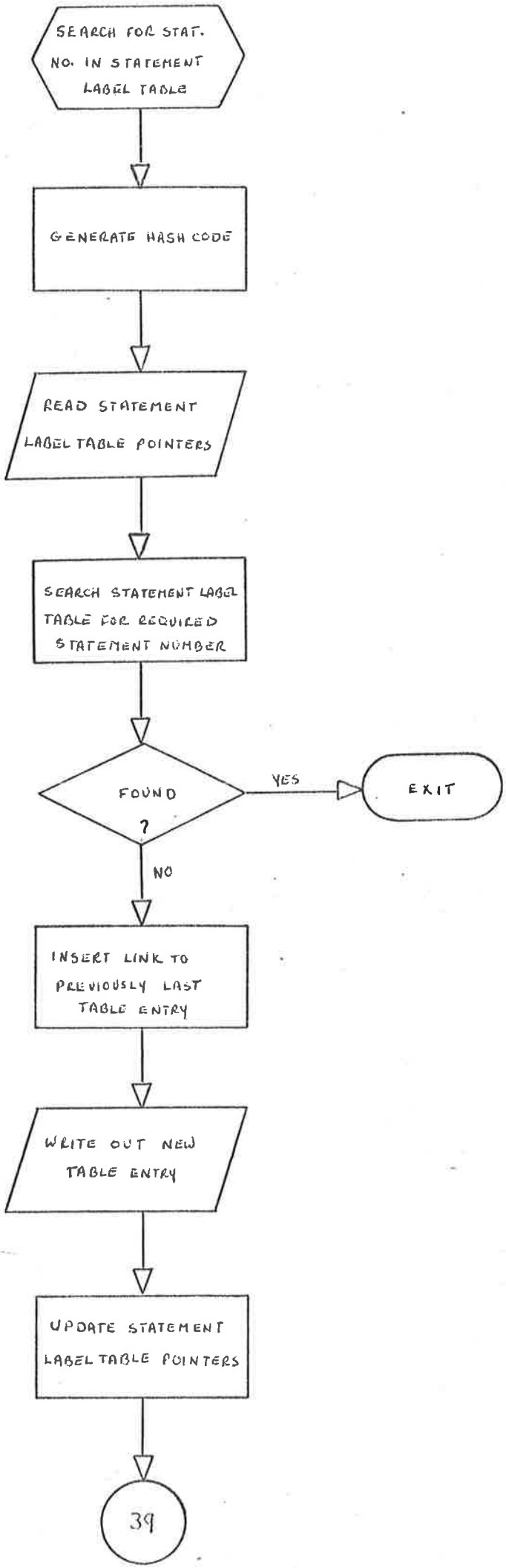

32
NON-ZERO INTEGER ?
NO
FLAG ERROR 39
YES
IS NEXT CHARACTER A LETTER ?
YES
33
NO
IS IT C ?
NO
28
YES
22

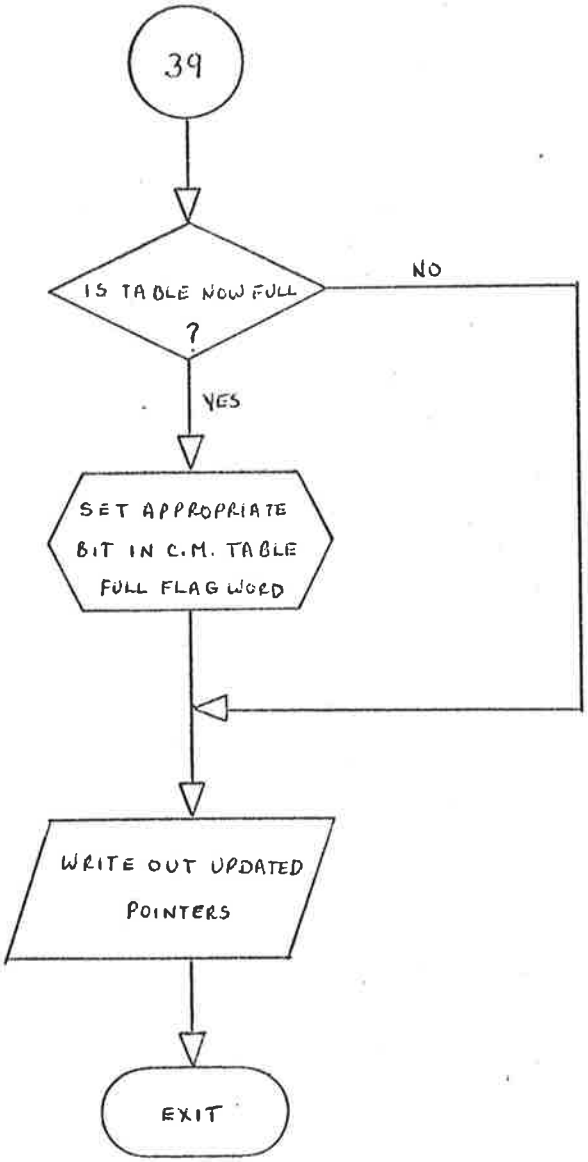

33
XIII
IS IT FIELD DESCRIPTOR X ?
YES
25
NO
IS IT FIELD DESCRIPTOR A, I, L, O OR R ?
NO
34
YES
IS NEXT CHARACTER A DIGIT ?
NO
28
YES
IS IT A NON-ZERO INTEGER ?
NO
FLAG ERROR 39
YES
36

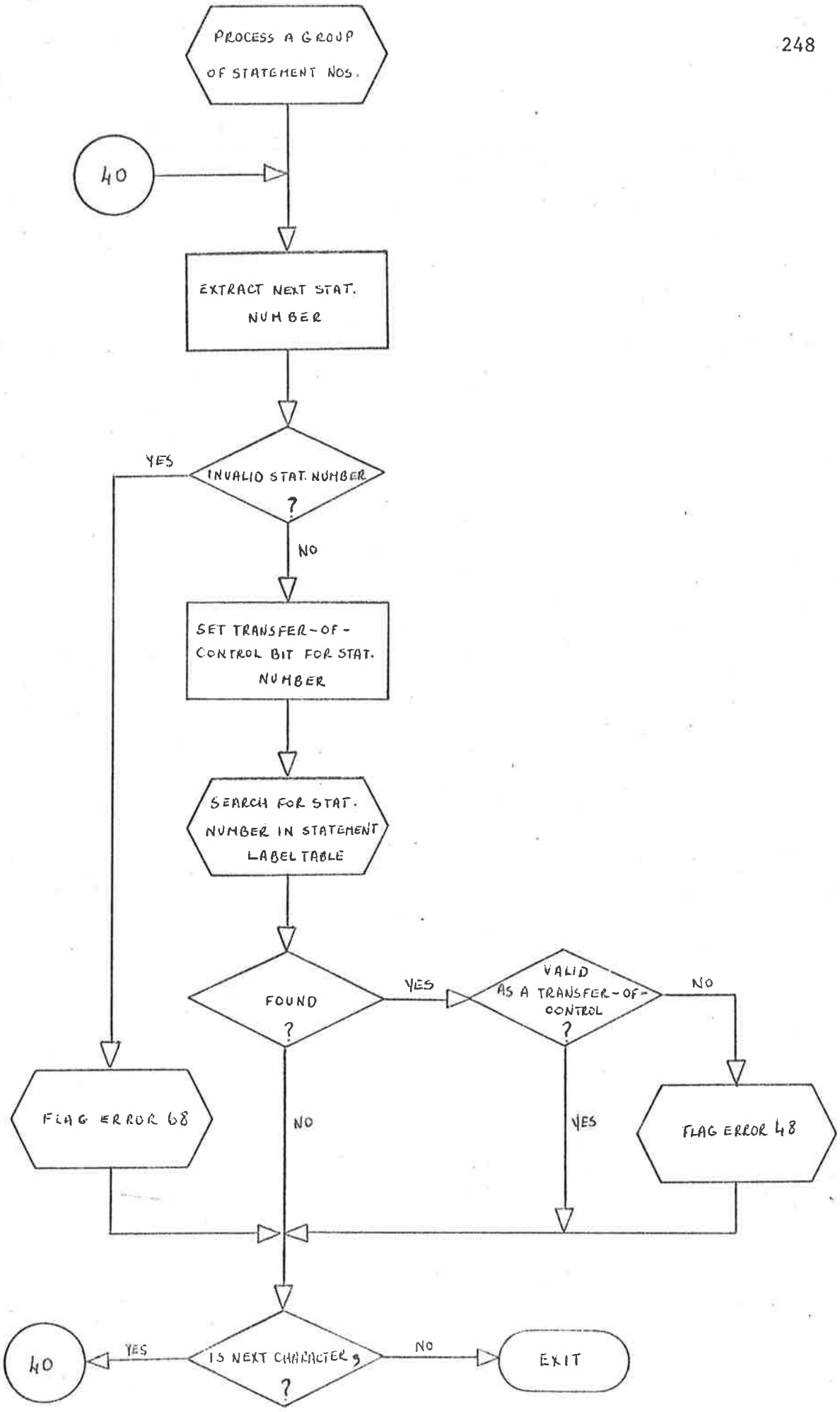

34
IS IT FIELD DESCRIPTOR D,E FORG ?
NO
28
YES
IS NEXT CHARACTER A DIGIT ?
NO
28
YES
CONVERT DIGIT STRING TO BINARY
IS IT ZERO ?
YES
28
NO
XIV
IS NEXT CHARACTER . ?
NO
36
YES
IS NEXT CHARACTER A DIGIT ?
NO
28
YES
CONVERT DIGIT STRING TO BINARY
35

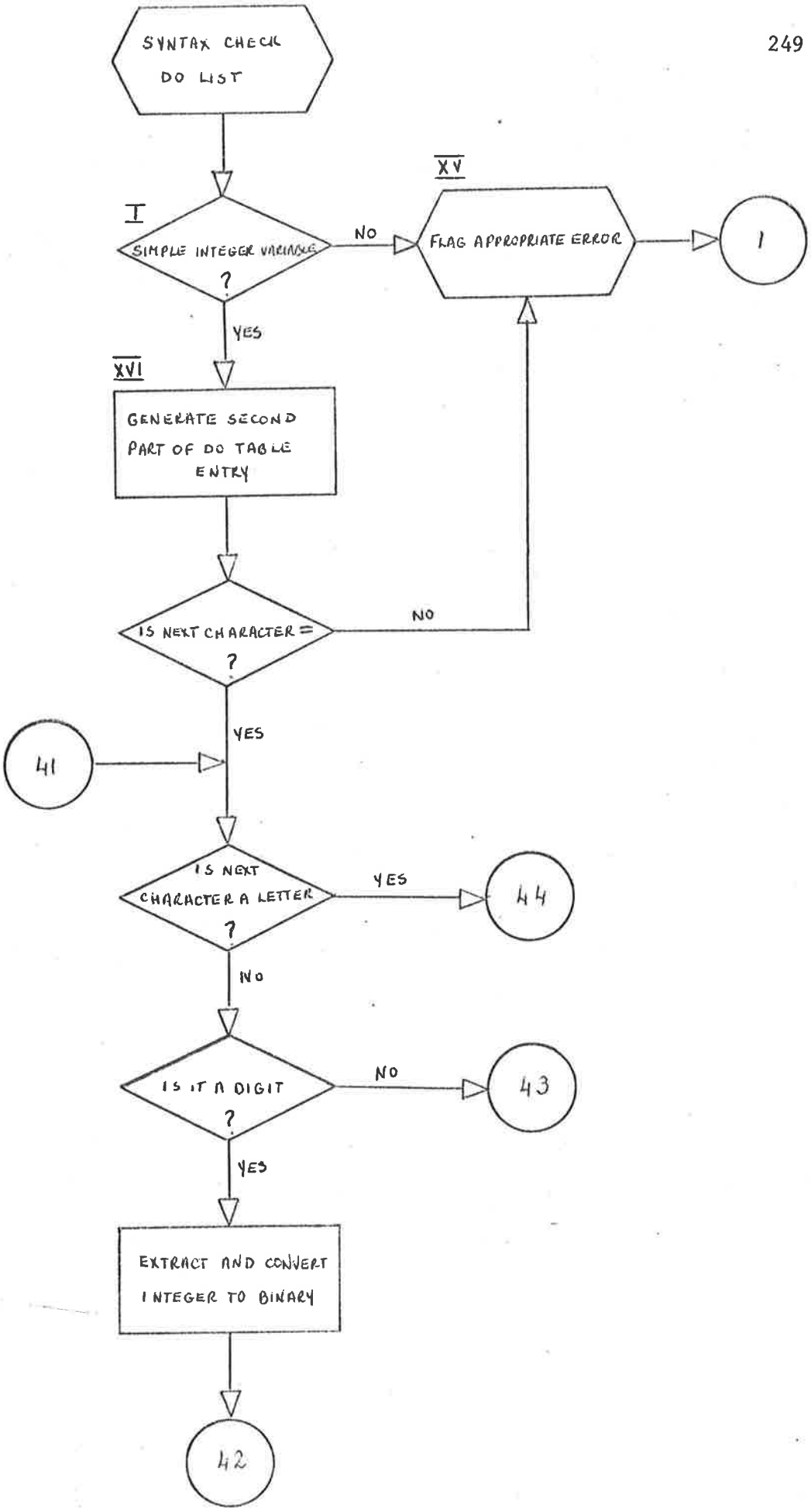

35
IS SECOND NUMBER GREATER THAN FIRST NUMBER ?
YES
28
NO
36
IS NEXT CHARACTER / ?
NO
XII
IS IT , ?
YES
22
NO
IS IT) ?
YES
38
NO
28
YES
37
IS NEXT CHARACTER) ?
NO
23
YES
38

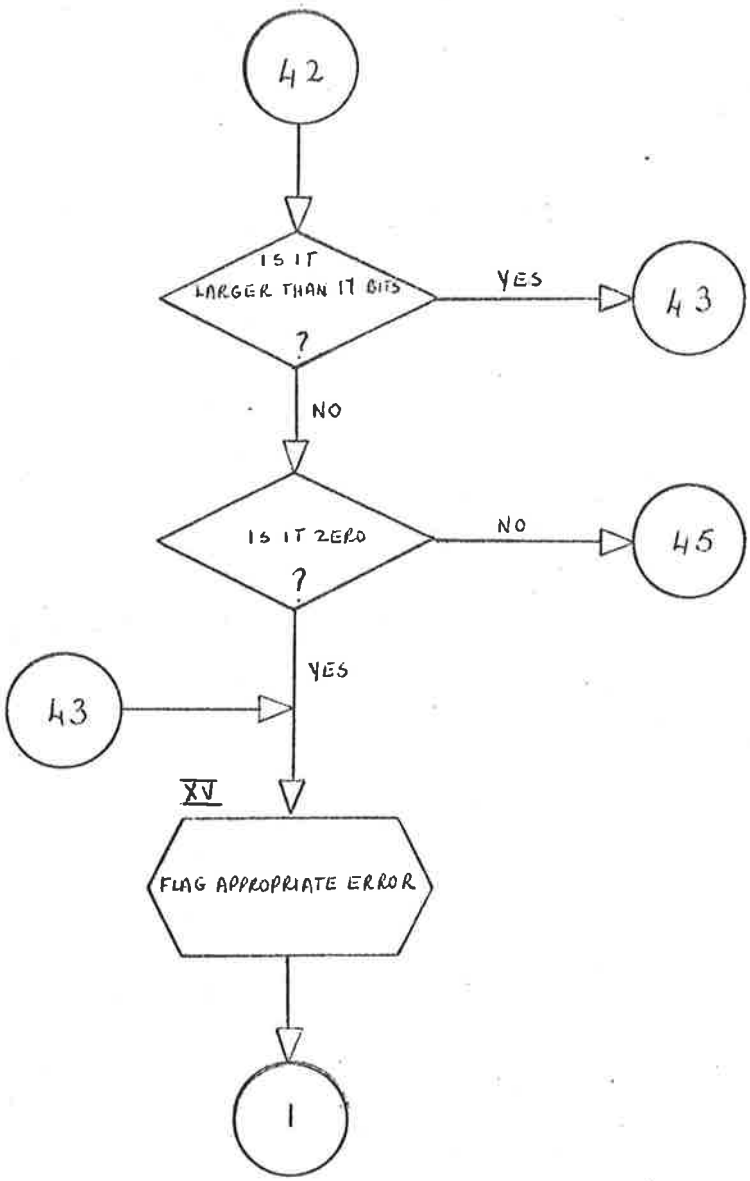

38
STRING TERMINATOR ?
YES
1
NO
IS NEXT CHARACTER) ?
YES
38
NO
36

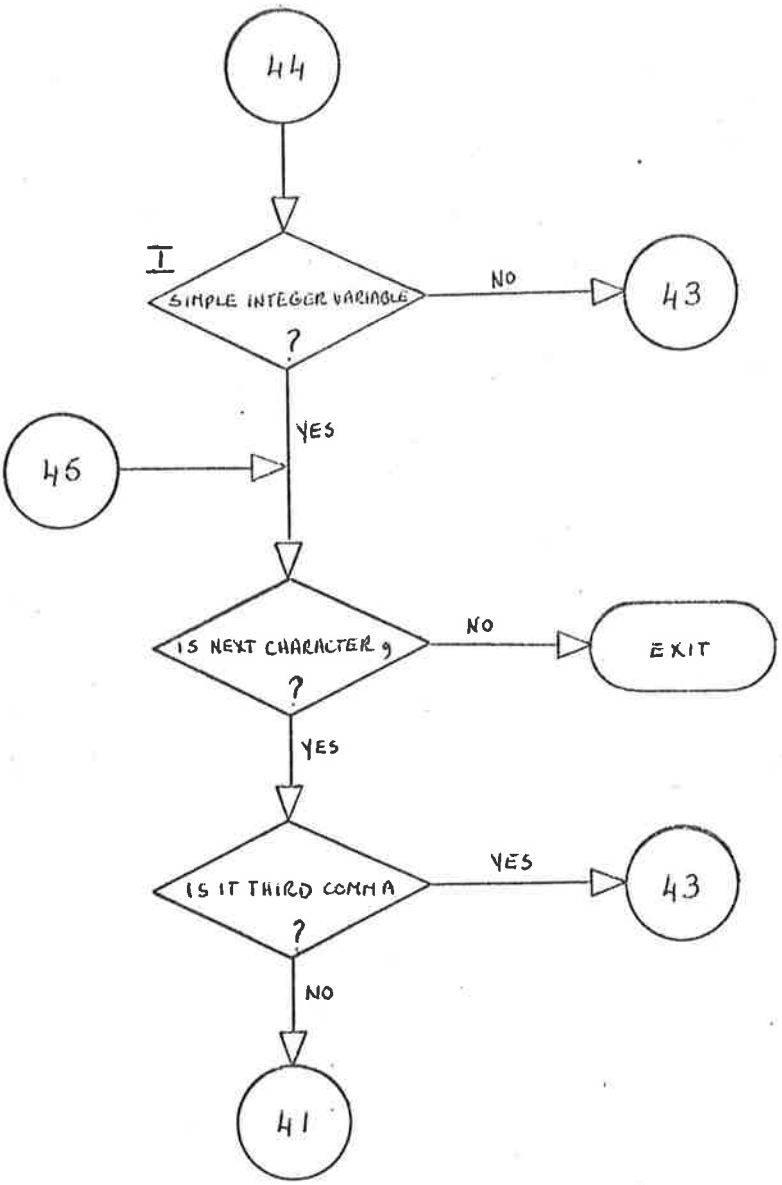

SEARCH FOR STAT. NO. IN STATEMENT LABEL TABLE
GENERATE HASH CODE
READ STATEMENT LABEL TABLE POINTERS
SEARCH STATEMENT LABEL TABLE FOR REQUIRED STATEMENT NUMBER
FOUND ?
YES
EXIT
NO
INSERT LINK TO PREVIOUSLY LAST TABLE ENTRY
WRITE OUT NEW TABLE ENTRY
UPDATE STATEMENT LABEL TABLE POINTERS
39

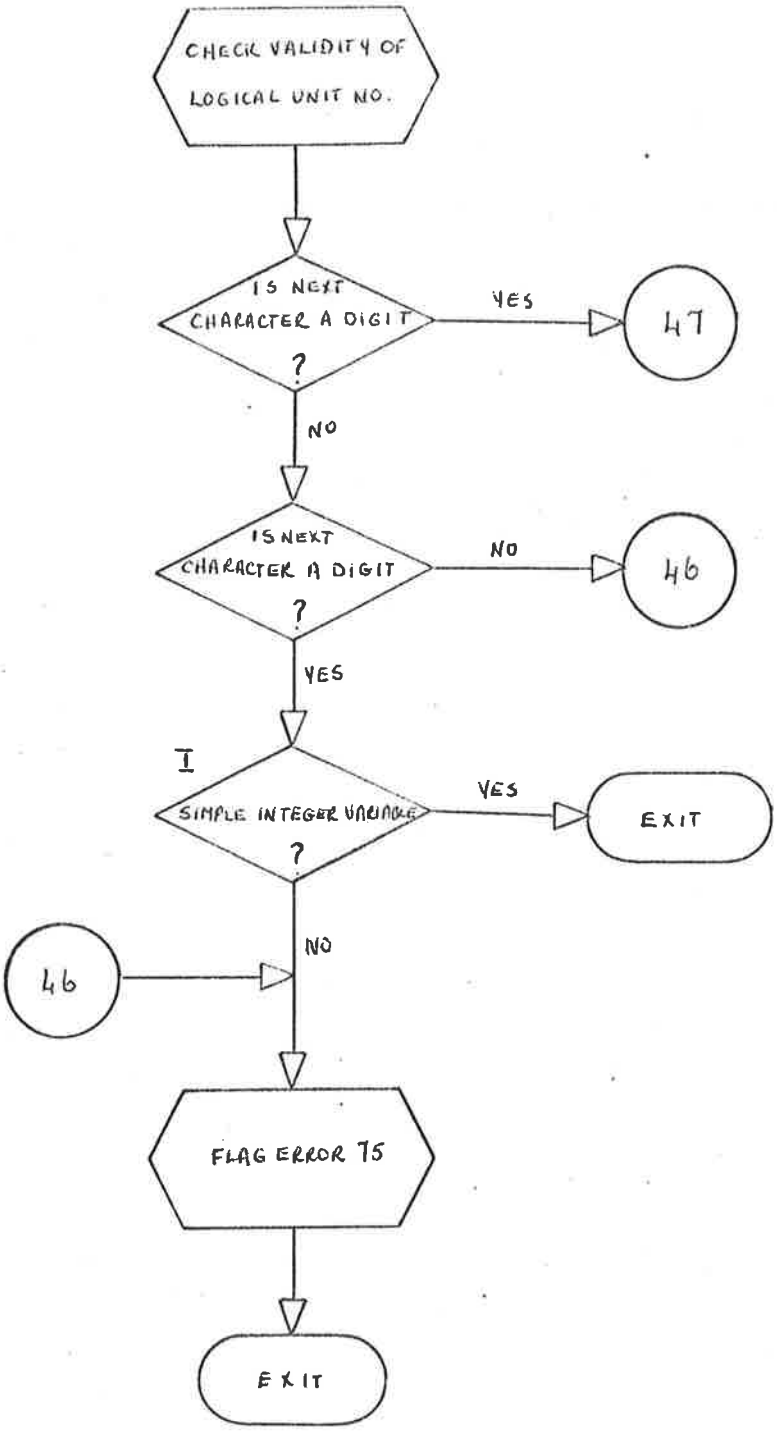

39
IS TABLE NOW FULL ?
NO
YES
SET APPROPRIATE BIT IN C.M. TABLE FULL FLAG WORD
WRITE OUT UPDATED POINTERS
EXIT

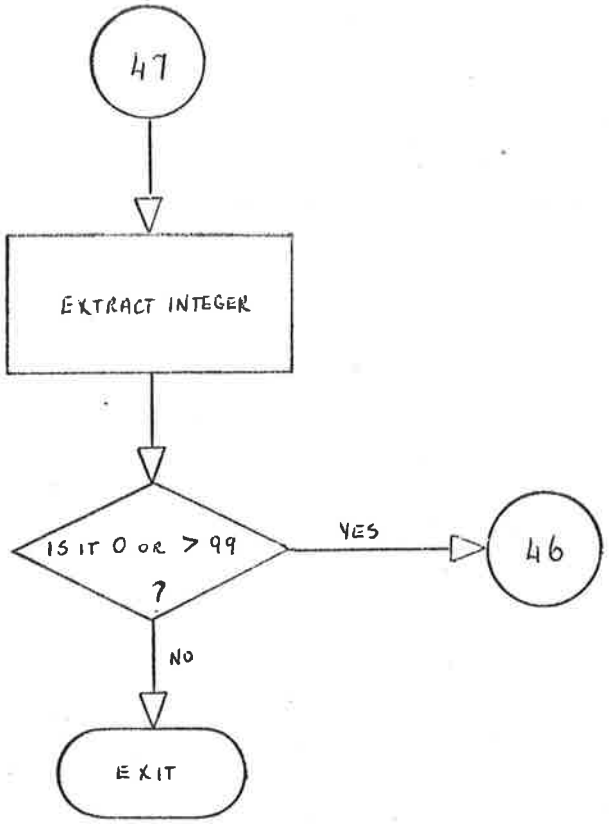

PROCESS A GROUP OF STATEMENT NOS.
40
EXTRACT NEXT STAT. NUMBER
INVALID STAT. NUMBER ?
YES
NO
SET TRANSFER-OF-CONTROL BIT FOR STAT. NUMBER
SEARCH FOR STAT. NUMBER IN STATEMENT LABEL TABLE
FOUND ?
YES
VALID AS A TRANSFER-OF-CONTROL ?
NO
FLAG ERROR 68
NO
YES
FLAG ERROR 48
40
YES
IS NEXT CHARACTER ; ?
NO
EXIT

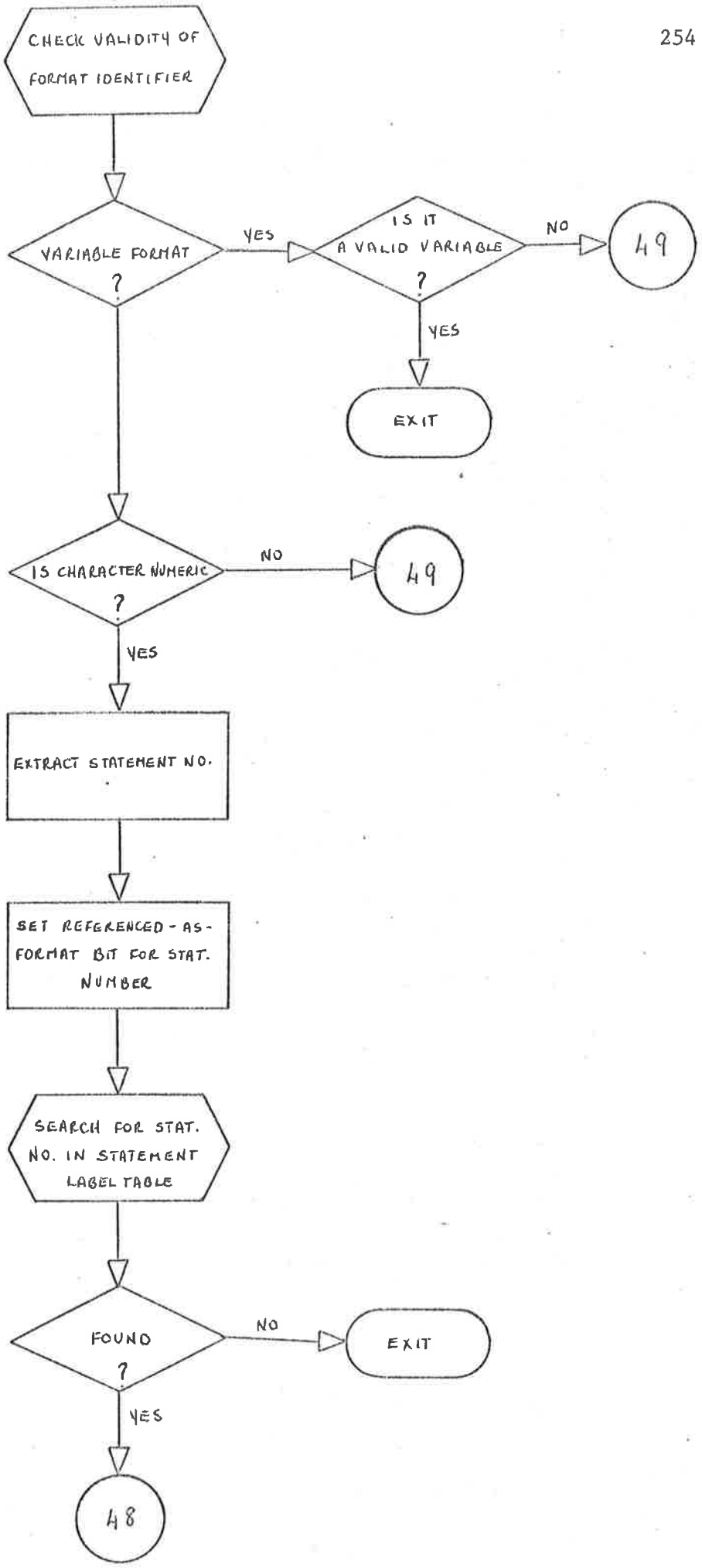

SYNTAX CHECK
DO LIST
I
SIMPLE INTEGER VARIABLE
?
NO
XV
FLAG APPROPRIATE ERROR
1
YES
XVI
GENERATE SECOND
PART OF DO TABLE
ENTRY
IS NEXT CHARACTER =
?
NO
YES
41
IS NEXT
CHARACTER A LETTER
?
YES
44
NO
IS IT A DIGIT
?
NO
43
YES
EXTRACT AND CONVERT
INTEGER TO BINARY
42

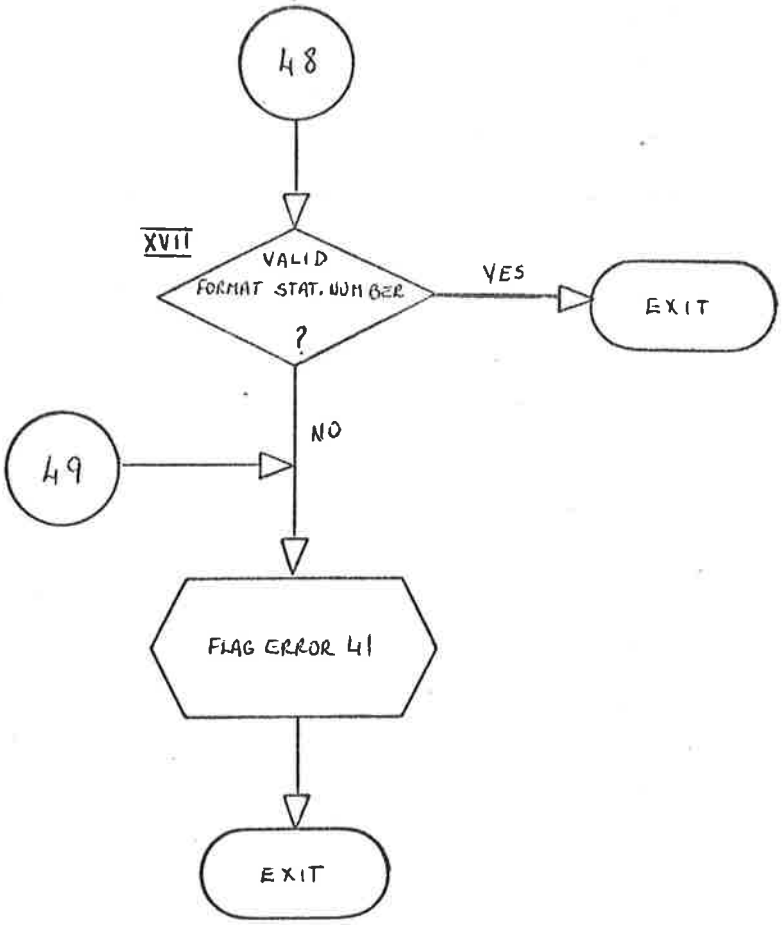

42
IS IT LARGER THAN 17 BITS ?
YES
43
NO
IS IT ZERO ?
NO
45
YES
43
XV
FLAG APPROPRIATE ERROR
1

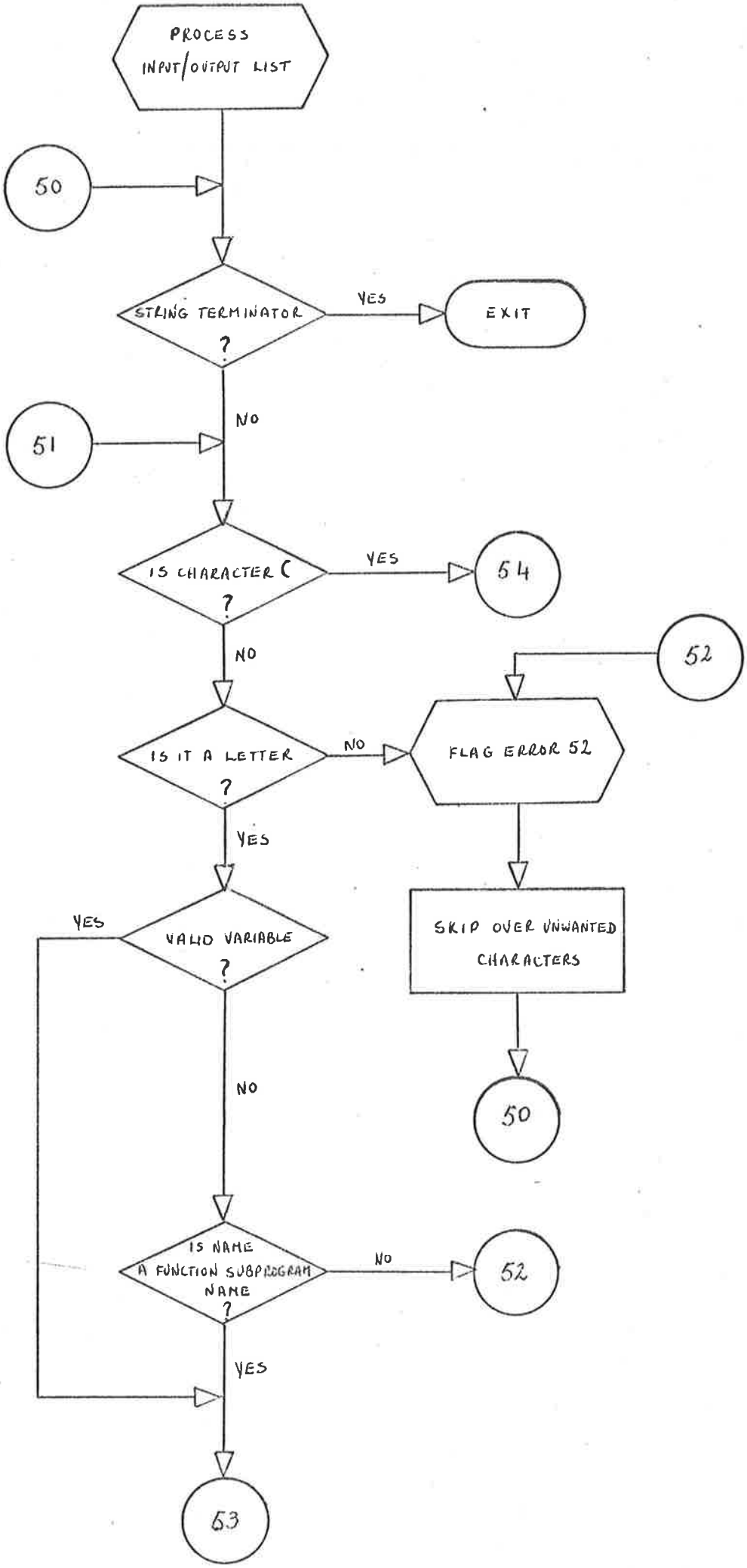

44
I
SIMPLE INTEGER VARIABLE ?
NO
43
YES
46
IS NEXT CHARACTER , ?
NO
EXIT
YES
IS IT THIRD COMMA ?
YES
43
NO
41

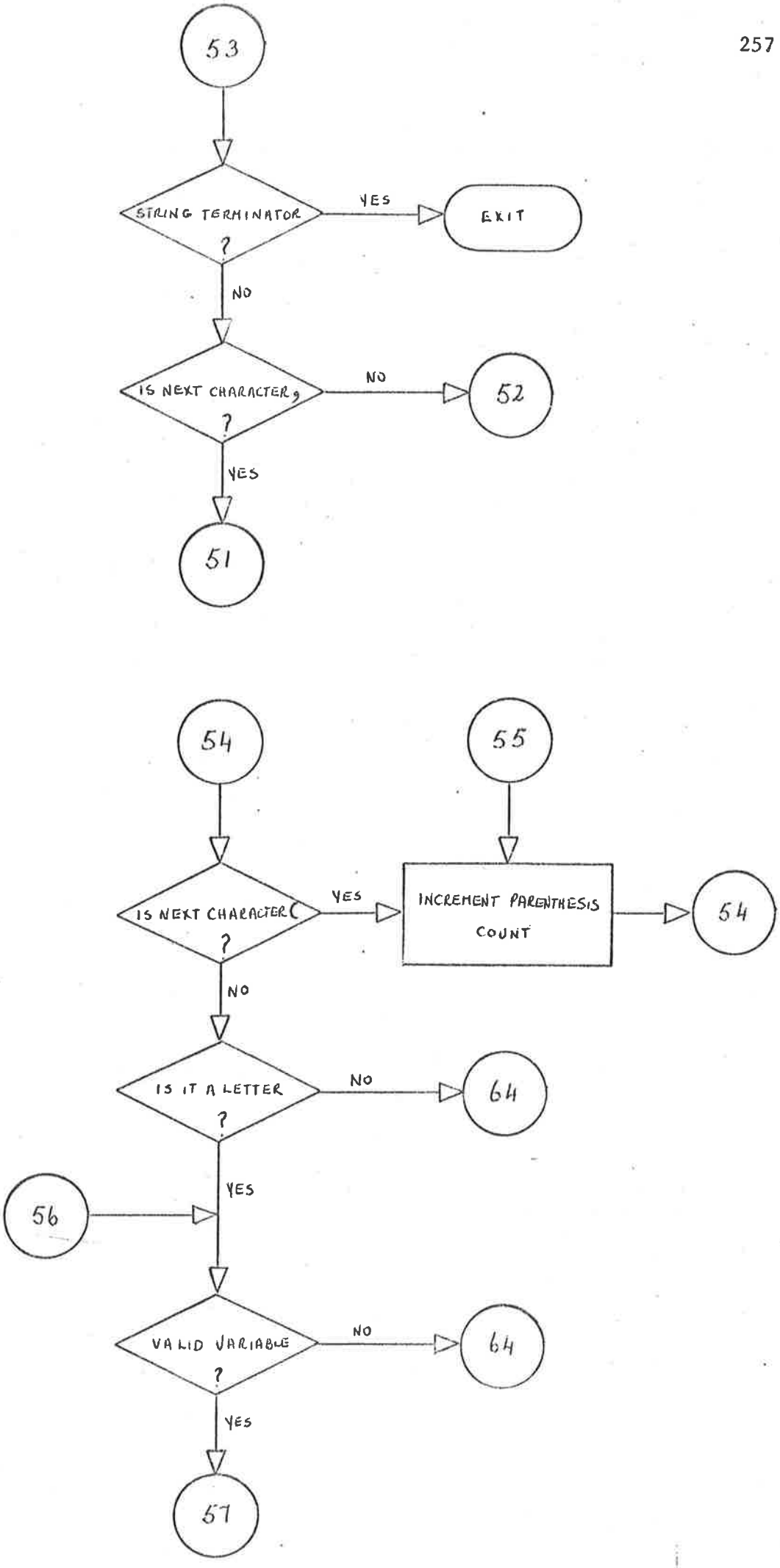

CHECK VALIDITY OF LOGICAL UNIT NO.
IS NEXT CHARACTER A DIGIT ?
YES
47
NO
IS NEXT CHARACTER A DIGIT ?
NO
46
YES
I
SIMPLE INTEGER VARIABLE ?
YES
EXIT
NO
46
FLAG ERROR 75
EXIT

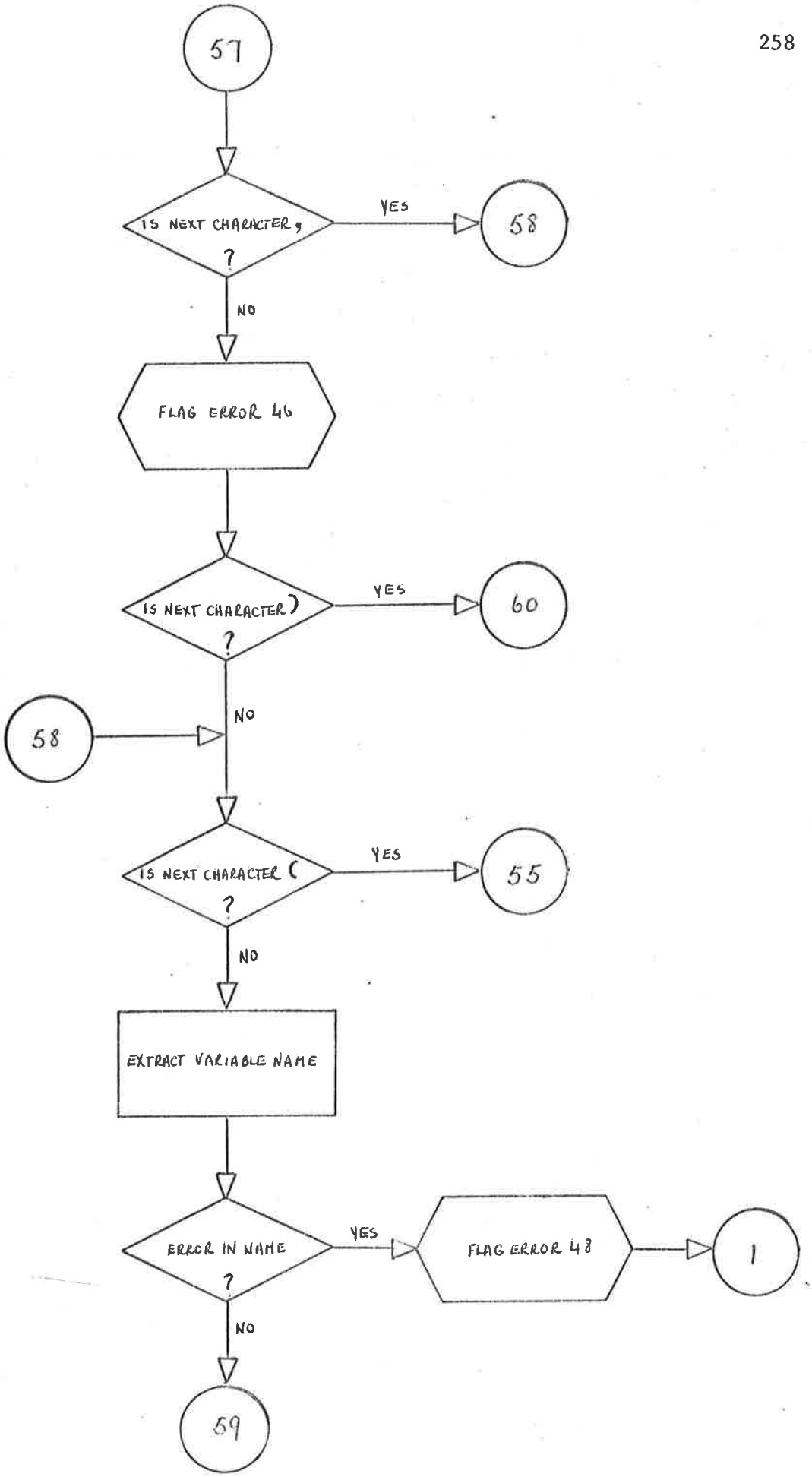

47
EXTRACT INTEGER
IS IT 0 OR > 99 ?
YES
46
NO
EXIT

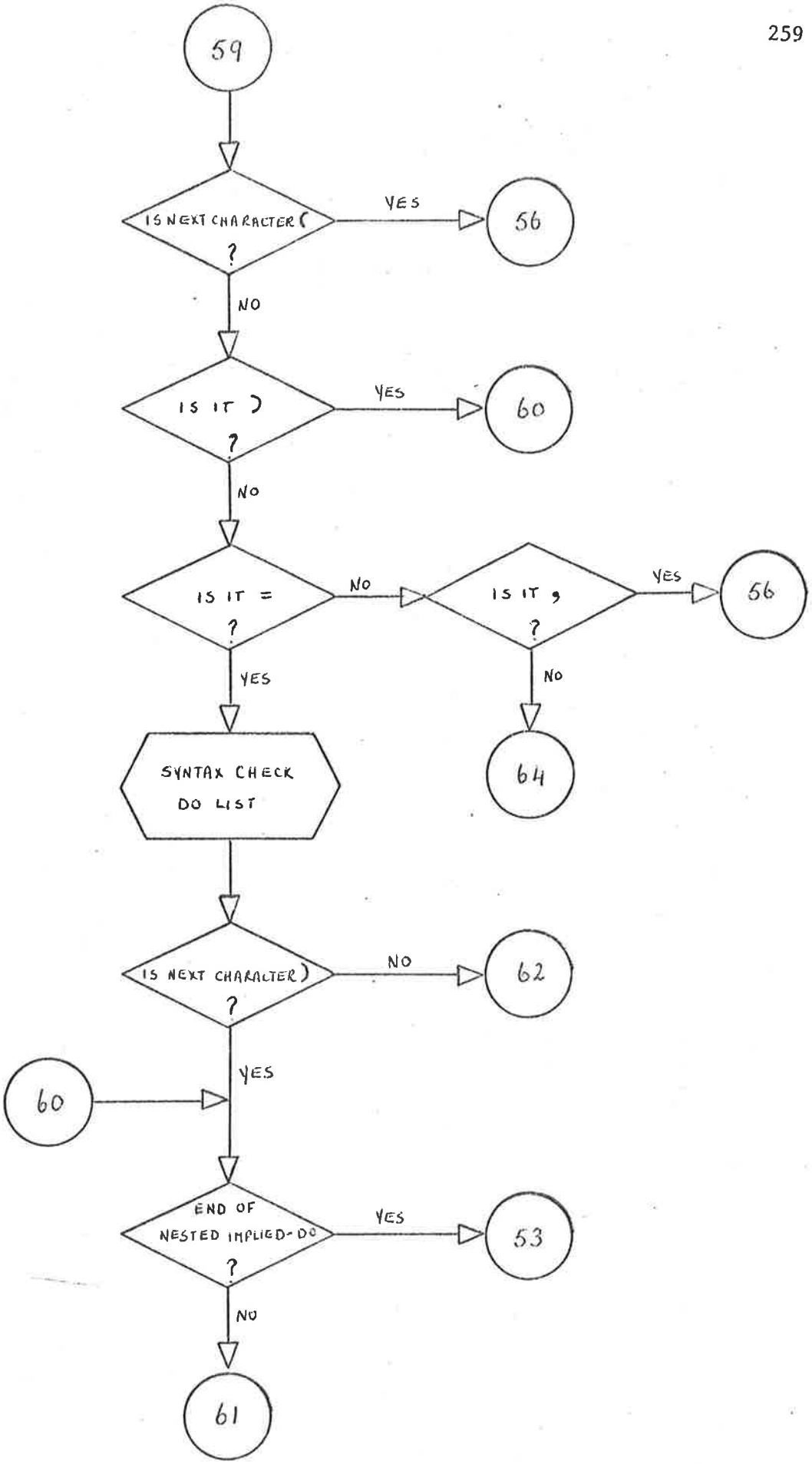

CHECK VALIDITY OF FORMAT IDENTIFIER
VARIABLE FORMAT ?
YES
IS IT A VALID VARIABLE ?
NO
49
YES
EXIT
IS CHARACTER NUMERIC ?
NO
49
YES
EXTRACT STATEMENT NO.
SET REFERENCED-AS-FORMAT BIT FOR STAT. NUMBER
SEARCH FOR STAT. NO. IN STATEMENT LABEL TABLE
FOUND ?
NO
EXIT
YES
48

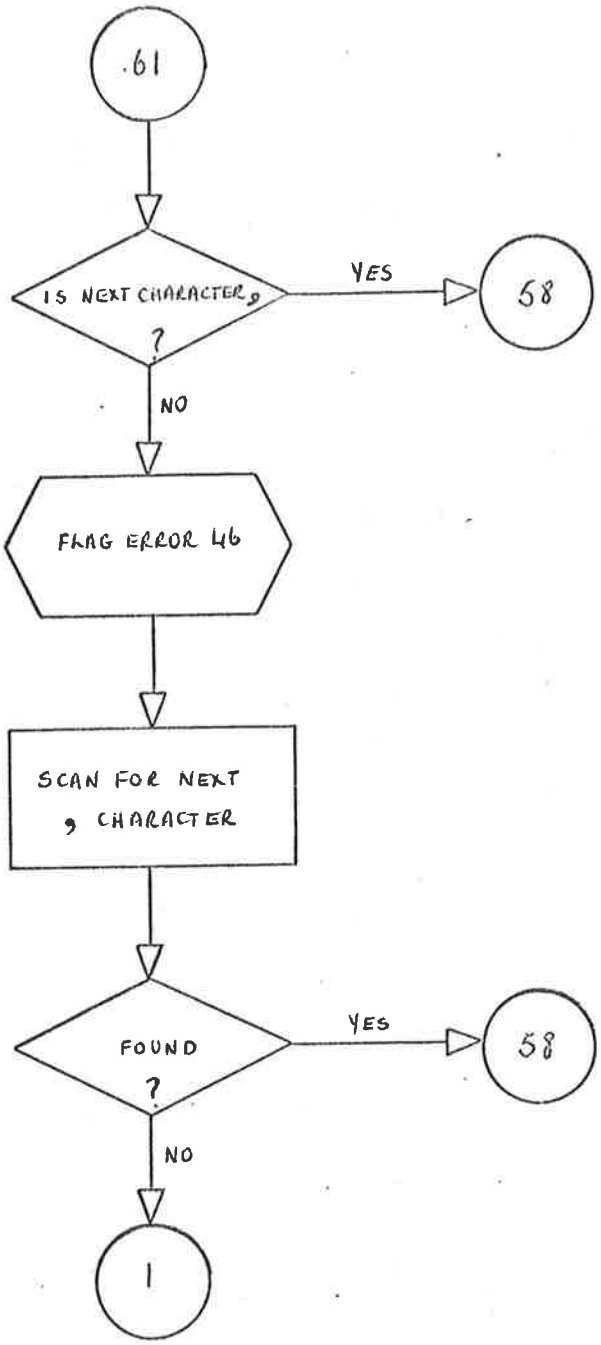

48
XVII
VALID FORMAT STAT. NUMBER ?
YES
EXIT
49
NO
FLAG ERROR 41
EXIT

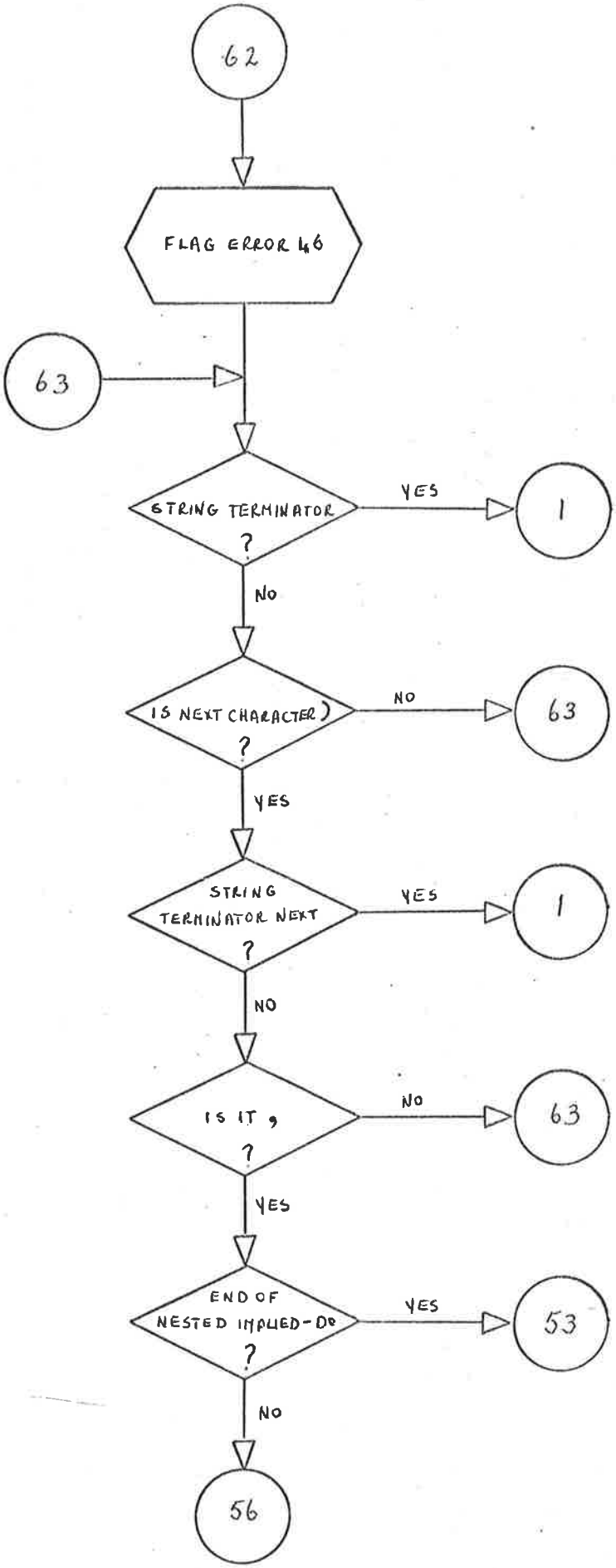

PROCESS INPUT/OUTPUT LIST
50
STRING TERMINATOR ?
YES
EXIT
NO
51
IS CHARACTER (?
YES
54
NO
52
IS IT A LETTER ?
NO
FLAG ERROR 52
YES
VALID VARIABLE ?
YES
SKIP OVER UNWANTED CHARACTERS
NO
50
IS NAME A FUNCTION SUBPROGRAM NAME ?
NO
52
YES
53

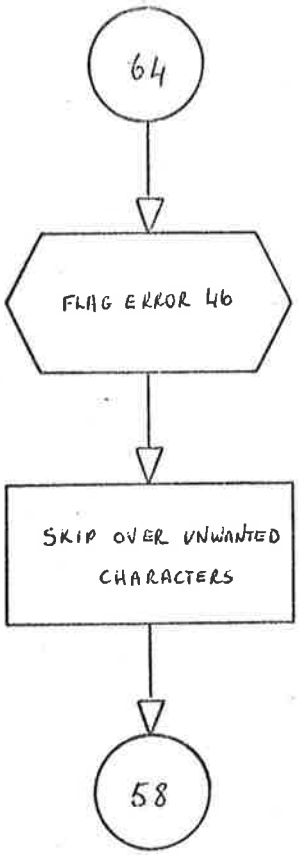

53
STRING TERMINATOR ?
YES
EXIT
NO
IS NEXT CHARACTER (?
NO
52
YES
51
54
55
IS NEXT CHARACTER (?
YES
INCREMENT PARENTHESIS COUNT
54
NO
IS IT A LETTER ?
NO
64
YES
56
VALID VARIABLE ?
NO
64
YES
57

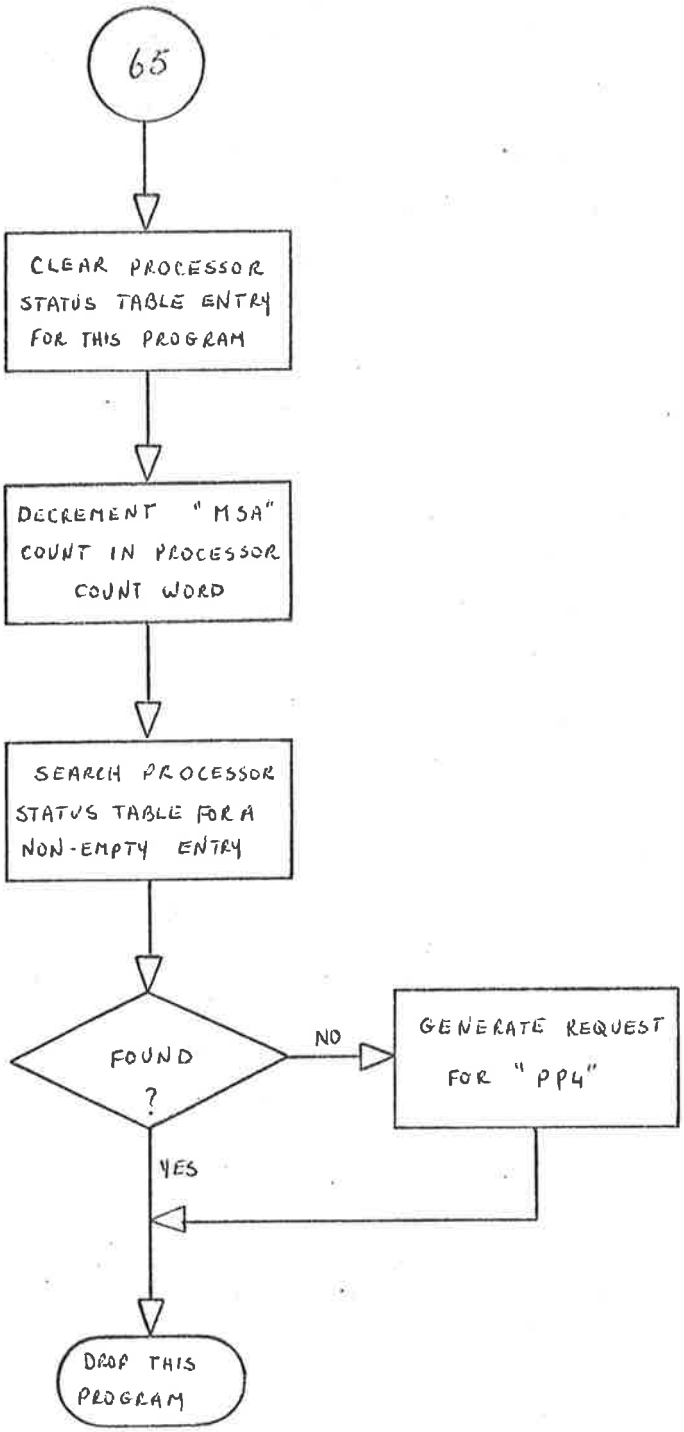

57
IS NEXT CHARACTER , ?
YES
58
NO
FLAG ERROR 46
IS NEXT CHARACTER) ?
YES
60
58
NO
IS NEXT CHARACTER (?
YES
55
NO
EXTRACT VARIABLE NAME
ERROR IN NAME ?
YES
FLAG ERROR 48
1
NO
59


59
IS NEXT CHARACTER (?
YES
56
NO
IS IT) ?
YES
60
NO
IS IT = ?
NO
IS IT , ?
YES
56
YES
NO
SYNTAX CHECK
DO LIST
64
IS NEXT CHARACTER) ?
NO
62
YES
60
END OF
NESTED IMPLIED-DO ?
YES
53
NO
61


61
IS NEXT CHARACTER ,
?
YES
58
NO
FLAG ERROR 46
SCAN FOR NEXT
, CHARACTER
FOUND
?
YES
58
NO
1


62
FLAG ERROR 46
63
STRING TERMINATOR ?
YES
1
No
IS NEXT CHARACTER) ?
NO
63
YES
STRING TERMINATOR NEXT ?
YES
1
NO
IS IT , ?
NO
63
YES
END OF NESTED IMPLIED-DO ?
YES
53
NO
56


64
FLAG ERROR 46
SKIP OVER UNWANTED CHARACTERS
58


65
CLEAR PROCESSOR STATUS TABLE ENTRY FOR THIS PROGRAM
DECREMENT "MSA" COUNT IN PROCESSOR COUNT WORD
SEARCH PROCESSOR STATUS TABLE FOR A NON-EMPTY ENTRY
FOUND ?
NO
GENERATE REQUEST FOR "PP4"
YES
DROP THIS PROGRAM

11. PROGRAM CTM - CENTRAL MEMORY TABLE MOVER

11.0 Introduction

Effectively, each central memory table comprises a main area followed by an overflow area. When a program, attempting to insert an entry in a table, finds that the main area of the table is full, it sets the appropriate bit in the central memory table full flag word (see 5.1.11) and inserts the entry in the usual way into the overflow area of the table. Furthermore, this program and any other program that is currently active, may make further insertions into this table, quite validly, and unaware that the table has previously been flagged as "full". This continues until eventually each of these programs reaches the end of its major loop and finds that the central memory table full flag is set. The first program to do so requests the monitor to load program CTM and then, like all the rest, drops itself with "pause for CTM completion" status in the processor status table.

Since program CTM requires exclusive access to the entire central memory table area, it cannot commence full operation until all programs defined in the processor status table have "pause for CTM completion" status. Therefore, if program CTM finds any program still active, it requests monitor to reload this program, that is program CTM, after a suitable delay to ensure that all currently active program have ceased execution, and then drops itself. Nevertheless, when the program is finally fully active, it calculates

the central memory storage increment required and requests monitor to allocate it. When monitor replies that the request has been satisfied in full, the program moves each table in turn, starting at the top with the statement label table. When this operation is complete, it requests monitor to load once again each of the programs defined in the processor status table. After the last of these programs has been loaded, the system is again fully operational and without apparent loss of continuity. Program CTM then drops itself.

When the windup routine, program PP4, detects that the INPUT buffer contains another source subprogram, it re-initializes all central memory tables, except the COMMON block name table (see 5.1.13), to the empty state and their initial size, and requests monitor to reduce central memory storage to its original size.

11.1 Explanatory Notes to the Flow-chart

Although not explicitly stated in the accompanying flow-chart, all references to the processor status table are made only after access to the table has been successfully requested, using the procedure described in 5.1.0. However, references to other tables are not made via this table accessing mechanism. This is because this program drops itself on entry if a scan through the processor status table shows that there is at least one program which does not have "pause for CTM completion" status.

The following notes refer to the flow-chart:

I. See above comments.

II. Before dropping itself, the program requests monitor to reload program CTM after a suitable delay. This is to allow further time for those programs which had not yet detected that the central memory table full flag is set, to do so.

III. The increment size for each table, when it is flagged as "full", is given by the leftmost bytes of the central memory table full flag word. However, for those tables which are not flagged as "full", the increment size is reduced to a quarter of this quantity. This is done to decrease the probability that any of these tables will become "full" in the near future and yet not allow the total central memory table area to become unnecessarily large.

IV. Monitor allocates central memory in sixty-four word increments.

V. If monitor replies that it cannot extend the central memory by the amount requested, the program issues the message

STORE LIMIT REACHED

to the operator's console and hence the job's DAYFILE (see 3.0) and then requests monitor to abort the job, since it is generally not meaningful to continue if the tables cannot be extended by the required increment size. This situation could only arise if the mix

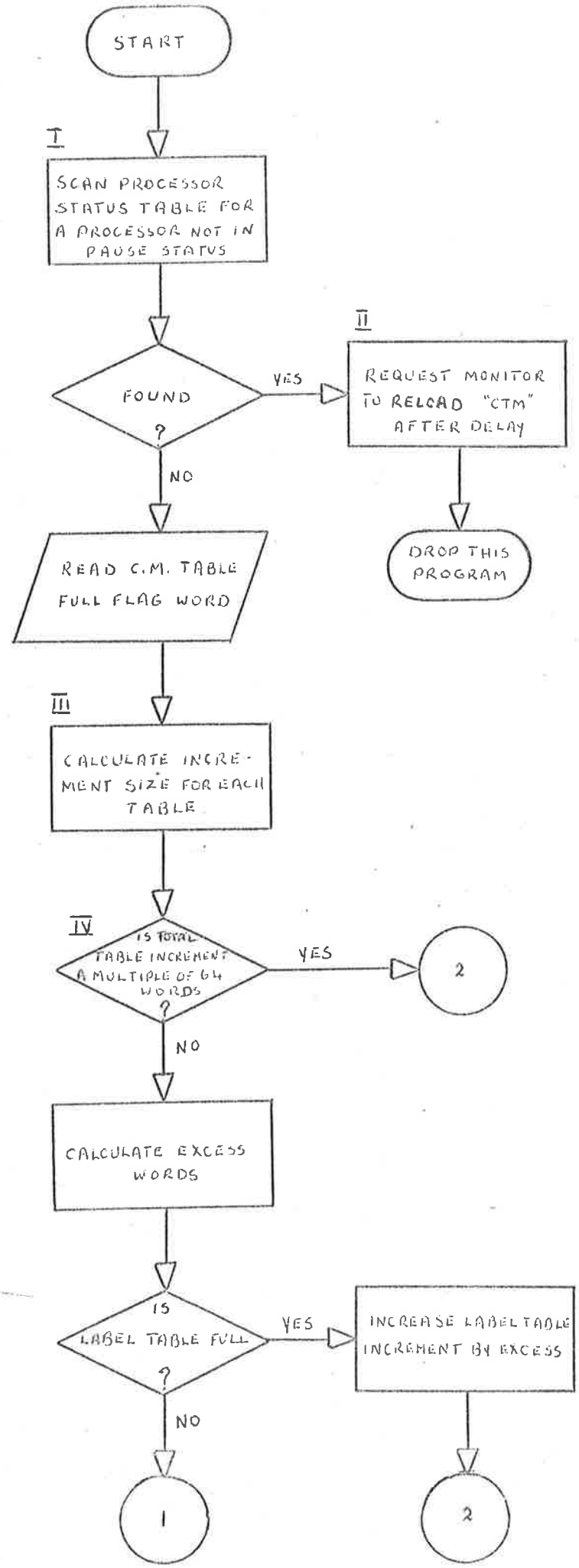
of central processor programs currently being held in central memory occupy every available word, but this is highly improbable.

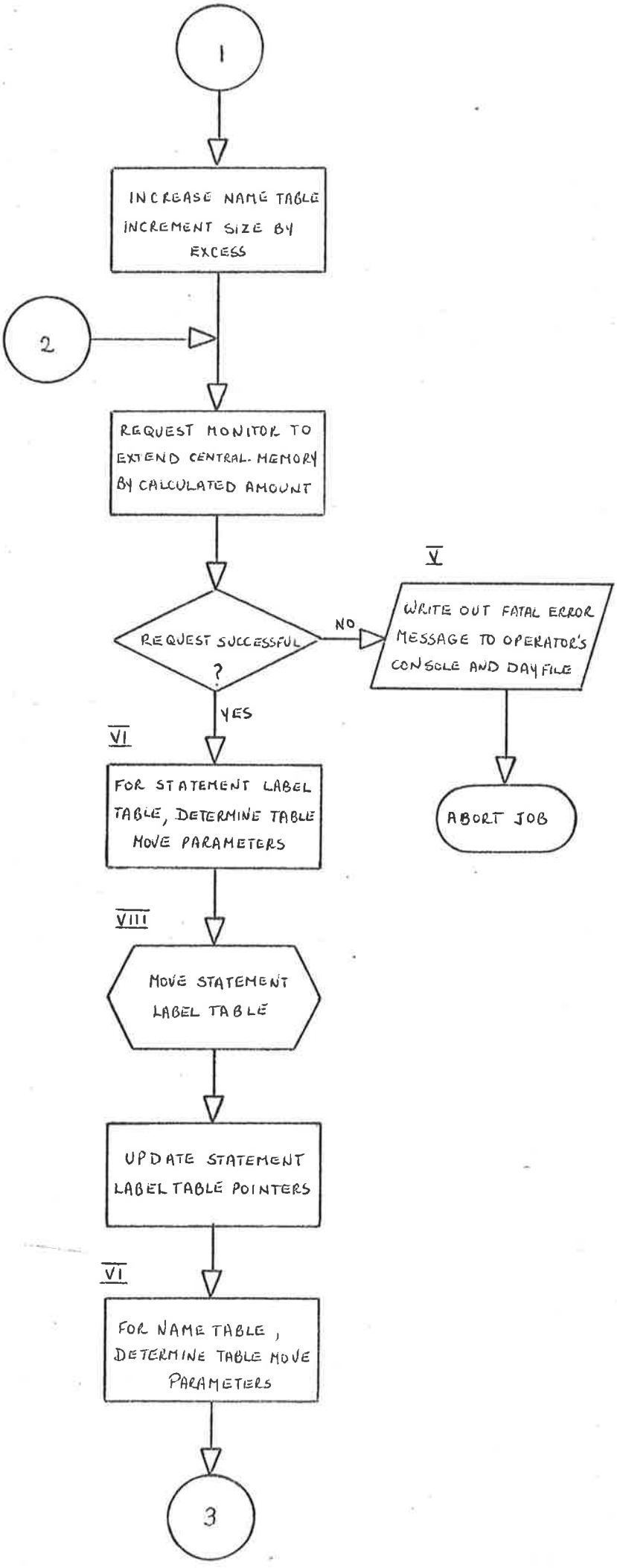
VI. The table move parameters are the source and destination addresses for the move and the number of words to be moved. In each case, these are calculated from the table pointers (see 5.1.4 through 5.1.7).

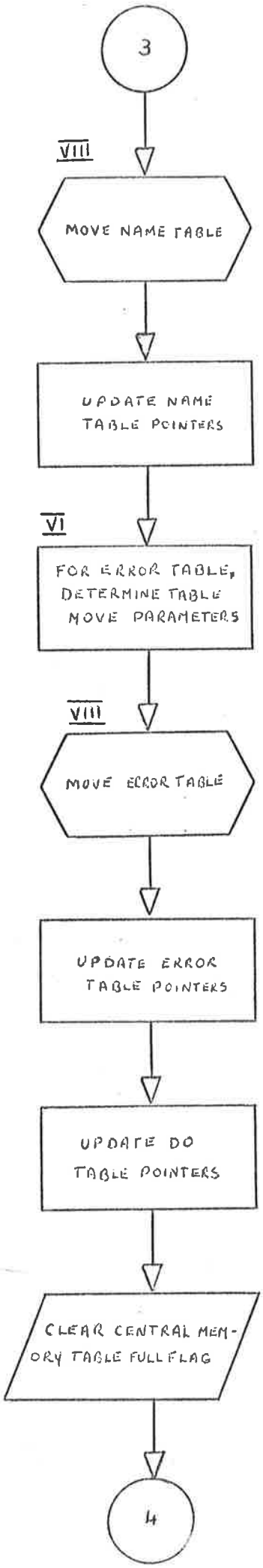
VII. The program scans through the processor status table and, for each program with "pause for CTM completion" status, requests monitor to reload the program.

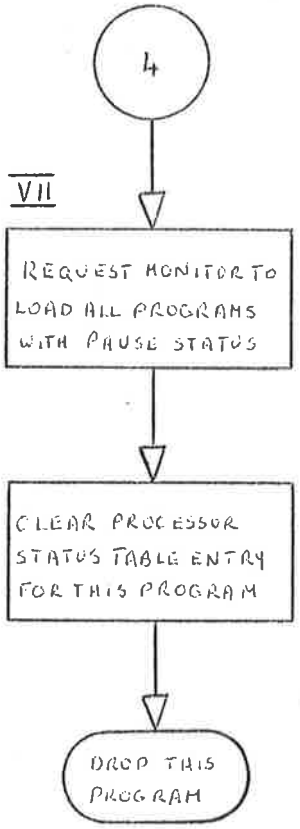
VIII. The table is moved in segments of sixty-four words via the program's memory until the required number of words have been moved.

PROGRAM CTM


START
I
SCAN PROCESSOR STATUS TABLE FOR A PROCESSOR NOT IN PAUSE STATUS
FOUND ?
YES
II
REQUEST MONITOR TO RELOAD "CTM" AFTER DELAY
DROP THIS PROGRAM
NO
READ C.M. TABLE FULL FLAG WORD
III
CALCULATE INCREMENT SIZE FOR EACH TABLE
IV
IS TOTAL TABLE INCREMENT A MULTIPLE OF 64 WORDS ?
YES
2
NO
CALCULATE EXCESS WORDS
IS LABEL TABLE FULL ?
YES
INCREASE LABEL TABLE INCREMENT BY EXCESS
NO
1
2


1
INCREASE NAME TABLE INCREMENT SIZE BY EXCESS
2
REQUEST MONITOR TO EXTEND CENTRAL MEMORY BY CALCULATED AMOUNT
V
REQUEST SUCCESSFUL ?
NO
WRITE OUT FATAL ERROR MESSAGE TO OPERATOR'S CONSOLE AND DAYFILE
ABORT JOB
YES
VI
FOR STATEMENT LABEL TABLE, DETERMINE TABLE MOVE PARAMETERS
VIII
MOVE STATEMENT LABEL TABLE
UPDATE STATEMENT LABEL TABLE POINTERS
VI
FOR NAME TABLE, DETERMINE TABLE MOVE PARAMETERS
3


3
VIII
MOVE NAME TABLE
UPDATE NAME TABLE POINTERS
VI
FOR ERROR TABLE, DETERMINE TABLE MOVE PARAMETERS
VIII
MOVE ERROR TABLE
UPDATE ERROR TABLE POINTERS
UPDATE DO TABLE POINTERS
CLEAR CENTRAL MEMORY TABLE FULL FLAG
4


4
VII
REQUEST MONITOR TO LOAD ALL PROGRAMS WITH PAUSE STATUS
CLEAR PROCESSOR STATUS TABLE ENTRY FOR THIS PROGRAM
DROP THIS PROGRAM

12. PROGRAM PP4 - WINDUP ROUTINE

12.0 Introduction

This program is the last to be loaded for each source subprogram. It is requested by either of programs FEA or MSA when these programs have emptied their processor communication buffers for the last time for the current subprogram. The latter event has occurred when a scan through the processor status table indicates that there are no other programs still running. However, if the current subprogram is BLOCK DATA, program PP4 will be requested by program DSA instead.

Program PP4 validates the nesting of DO loops by scanning the entries in the DO table (see 5.1.4) which are created by program MSA. It next searches the statement label table for entries corresponding to statement numbers referenced but undefined, for subsequent printing after the listing of the current source subprogram.

A sample source listing is shown in figure 12-1. Each source line is preceded by a four digit octal line number, commencing at zero. Source lines containing one or more errors are followed by an error line for each such error. The error line comprises a string of asterisk characters in which is embedded a two-letter error code, specifying the nature of the error. For source subprograms containing at least one error, the listing is followed by an explanation of the significance of each two-letter code which occurred in the subprogram. Each such explanation is cross-referenced by line number to every occurrence of

```
                FORTRAN PRE-COMPILATION BY MULTI-PROCESSED FORTRAN PRE-COMPILER SYSTEM.
0000            PROGRAM JONESY(INPUT,OUTPUT)
0001    C       *****TO FIND SINX USING A SERIES(NOT NESTED)*****
0002    C
0003            DO 200 I=0,90,5
*DOF*********************************
0004              X=I*0.01745329
0005              SUM=X
0006              TERM=X
0007              DENOM=3.0
0010              FAC=X*X
0011           30 TERM=-TERM*FAC/(DENOM*(DENOM-1.0))
0012              SUM=SUM+TERM
0013              IF (ABS(TERM)-0.000005) 40,40,50
0014           50 DENOM=DENOM+2.
0015              GO TO 30
0016           40 PRINT 60,I,X,I,SUM
0017           60 FORMAT(5X,I3,5X,F10.5,5X,3HSIN(I3,1H=F10.5)
0020          200 CONTINUE
0021              STOP
0022              END


DO******SYNTAX ERROR IN DO STATEMENT
0003
```

Figure 12-1

the corresponding error. The listing itself is produced by simply extracting source statements from the INPUT buffer where they have been kept intact since being transferred from the source file. A description of the method by which large subprograms are listed is given in 4.3.

As soon as the contents of the INPUT buffer are no longer required, a request is made to program CIO to refill it unless the end of the source file has already been detected. This operation is initiated as early as possible so that it will be completed before program GSR is loaded to process the next source subprogram.

If there is another subprogram, program PP4 re-initializes all central memory tables and, if necessary, requests monitor to reduce the central memory storage allocation to the 4K word minimum. It then requests monitor to load program GSR and finally drops itself.

12.1 Explanatory Notes to the Flow-Chart

This program does not request table access via the procedure described in 5.1.0 but accesses all tables directly, since it never runs concurrently with any other program.

The following notes refer to the accompanying flow-chart:

I. If false, program PP4 has already been partially executed for the current source subprogram. In that case, the program retrieves its data areas from central memory, into which it dumped them before dropping itself, and continues executing at the appropriate point. The dump area is situated in the first half of the processor 1 communication buffer.

II. Program CIO will fill the INPUT buffer with source starting at the beginning of the source file, since, before dropping itself, program GSR would have requested program CIO to backspace over the whole of the source file. While the INPUT buffer is being filled in this way, program PP4 continues execution. See also 4.3.

III. The statement label table entry pointer of each DO table entry (see notes IV and XVI of 10.1) is replaced by the source line number at which that statement number was defined. This is done so that the range of each DO loop in the original source subprogram is known.

IV. The error is flagged for the source line containing the erroneous DO statement.

V. This is done when every DO table entry has been set as the "current DO". In this way, for each DO loop, a check is made that all DO loops defined within its range are also terminated within the same range.

VI. It is invalid for nested DO loops to use the same control variable (see note XVI of 10.1).

VII. If this is false, program CIO has not yet completed filling the INPUT buffer with further source. Therefore program PP4 dumps its data areas into central memory and requests to be reloaded after a suitable delay. It then drops itself. See also note I above.

VIII. When program PP4 is subsequently restarted, execution will resume here. See also note I above.

IX. For large source subprograms, the logical buffer end-point is set at two-thirds of the buffer capacity, so that sufficient warning is available that the buffer is nearing exhaustion. This enables program CIO to be called to refill the buffer before it becomes completely empty. If the latter should still occur, program PP4 will drop itself in the usual way (see note VII above) and be subsequently reloaded after a suitable delay.

X. A circular buffer for the OUTPUT file is created in central memory. It occupies the region from the middle of the processor 1 communication buffer to the end of the job's central memory field. It includes the storage space occupied by the remaining two processor communication buffers and the DO, error, name and statement label tables since their contents are no longer required. However, the buffer allocated will not be greater than 4K words so that addressing within the buffer can be expressed in twelve bits.

XI. A source card with a point character punched in column 1 is treated as a comment card. For listing purposes, it causes the source listing to skip to the top of the next page but is not itself printed.

XII. The listing of a source line contains the source line number as four octal digit characters at the start of the line followed by the card image itself. See figure 12-1.

XIII. Error numbers are only used internally. In the listing, a source statement in error is flagged by a two-letter error code. The codes used are those of Control Data's RUN compiler. They are shown in appendix 3.

XIV. Since the listing of the source statements themselves has now been completed, the contents of the INPUT buffer are now no longer required. Therefore, if the end of the source file has not yet been detected by program CIO, it is called to refill the INPUT buffer once again.

XV. This is done so that the full text of the error message corresponding to each error code which was flagged for the current source subprogram can be printed together with a cross-reference by line number of each occurrence of a particular error.

XVI. The error message table and its associated index table are stored in central memory (see 5.1.10).

XVII. If program CIO has been previously called to process the OUTPUT buffer (see note XXIII below), it is necessary to check that the operation has been completed before making further requests to program CIO for OUTPUT buffer processing. If such processing is incomplete, it will be necessary for program PP4 to drop itself (see note I above).

XVIII. Since the generation of output for the current source subprogram has been completed, the contents of the OUTPUT buffer must be written out to the OUTPUT file. Therefore, program CIO is requested to perform the "write end record" function to ensure the buffer is completely emptied, even if it does not contain an exact multiple of sixty-four words.

XIX. This starts the compilation of the next source subprogram.

XX. If false, program CIO is called to refill the INPUT buffer. Thereafter programs CIO and PP4 will run concurrently. Thus, program PP4 continues extracting from the buffer until eventually the physical end of data in the buffer is reached.

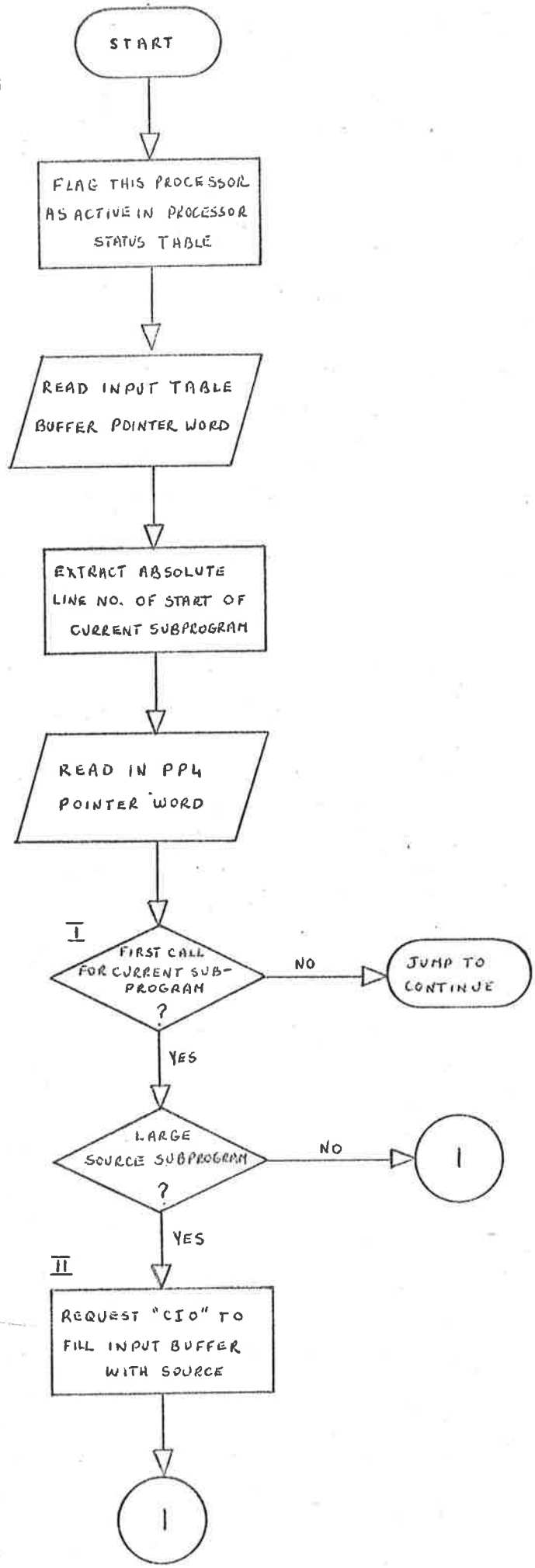
XXI. A check is made that program CIO has finished refilling the buffer and if not (it is possible that it may not have even started yet because of the unavailability of peripheral processors in which to run), program PP4 drops itself after requesting to be recalled after a suitable delay.

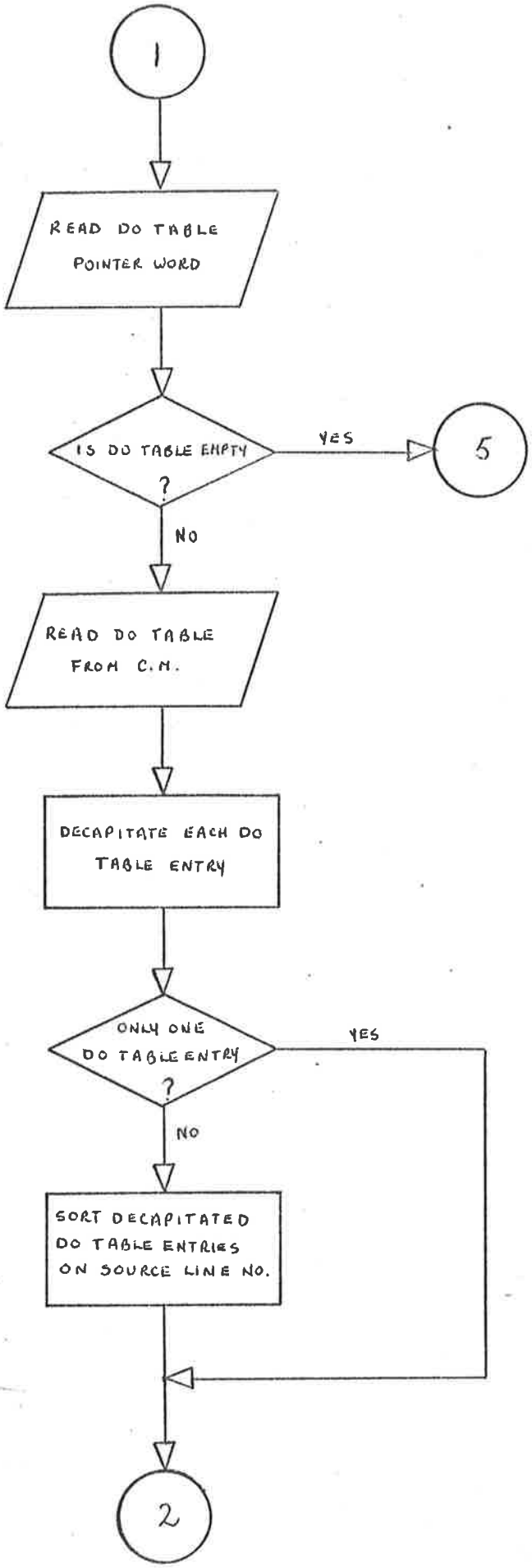
XXII. If program CIO detected the end of the source file while refilling the INPUT buffer, the current subprogram is no longer regarded as being large.

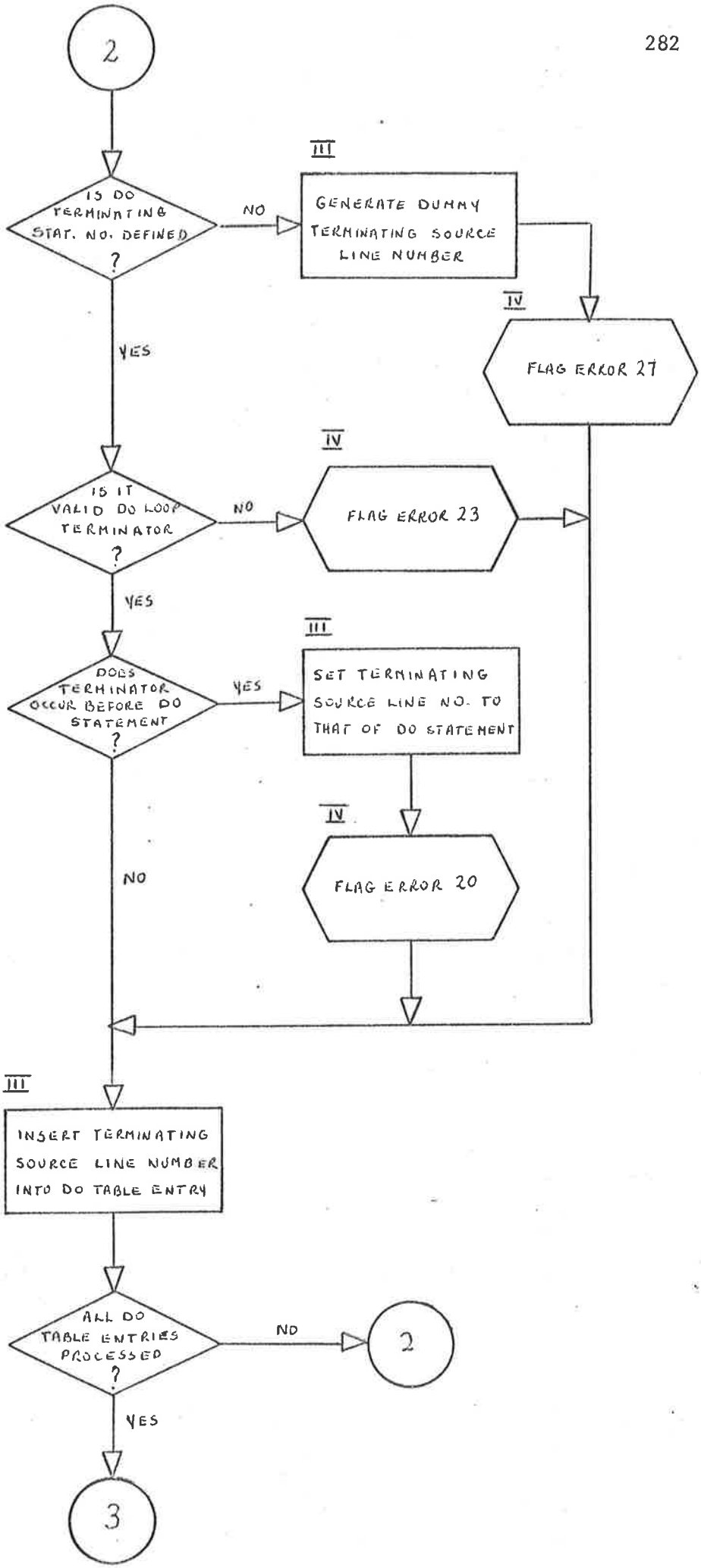
XXIII. This is a similar mechanism to that used for the INPUT file. It ensures that program CIO is called before the OUTPUT buffer becomes completely full. Nevertheless, program PP4 continues filling the buffer until there is insufficient space (that is, fourteen buffer words) for another line of maximum length. If at this point, program CIO has not completed its processing of the buffer, program PP4 drops itself (see also notes I, XX and XXI above).

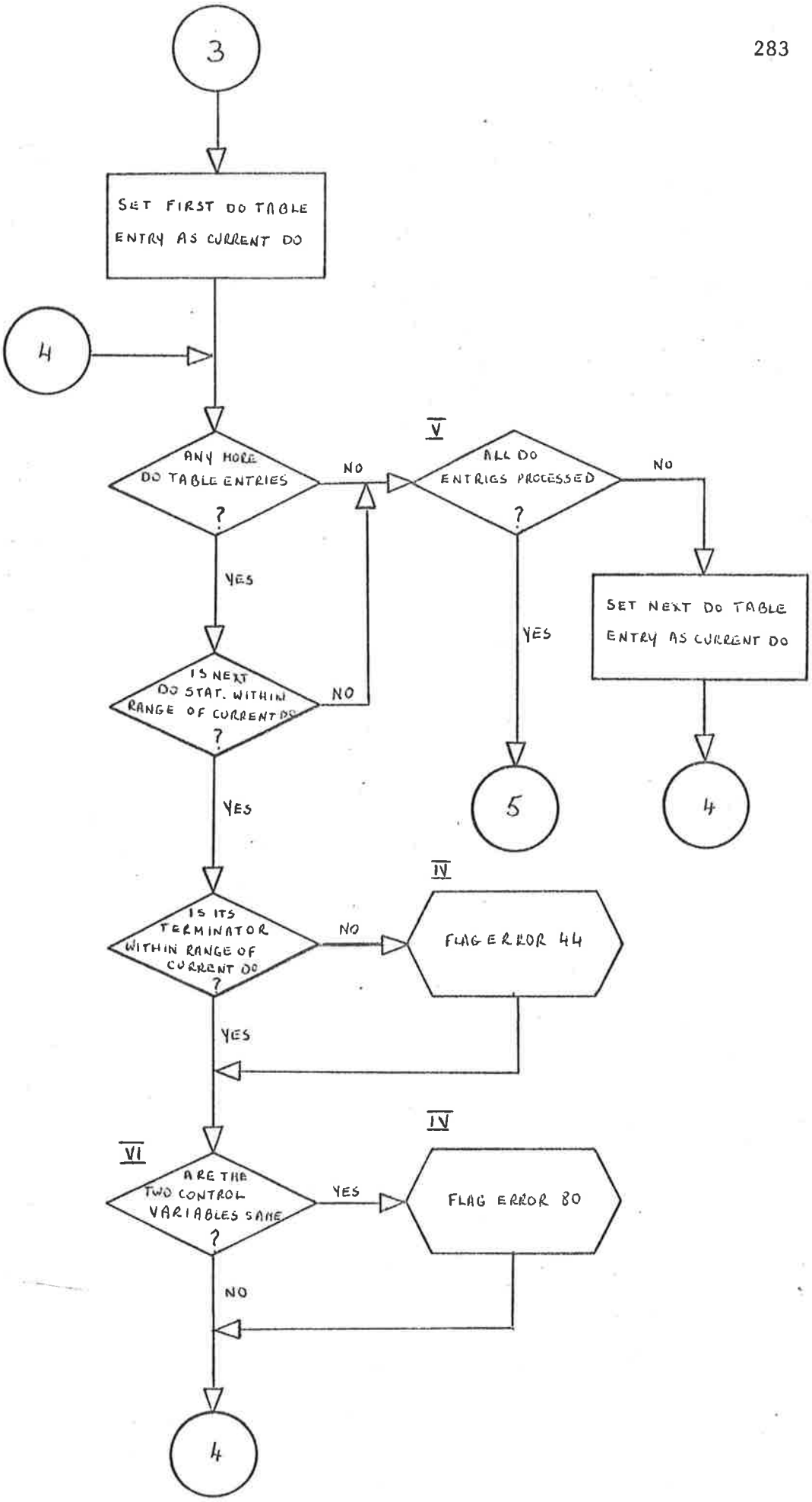
PROGRAM

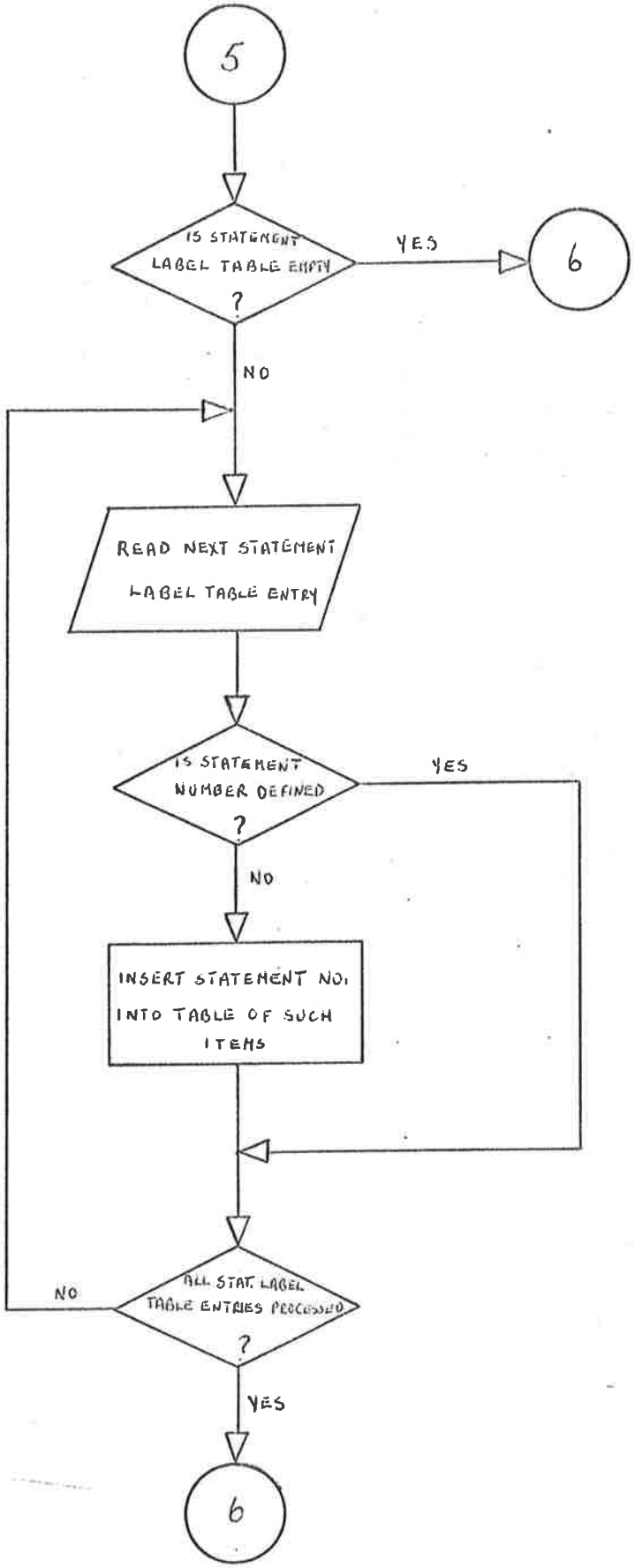
PP4

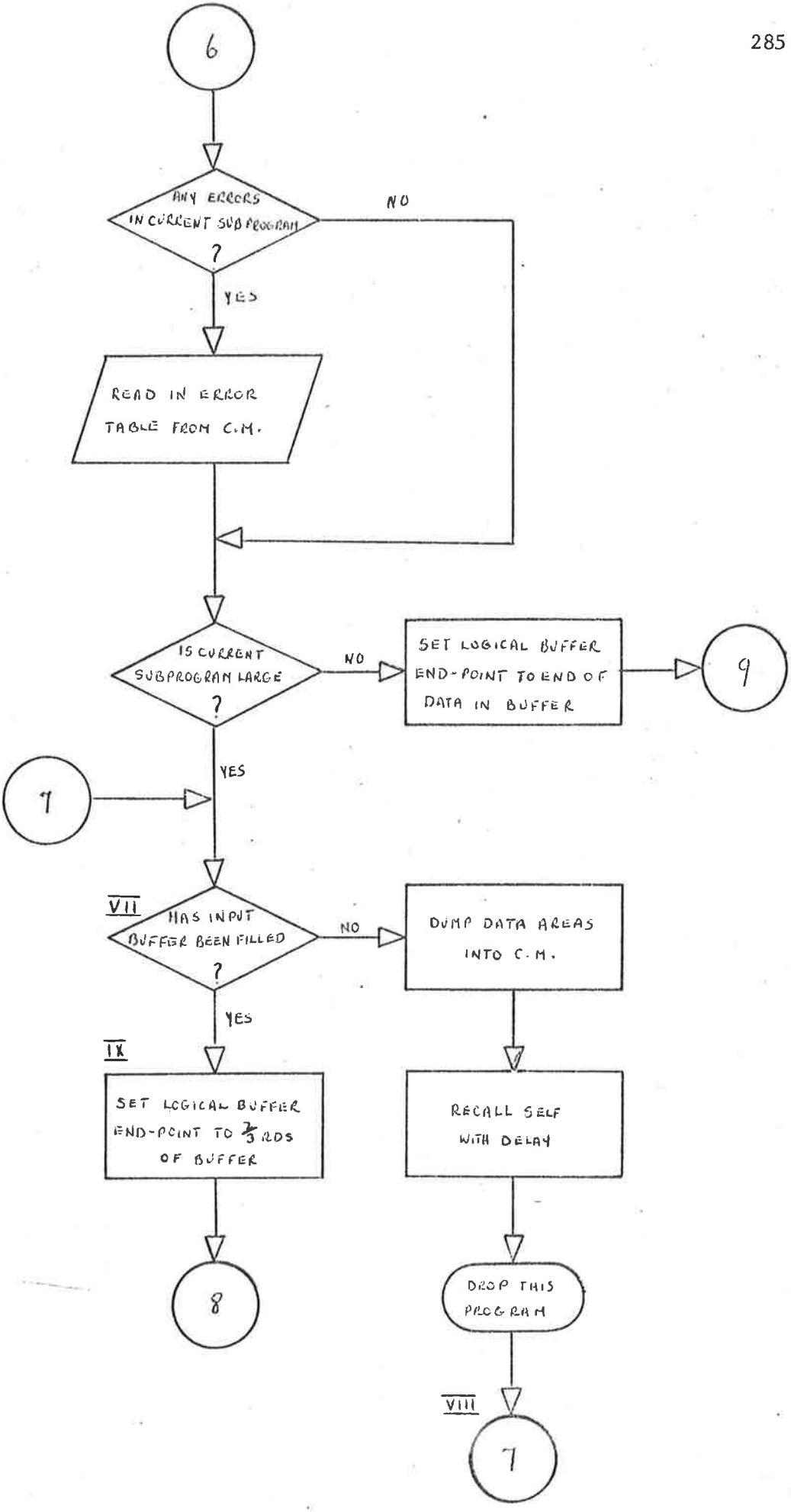

START
FLAG THIS PROCESSOR AS ACTIVE IN PROCESSOR STATUS TABLE
READ INPUT TABLE BUFFER POINTER WORD
EXTRACT ABSOLUTE LINE NO. OF START OF CURRENT SUBPROGRAM
READ IN PP4 POINTER WORD
I
FIRST CALL FOR CURRENT SUB-PROGRAM ?
NO
JUMP TO CONTINUE
YES
LARGE SOURCE SUBPROGRAM ?
NO
1
YES
II
REQUEST "CIO" TO FILL INPUT BUFFER WITH SOURCE
1

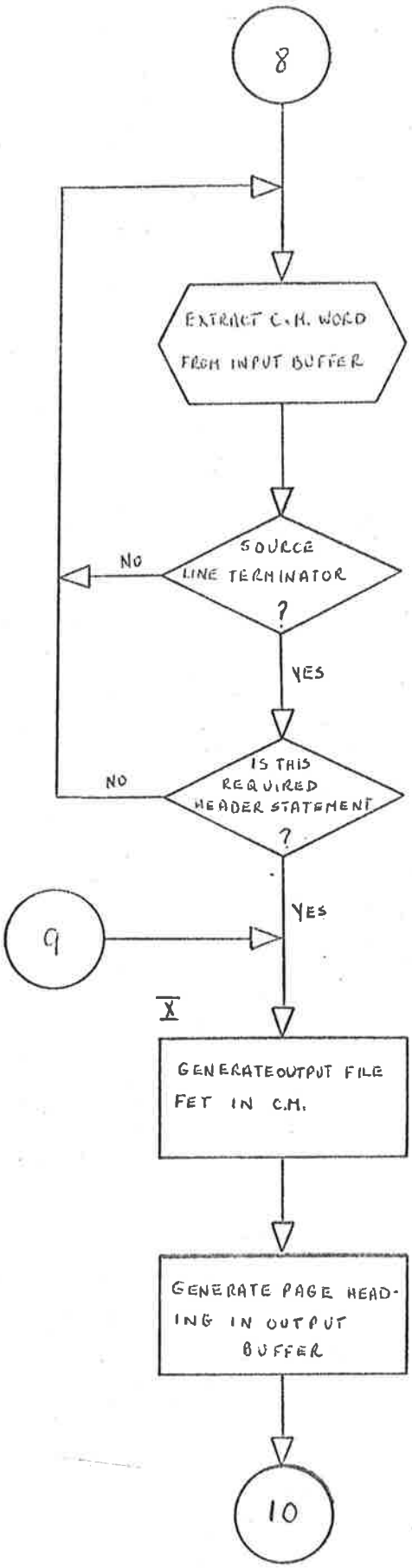

1
READ DO TABLE POINTER WORD
IS DO TABLE EMPTY ?
YES
5
NO
READ DO TABLE FROM C.M.
DECAPITATE EACH DO TABLE ENTRY
ONLY ONE DO TABLE ENTRY ?
YES
NO
SORT DECAPITATED DO TABLE ENTRIES ON SOURCE LINE NO.
2

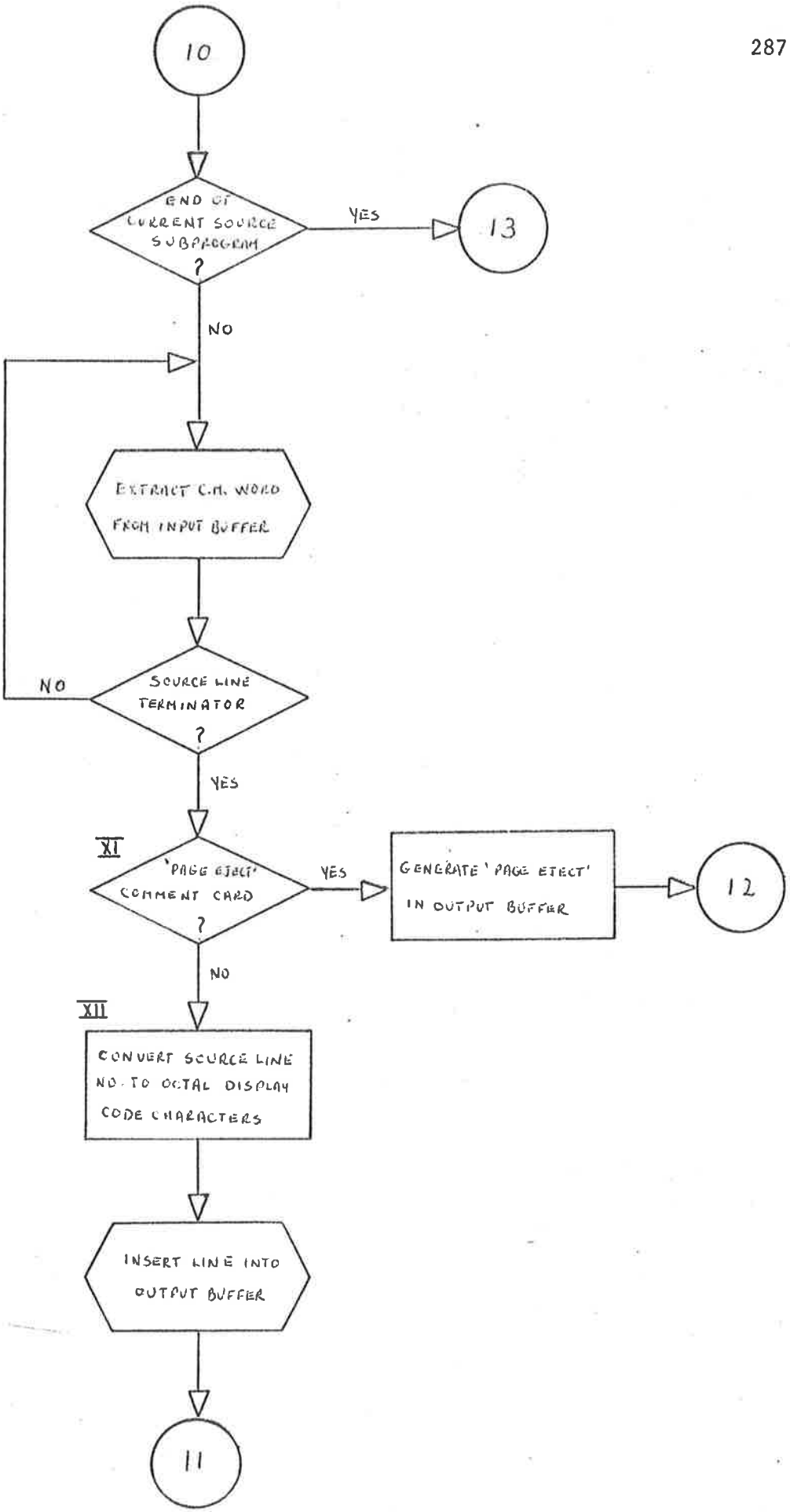

2
IS DO TERMINATING STAT. NO. DEFINED ?
NO
III
GENERATE DUMMY TERMINATING SOURCE LINE NUMBER
IV
FLAG ERROR 27
YES
IV
IS IT VALID DO LOOP TERMINATOR ?
NO
FLAG ERROR 23
YES
III
DOES TERMINATOR OCCUR BEFORE DO STATEMENT ?
YES
SET TERMINATING SOURCE LINE NO. TO THAT OF DO STATEMENT
IV
FLAG ERROR 20
NO
III
INSERT TERMINATING SOURCE LINE NUMBER INTO DO TABLE ENTRY
ALL DO TABLE ENTRIES PROCESSED ?
NO
2
YES
3

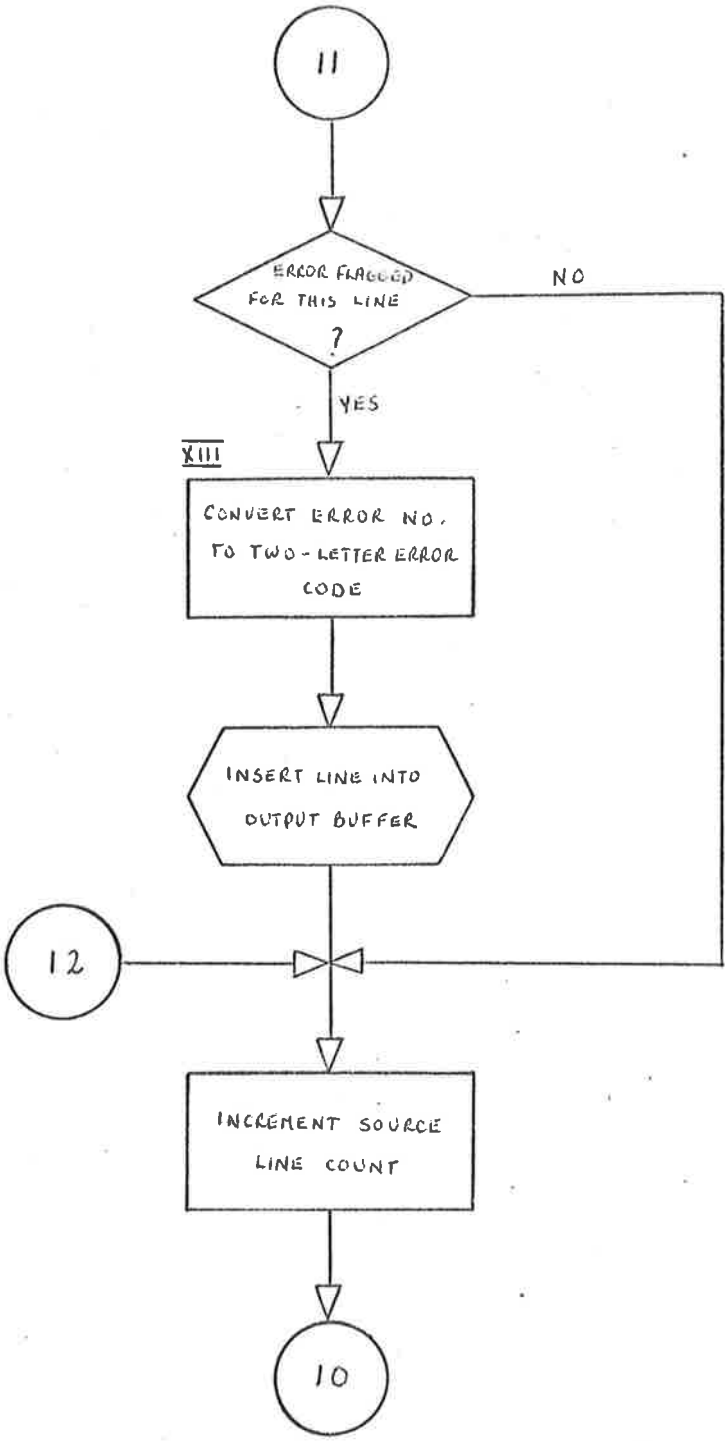

3
SET FIRST DO TABLE ENTRY AS CURRENT DO
4
ANY MORE DO TABLE ENTRIES ?
NO
V
ALL DO ENTRIES PROCESSED ?
NO
YES
SET NEXT DO TABLE ENTRY AS CURRENT DO
IS NEXT DO STAT. WITHIN RANGE OF CURRENT DO ?
NO
YES
5
4
IV
IS ITS TERMINATOR WITHIN RANGE OF CURRENT DO ?
NO
FLAG ERROR 44
YES
IV
VI
ARE THE TWO CONTROL VARIABLES SAME ?
YES
FLAG ERROR 80
NO
4

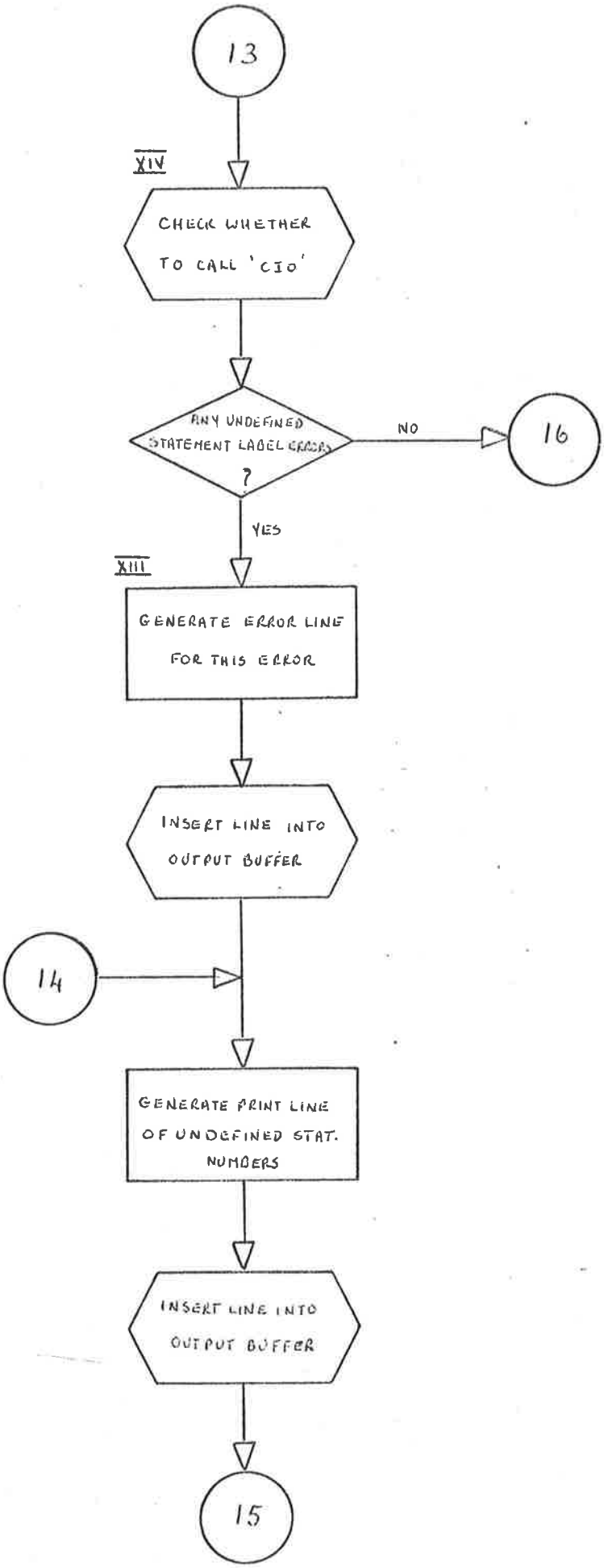

5
IS STATEMENT LABEL TABLE EMPTY ?
YES
6
NO
READ NEXT STATEMENT LABEL TABLE ENTRY
IS STATEMENT NUMBER DEFINED ?
YES
NO
INSERT STATEMENT NO. INTO TABLE OF SUCH ITEMS
ALL STAT. LABEL TABLE ENTRIES PROCESSED ?
NO
YES
6

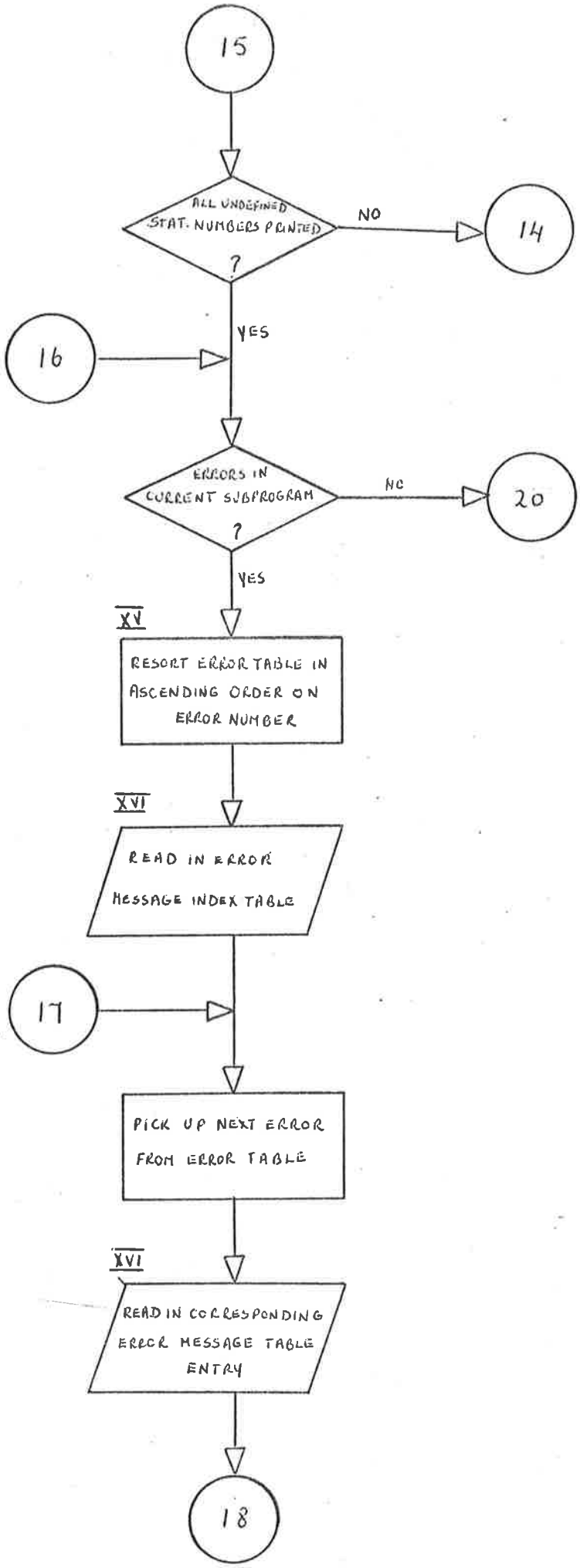

6
ANY ERRORS IN CURRENT SUBPROGRAM ?
NO
YES
READ IN ERROR TABLE FROM C.M.
IS CURRENT SUBPROGRAM LARGE ?
NO
SET LOGICAL BUFFER END-POINT TO END OF DATA IN BUFFER
9
YES
7
VII
HAS INPUT BUFFER BEEN FILLED ?
NO
DUMP DATA AREAS INTO C.M.
YES
IX
SET LOGICAL BUFFER END-POINT TO 2/3RDS OF BUFFER
RECALL SELF WITH DELAY
8
DROP THIS PROGRAM
VIII
7

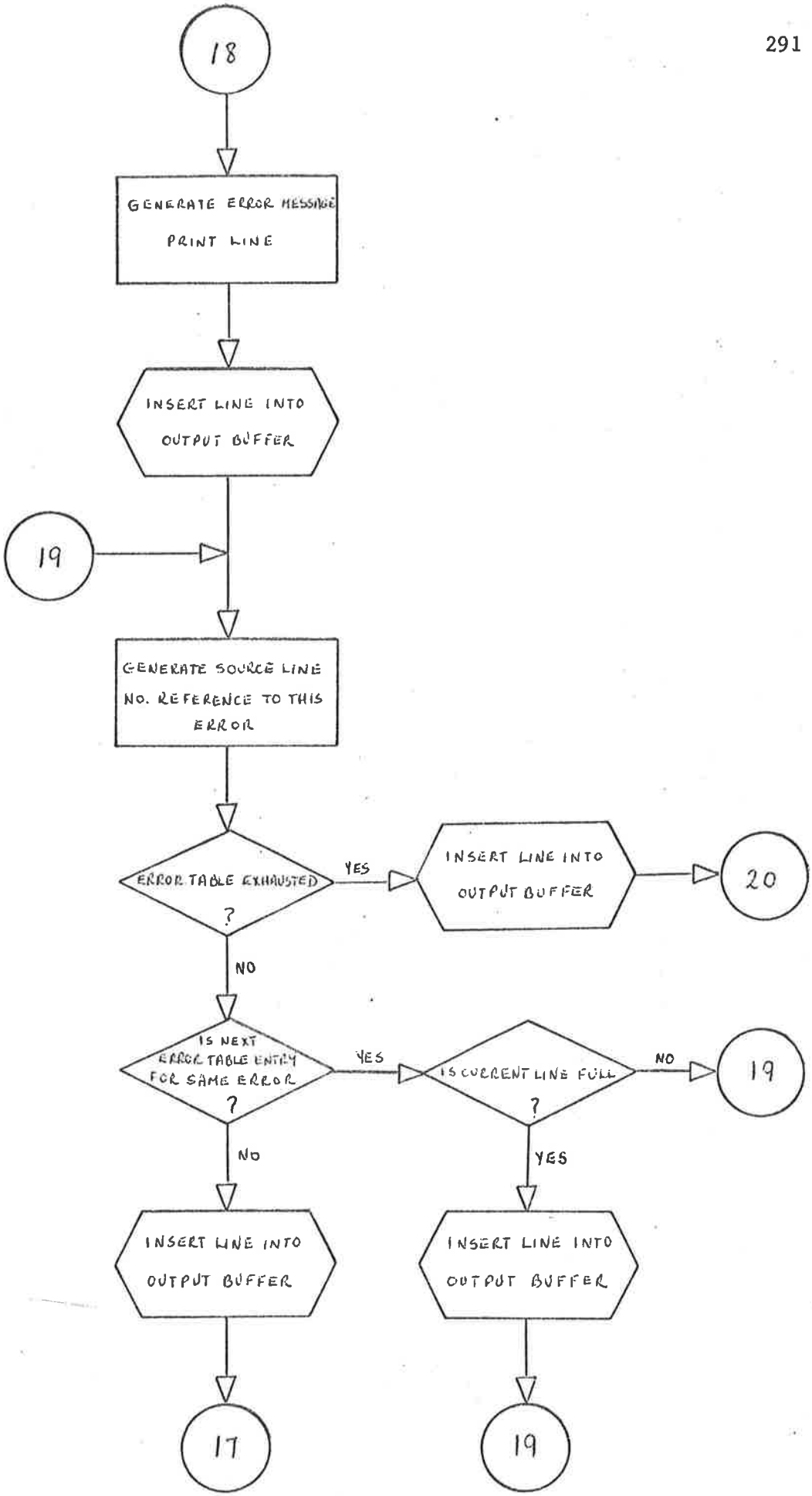

8
EXTRACT C.M. WORD FROM INPUT BUFFER
SOURCE LINE TERMINATOR ?
NO
YES
IS THIS REQUIRED HEADER STATEMENT ?
NO
YES
9
X
GENERATE OUTPUT FILE FET IN C.M.
GENERATE PAGE HEADING IN OUTPUT BUFFER
10

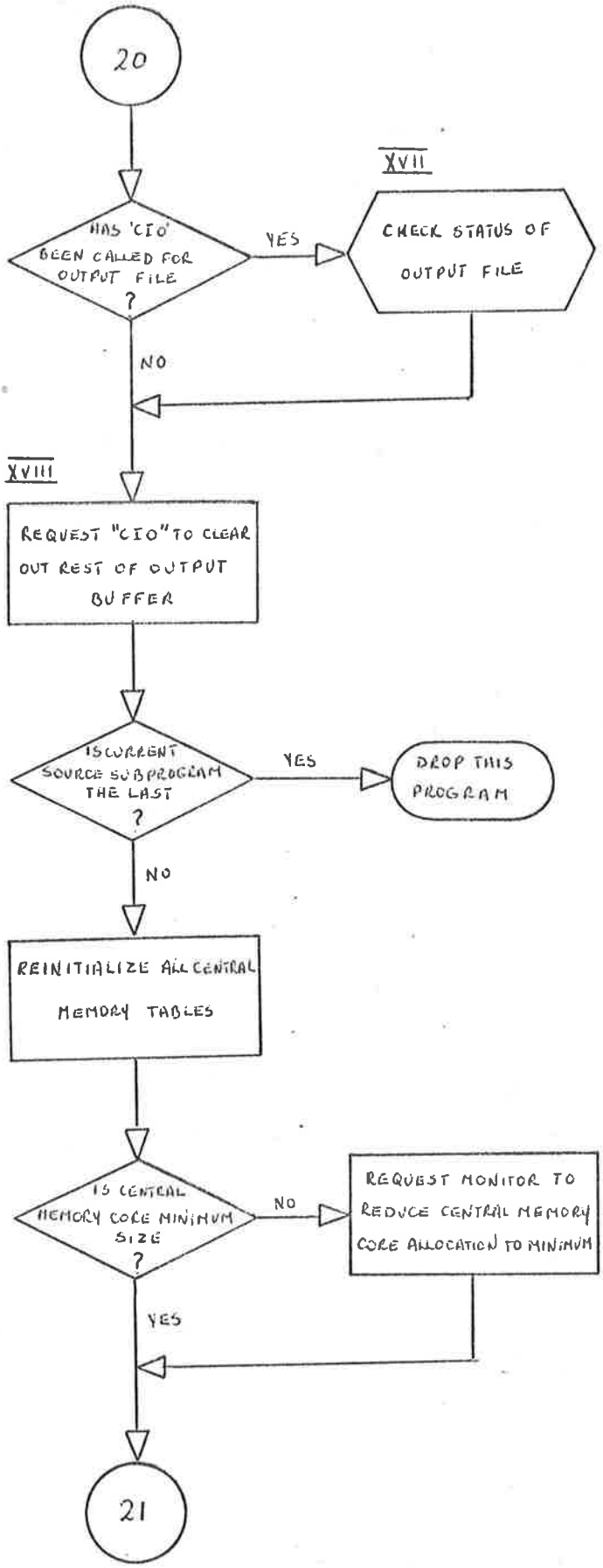

10
END OF CURRENT SOURCE SUBPROGRAM ?
YES
13
NO
EXTRACT C.M. WORD FROM INPUT BUFFER
SOURCE LINE TERMINATOR ?
NO
YES
XI
'PAGE EJECT' COMMENT CARD ?
YES
GENERATE 'PAGE EJECT' IN OUTPUT BUFFER
12
NO
XII
CONVERT SOURCE LINE NO. TO OCTAL DISPLAY CODE CHARACTERS
INSERT LINE INTO OUTPUT BUFFER
11

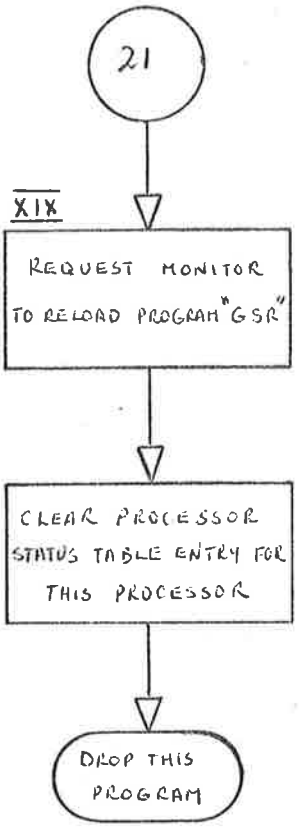

11
ERROR FLAGGED FOR THIS LINE ?
NO
YES
XIII
CONVERT ERROR NO. TO TWO-LETTER ERROR CODE
INSERT LINE INTO OUTPUT BUFFER
12
INCREMENT SOURCE LINE COUNT
10

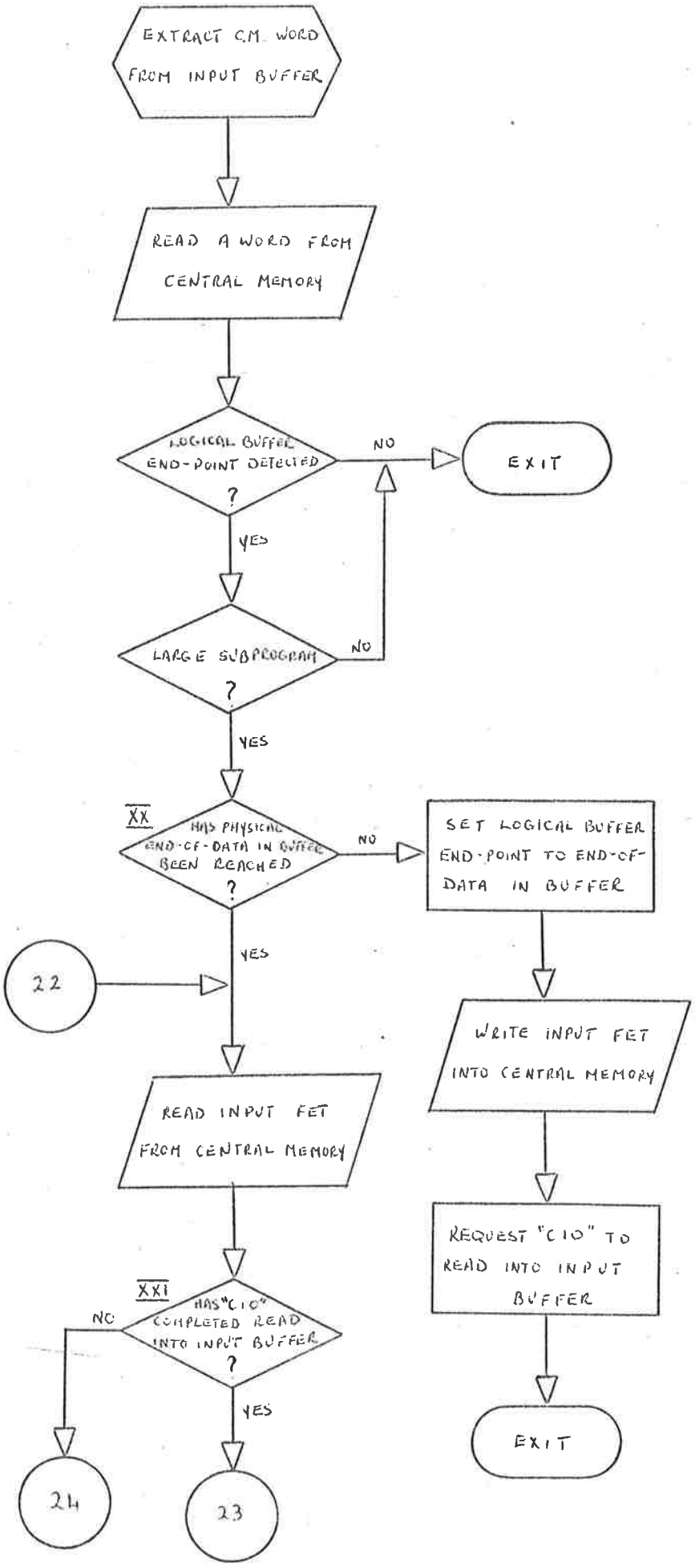

13
XIV
CHECK WHETHER TO CALL 'CIO'
ANY UNDEFINED STATEMENT LABEL ERRORS ?
NO
16
YES
XIII
GENERATE ERROR LINE FOR THIS ERROR
INSERT LINE INTO OUTPUT BUFFER
14
GENERATE PRINT LINE OF UNDEFINED STAT. NUMBERS
INSERT LINE INTO OUTPUT BUFFER
15

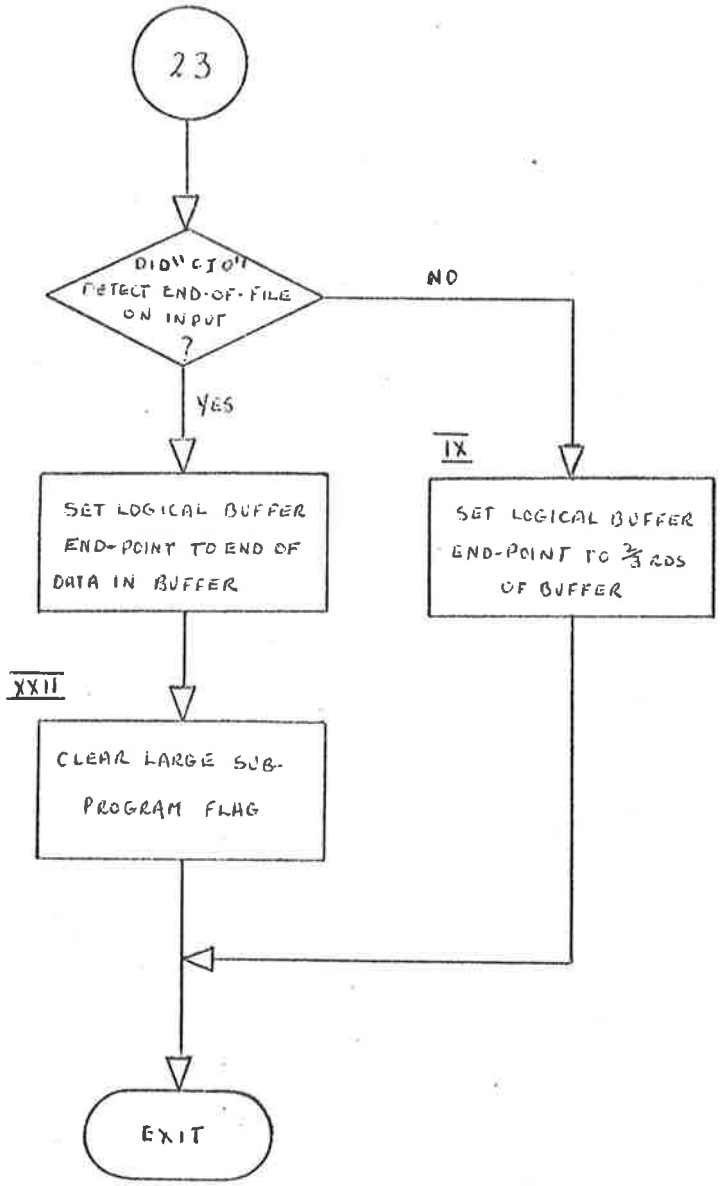

15
ALL UNDEFINED STAT. NUMBERS PRINTED ?
NO
14
YES
16
ERRORS IN CURRENT SUBPROGRAM ?
NO
20
YES
XV
RESORT ERROR TABLE IN ASCENDING ORDER ON ERROR NUMBER
XVI
READ IN ERROR MESSAGE INDEX TABLE
17
PICK UP NEXT ERROR FROM ERROR TABLE
XVI
READ IN CORRESPONDING ERROR MESSAGE TABLE ENTRY
18

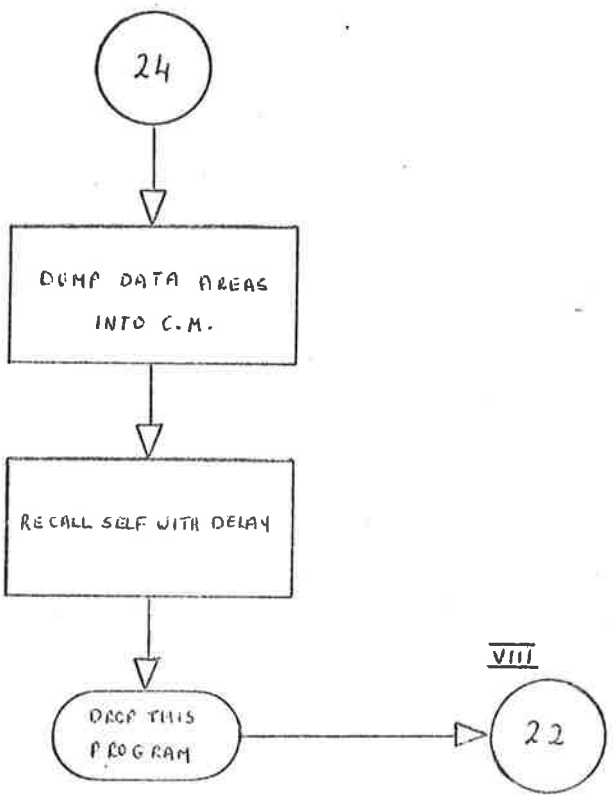

18
GENERATE ERROR MESSAGE PRINT LINE
INSERT LINE INTO OUTPUT BUFFER
19
GENERATE SOURCE LINE NO. REFERENCE TO THIS ERROR
ERROR TABLE EXHAUSTED ?
YES
INSERT LINE INTO OUTPUT BUFFER
20
NO
IS NEXT ERROR TABLE ENTRY FOR SAME ERROR ?
YES
IS CURRENT LINE FULL ?
NO
19
No
YES
INSERT LINE INTO OUTPUT BUFFER
INSERT LINE INTO OUTPUT BUFFER
17
19

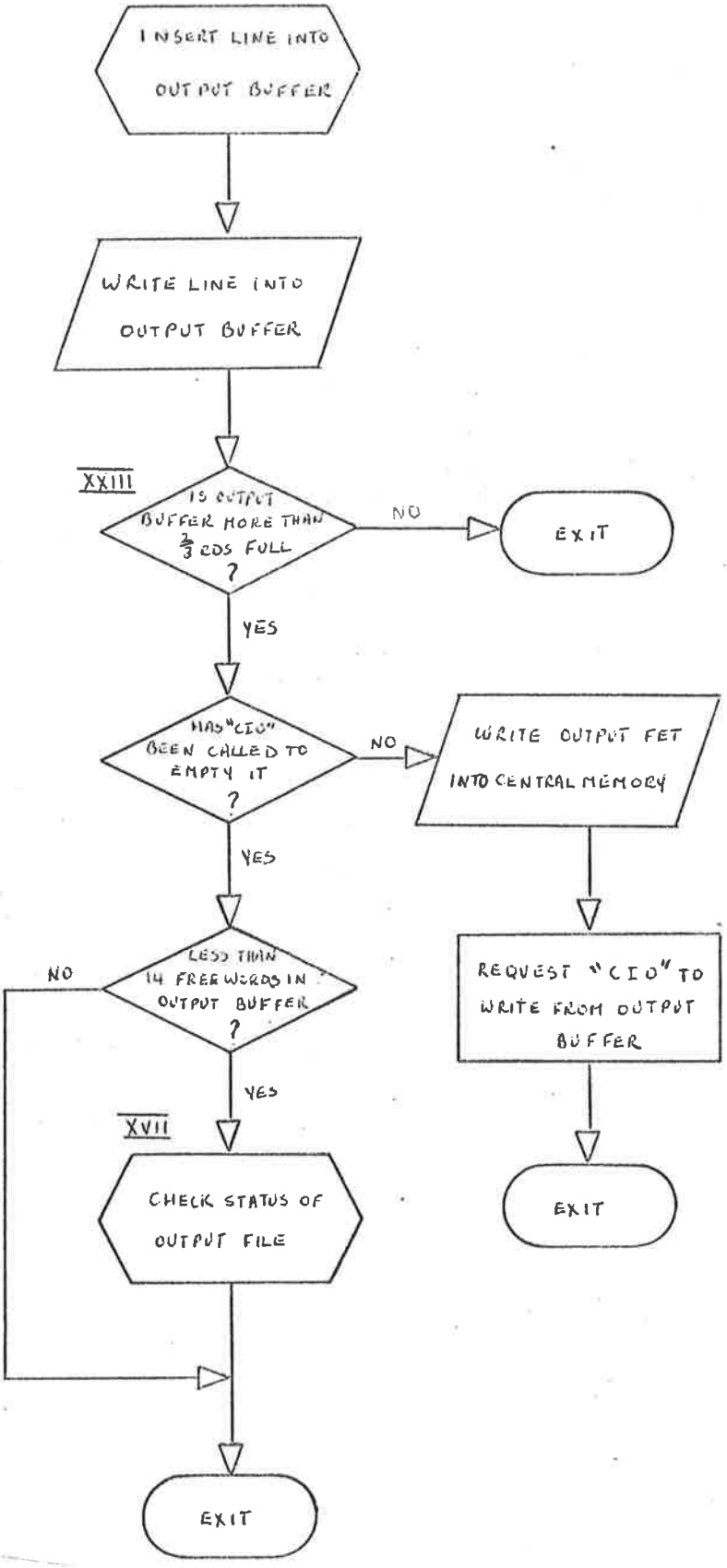

20
XVII
HAS 'CIO' BEEN CALLED FOR OUTPUT FILE ?
YES
CHECK STATUS OF OUTPUT FILE
NO
XVIII
REQUEST "CIO" TO CLEAR OUT REST OF OUTPUT BUFFER
IS CURRENT SOURCE SUBPROGRAM THE LAST ?
YES
DROP THIS PROGRAM
NO
REINITIALIZE ALL CENTRAL MEMORY TABLES
IS CENTRAL MEMORY CORE MINIMUM SIZE ?
NO
REQUEST MONITOR TO REDUCE CENTRAL MEMORY CORE ALLOCATION TO MINIMUM
YES
21


21
XIX
REQUEST MONITOR TO RELOAD PROGRAM "GSR"
CLEAR PROCESSOR STATUS TABLE ENTRY FOR THIS PROCESSOR
DROP THIS PROGRAM


EXTRACT C.M. WORD FROM INPUT BUFFER
READ A WORD FROM CENTRAL MEMORY
LOGICAL BUFFER END-POINT DETECTED ?
NO
EXIT
YES
LARGE SUBPROGRAM ?
NO
YES
XX
HAS PHYSICAL END-OF-DATA IN BUFFER BEEN REACHED ?
NO
SET LOGICAL BUFFER END-POINT TO END-OF-DATA IN BUFFER
YES
22
WRITE INPUT FET INTO CENTRAL MEMORY
READ INPUT FET FROM CENTRAL MEMORY
REQUEST "CIO" TO READ INTO INPUT BUFFER
XXI
HAS "CIO" COMPLETED READ INTO INPUT BUFFER ?
NO
YES
EXIT
24
23


23
DID CIO NOT DETECT END-OF-FILE ON INPUT ?
NO
YES
IX
SET LOGICAL BUFFER END-POINT TO END OF DATA IN BUFFER
SET LOGICAL BUFFER END-POINT TO 2/3 RDS OF BUFFER
XXII
CLEAR LARGE SUB-PROGRAM FLAG
EXIT


24
DUMP DATA AREAS INTO C.M.
RECALL SELF WITH DELAY
VIII
DROP THIS PROGRAM
22


INSERT LINE INTO OUTPUT BUFFER
WRITE LINE INTO OUTPUT BUFFER
XXIII
IS OUTPUT BUFFER MORE THAN 2/3 CDS FULL ?
NO
EXIT
YES
HAS "CIO" BEEN CALLED TO EMPTY IT ?
NO
WRITE OUTPUT FET INTO CENTRAL MEMORY
YES
LESS THAN 14 FREE WORDS IN OUTPUT BUFFER ?
NO
REQUEST "CIO" TO WRITE FROM OUTPUT BUFFER
YES
XVII
CHECK STATUS OF OUTPUT FILE
EXIT
EXIT

13. COMPILATION SYSTEM TESTING

13.0 Introduction

From the commencement of the experiment, it was decided to implement the system in two stages. The first stage was to produce an essentially single processor version in which there was only one program executing at any one time and when its task was completed, it requested monitor to load the next program in sequence. Thus, the compilation began with the running of program GSR, followed by programs DSA, FEA, MSA and PP4. Then, if the latter detected that the buffer contained another source subprogram, it would eventually recall program GSR. The main logic of program CTM was originally incorporated in each of programs GSR, DSA, FEA and MSA until the code of programs GSR and FEA grew too large, after which a non-multiprocessing version of program CTM was created and the other programs modified accordingly. The logic of program PCR was, of course, not required at this stage. The second stage was to produce a multiprocessing version of the first stage. This approach had a number of advantages. Firstly, it was considerably simpler to write, but particularly so to test, a working compiler in stage one without the added complexity of multiprocessing. Secondly, the development of stage one rendered considerable opportunity to gain further insight into the difficulties that lay ahead in the conversion of this first stage into the second. Indeed, a number of changes (see 2.3) had to be made to the original design during, and as a result of, the implementation of stage one.

During stage one, the system was written and tested in modules, so that, on the successful completion of each module, the fraction of the total system so far implemented was working satisfactorily. However, due to the immensity of the task, more than one module was being developed at any one time. Nevertheless, only one module from each program was being produced in this way and all such modules were independent of each other. Source programs under test were compiled by an assembler which was written by the author for an ICL 1903A system and run against a specially prepared central memory core image by a hardware simulator program written by the author for the same machine. The key features of these programs are described in the next two sections.

13.1 The Assembler

This program, which is written in a mixture of FORTRAN and the assembly language of the ICL 1903A system, accepts CDC 6400 peripheral processor source language on cards or magnetic tape in ICL subfile format and produces binary object code on tape or disc as well as a listing of the assembled source program. Both outputs are the equivalents of those produced by the Control Data supplied software. The binary object code will subsequently be input to the simulator for "execution".

13.2 The Simulator

This program, which is coded in a combination of FORTRAN and the assembly language of the ICL 1903A, is driven by parameter cards presented to it. These parameters specify the programs to be loaded into peripheral processors and

the names of files from which the loading operation is to be performed and similar information about the required central memory core image. There are also parameters to control the collection and printing of diagnostic data about the programs being simulated.

Peripheral processor memories are held in FORTRAN arrays within the simulator program itself whilst, during the simulation, the central memory, because of its size, is held as a random access file. Each record of this file represents a segment of one hundred central memory words and is swapped in and out of the simulator's data area as required.

The peripheral processor resident (see 3.2), which is a part of SCOPE, is loaded from a library file into the appropriate part of every peripheral processor memory before simulation begins. The rest of the required part of SCOPE, including the monitor (MTR) and program CIO, is built into the simulator itself and, as far as the peripheral processors are concerned, provide facilities equivalent to their CDC 6400 counterparts. For example, if a simulator parameter card specifies that data cards, representing a FORTRAN source program punched in CDC card code, are present, the simulator converts them into a disc file from which they will be loaded into an INPUT buffer in central memory by the simulated "program CIO" facility on request from a peripheral processor program.

The diagnostic data, which can be produced by the simulator, includes dumps of the central and peripheral processor memories and instruction traces of either all instructions, all jump instructions or subroutine jump instructions only,

specified by parameter card and controlled by switches within the simulator. However, in order to reduce the amount of unnecessary printer output produced, there is the facility to store diagnostic data (unless already printed) about all executed instructions in a circular buffer. The contents of this buffer is output on the occurrence of various specifiable conditions and represents the results of the execution of the last sixty instructions approximately.

13.3 An Example

As an example of the way in which the system performs compilation, consider the source subprogram shown in figure 13-1. When this source subprogram is input to the multiprocessing compiler, the state of the system at various stages during execution is shown in figure 13-2. Each of the twenty stages in the figure represents the state of the system just after a program has been loaded into or dropped from a peripheral processor. The time at which each stage occurs is shown in the figure in terms of a relative time unit (expressed in octal) which should not be interpreted as time elapsed by some clock but rather as an indication of the relative amounts of processing performed by the programs between stages. The following is a brief description of the actions of each program at each stage and should be read in conjunction with figures 13-1 and 13-2:

```
1           PROGRAM JONSEY(INPUT,OUTPUT)
2     C     *****TO FIND SINX USING A SERIES(NOT NESTED)*****
3     C
4           DO 200 I = 0,90,5
5           X=I*0.01745329
6           SUM=X
7           TERM=X
8           DENOM=3.0
9           FAC=X*X
10       30 TERM=-TERM*FAC/(DENOM*(DENOM-1.0))
11          SUM=SUM+TERM
12          IF(ABS(TERM)-0.000005)40,40,50
13       50 DENOM=DENOM+2
14          GOTO 30
15       40 PRINT 60,I,X,I,SUM
16       60 FORMAT(5X,I3,5X,F10.5,5X,3HSINI3,1H=F10.5)
17      200 CONTINUE
18          STOP
19          END
```

Figure 13-1

STAGE	TIME	PERIPHERAL PROCESSOR 1	PERIPHERAL PROCESSOR 2	PERIPHERAL PROCESSOR 3	PERIPHERAL PROCESSOR 4
1	0	GSR	-	-	-
2	436	GSR	DSA	-	-
3	1021	GSR	DSA	MSA	-
4	1053	GSR	-	MSA	-
5	1456	GSR	-	-	-
6	2533	GSR	PCR	-	-
7	2635	GSR	PCR	FEA	-
8	2665	GSR	-	FEA	-
9	5517	GSR	PCR	FEA	-
10	5652	GSR	PCR	FEA	PCR
11	5653	-	PCR	FEA	PCR
12	5711	MSA	PCR	FEA	PCR
13	5721	MSA	-	FEA	PCR
14	6123	MSA	MSA	FEA	PCR
15	6133	MSA	MSA	FEA	-
16	6166	MSA	MSA	-	-
17	7004	-	MSA	-	-
18	7006	PP4	MSA	-	-
19	7022	PP4	-	-	-
20	10243	-	-	-	-

Figure 13-2

Stage 1

Program CIO has completed filling the INPUT buffer with source.

Program GSR is about to extract source line 1.

Stage 2

Program GSR has recognised source line 1 and has requested monitor to load program DSA. Program GSR is about to extract source line 4.

Program DSA has just been loaded. It is about to extract source line 1 from processor communication buffer 1.

Processor 1 communication buffer contains only source line 1.

Stage 3

Program GSR has extracted and processed source line 4. It has extracted and is still processing source line 5. It has set the executable statement flag.

Program DSA has completed processing of source line 1. Finding its buffer empty and the executable statement flag set, it checks the status of processor communication buffers 2 and 3 and thereafter requests monitor to load program MSA.

Processor 3 communication buffer contains only source line 4.

Stage 4

Program GSR is still processing source line 5.

Program DSA has dropped itself.

Program MSA is processing source line 4.

All processor communication buffers are empty.

Stage 5

Program GSR has finished processing source lines 5 and 6. It has extracted and is still processing source line 7.
Program MSA has finished processing source line 4 and detected a syntax error. Since its processor communication buffer is empty and there is at least one other program running, it has dropped itself.
Processor 2 communication buffer contains source lines 5 and 6.

Stage 6

Program GSR has finished processing source lines 7 to 10 inclusive. Since it has just completed its eighth insertion into a buffer, it has requested monitor to load program PCR. It is now extracting source line 11.
Program PCR has just been loaded.
Processor 2 communication buffer contains source lines 5, 6, 7, 8, 9 and 10.

Stage 7

Program GSR is still extracting source line 11.
Program PCR has discovered that processor 2 communication buffer contains six entries and that program FEA is not running. It has therefore requested monitor to load program FEA. It is about to check the processor 3 communication buffer.
Program FEA has just been loaded.

Stage 8

Program GSR is processing source line 11.

Program PCR has dropped itself on finding that the processor 3 communication buffer is empty.

Program FEA is processing source line 5.

Processor 2 communication buffer contains source lines 6, 7, 8, 9 and 10.

Stage 9

Program GSR has finished extracting and processing source lines 11, 12, 13, 14, 15, 16, 17 and 18. It split source line 12 into two parts, an expression part was loaded into the processor 2 communication buffer and a statement number list part was loaded into the processor 3 communication buffer. Since it has just completed the sixteenth insertion into a buffer, it has requested monitor to load program PCR into a peripheral processor. It is now extracting source line 19.

Program PCR has just been loaded.

Program FEA has finished processing source lines 5, 6, 7, 8, 9, 10 and 11. It is now processing source line 12.

Processor 2 communication buffer contains only source line 13.

Processor 3 communication buffer contains source lines 12, 14, 15, 16 and 18.

Stage 10

Program GSR has extracted and processed source line 19, thereby detecting the end of the current source subprogram. It has therefore requested monitor to load program PCR for the last time.

Program PCR in processor 2 is about to check the contents of the processor 3 communication buffer, having found that the processor 2 communication buffer contains only one entry and that a copy of program FEA is running.

Program FEA is still processing source line 12.

Program PCR in processor 4 has just been loaded.

Stage 11

Program GSR has dropped itself.

Program PCR in processor 2 is checking the contents of the processor 3 communication buffer.

Program FEA is still processing source line 12.

Program PCR in processor 4 is about to check the contents of the processor 2 communication buffer.

Stage 12

Program PCR in processor 2 has found that the processor 3 communication buffer is non-empty and has therefore requested monitor to load program MSA.

Program MSA has just been loaded on request by program PCR in processor 2.

Program FEA has finished processing source line 12 and is now processing source line 13.

Program PCR in processor 4 has just found that the processor 2 communication buffer is empty.

Stage 13

Program MSA is processing source line 12.

Program PCR has dropped itself from processor 2.

Program FEA is still processing source line 13.

Program PCR in processor 3 is checking the contents of the processor 3 communication buffer.

Processor 3 communication buffer contains source lines 14, 15, 16 and 18.

Stage 14

Program MSA in processor 1 is still processing source line 12.

Program FEA is still processing source line 13.

Program PCR has requested monitor to load another copy of program MSA since the processor 3 communication buffer has four entries which therefore warrants the running of two copies of program MSA but one copy is already loaded.

Program MSA has just been loaded into processor 2.

Stage 15

Program MSA in processor 1 is still processing source line 12.

Program MSA in processor 2 is processing source line 14.

Program FEA has almost finished processing source line 13

Program PCR has dropped itself.

Processor 3 communication buffer contains source lines 15, 16 and 18.

Stage 16

Program MSA in processor 1 is still processing source line 12.

Program MSA in processor 2 is still processing source line 14.

Program FEA has finished processing source line 13. Since its buffer is now empty and there is at least one other program still running, it has dropped itself.

Stage 17

Program MSA in processor 1 has finished processing source lines 12, 16 and 18 and since the communication buffer is now empty and there is at least one other program still running, it has dropped itself.

Program MSA in processor 2 has finished processing source lines 14 and 15 and since the communication buffer is now empty, it is about to scan the processor status table to check whether there are any other processors still running.

Stage 18

Program MSA has requested monitor to load program PP4 since there are no other processors still running.

Program PP4 has just been loaded.

Stage 19

Program PP4 is scanning the DO table.

Program MSA has dropped itself.

Stage 20

Program PP4 has finished producing the listing and has requested program CIO to completely empty the OUTPUT buffer. Knowing that the INPUT buffer is now empty and that the end of the source file had been previously detected, it concludes that there is no more source to be compiled and therefore drops itself.

The listing of the source subprogram in figure 13-1 produced by the system is shown in figure 12-1. The compiler has detected a syntax error in the DO statement which is source line 4 (referred to as line 0003 in the listing). This is due to the requirement that the numeric indexing parameters of DO statements must be non-zero positive integers.

An inspection of figure 13-2 may suggest that a greater degree of multiprocessing would be achieved if the second copy of program MSA were loaded earlier than stage 12, by organising program GSR to call program PCR more frequently. However, since programs FEA and MSA tend to empty their processor communication buffers slightly faster on average than program GSR can fill them, the general effect of loading these programs earlier, is that they tend to drop themselves prematurely. They therefore have to be subsequently reloaded when their processor communication buffers once more contain entries. Since the program loading overhead is not insignificant, such

multiple program loading is to be avoided. Three program PCR loading frequencies were tried, as follows: program PCR loaded after every two, four or eight processor communication buffer insertions. Observation of system behaviour over a large number of test decks, selected at random from computer job input queues, indicated that, of these three possible schemes, the latter is the most satisfactory overall.

14. CONCLUSION

14.0 The Multiprocessing Compiler In Retrospect

The experiment described in this thesis shows that it is possible to multiprocess the compilation task on a set of highly parallel, independent processors. Nevertheless a number of problem areas were brought to light and are listed here again. Possible solutions to some of these are discussed later (see 14.2) as part of an alternate approach.

Because of the critical effect that the declaratives may have on the subsequent course of the compilation, the processing of declarative statements in a parallel environment must be handled with care. It is clearly essential that the source language be so restricted that it is compulsory for all declarative statements to appear before executable statements. Most compiler systems already have this restriction but there are some exceptions, the most notable of which is Control Data Corporation's RUN compiler itself. The latter compiler flags declaratives appearing after the first executable statement with a warning but otherwise accepts them fully so that they will affect the compilation of subsequent source statements. This approach is, however, untenable in a parallel processor system since the order in which statements are processed is indeterminate and indeed may vary from one compilation run to the next. The latter does not apply to the initial set of declaratives which are processed by program DSA strictly in the order

presented. A further requirement for declarative processing is that it must be fully concluded before the analysis of other statement types can commence, but this has been previously discussed (see 8.0). The executable statements can be processed in any order. Only nested DO statements cause a problem in this regard. However, the difficulty is overcome by delaying the checking of the validity of nested DO loops until all statements have been processed. The method has been previously discussed in detail (see 4.2, 5.1.4 and 10.0). Observation of the compilation system whilst running shows that there are seldom more than four programs running concurrently, and that usually this number is three. This is largely due to the fact that programs FEA and MSA generally empty their communication buffers slightly faster than program GSR can fill them. The solution to this would be to run multiple copies of program GSR but this itself generates further problems which have already been discussed (see 4.2 and 14.2).

14.1 The Generation of Object Code

For reasons discussed in Chapter 1, the experimental multiprocessing compiler which was implemented, does not generate object code. However, the design of this system permits the inclusion of this facility with only few modifications. Nevertheless the implementation of these modifications would be troublesome since the present core sizes of programs such as GSR and FEA are already maximal.

Implementation would require the inclusion of an object code table in central memory. Its structure could take the form of a linked list of variable length entries very much like the processor communication buffers. Programs GSR, FEA and MSA would have to be expanded (and possibly split because of the core size difficulties discussed above) to include object code generation. This object code would be entered into the table and would include the corresponding source statement's line number for subsequent reordering purposes. However, at this stage jump addresses could not be generated for statement number references. Instead, program PP4 would need to be altered to have it replace these statement number references with actual addresses. As well, program PP4 would rearrange the contents of the object code table in accordance with the line numbers of the original source subprogram and output it to backing-store in the required format. The size of such an object code table may, however, cause a problem. The simplest approach would be to allow it to be expanded in the usual way by program CTM. Then at the end of the current subprogram, the table is output to backing-store as described above and then, as with all tables, reduced to its original size. Whilst this approach would be satisfactory for relatively small subprograms, large ones would impose an undue core load. An alternative approach would be to cause the system to wind down when the object code

table became almost full and then load a program to dump its contents to disc. Program PP4 would then have to manipulate the generated object code differently depending on whether or not a dump to disc had taken place.

14.2 Alternative Approaches

There are two possible ways in which the running of multiple copies of program GSR could be organised, despite the problems discussed in 4.2.

Firstly, the problem of overshooting the end of the current source subprogram as well as the detection of the occurrence of a large source subprogram could be overcome by locking out access to the INPUT buffer from the time that a copy of program GSR starts extracting the next source statement from it until the type of the statement is determined not to be END. During this period, flags could be set in central memory indicating the occurrence of a large subprogram or the end of subprogram and the actions of all copies of program GSR would be subject to the state of these flags. It is, however, by no means certain that a worthwhile gain in the speed of source statement recognition would be achieved by multiple copies of program GSR operating in this mode due to the high proportion of time during which the INPUT buffer would be inaccessible.

Secondly, these problems could be solved by creating a new program which would handle the extraction of source statements from the INPUT buffer and insert them into a new buffer in central memory, called the processor 0 communication buffer, from which modified versions of

program GSR would extract entries and perform statement recognition in the usual way. This new program would also check for the occurrence of an END statement. This approach has, however, two drawbacks. One is that another peripheral processor and an additional central memory table are required. The other is that the detection of events such as the first header statement (and the resultant loading of program DSA) and the first executable statement would still have to be controlled by the setting and checking of flags in the central memory. To what extent the latter negates the advantages of multiprocessing multiple copies of program GSR could only be determined by further experimentation.

The central memory table access mechanism used cannot ensure that access to tables is granted in the order that requests are made. There is, however, an alternate approach to this problem which would ensure this and would also permit access requests to be granted on a priority basis so that programs such as FEA, which are highly dependent on access to certain tables, for example the name table, would get preferential treatment when required. However, this method requires the use of a dedicated processor for the purpose which would continuously scan the processor status table looking for flags set by the other peripheral processor programs when requesting table access. This dedicated processor program would accept these requests, queue them according to their priority and grant table access accordingly. It is doubtful whether the improvement in the granting of table access could justify the load on the system of a

dedicated processor for the purpose.

14.3 Acknowledgements

The author wishes to express his sincere thanks to Dr. J.G. Sanderson for his guidance in the preparation and presentation of this thesis. Professor F. Hirst's advice was also appreciated. Other thanks are due to the author's wife for the time and effort spent drawing the flow-charts and to Pat Read for her proficient typing.

TABLE 1
CENTRAL PROCESSOR OPERATION CODES

Octal Op. Code	Mnemonic	Address	Comments
BRANCH UNIT			
00	PS		Program stop
01	RJ	K	Return jump to K
02	JP	Bi + K	Jump to Bi + K
030	ZR	Xj K	Jump to K if Xj = 0
031	NZ	Xj K	Jump to K if Xj ≠ 0
032	PL	Xj K	Jump to K if Xj = plus (positive)
033	NG	Xj K	Jump to K if Xj = negative
034	IR	Xj K	Jump to K if Xj is in range
035	OR	Xj K	Jump to K if Xj is out of range
036	DF	Xj K	Jump to K if Xj is definite
037	ID	Xj K	Jump to K if Xj is indefinite
04	EQ	Bi Bj K	Jump to K if Bi = Bj
04	ZR	Bi K	Jump to K if Bi = B0
05	NE	Bi Bj K	Jump to K if Bi ≠ Bj
05	NZ	Bi K	Jump to K if Bi ≠ B0
06	GE	Bi Bj K	Jump to K if Bi ≥ Bj
06	PL	Bi K	Jump to K if Bi ≥ B0
07	LT	Bi Bj K	Jump to K if Bi < Bj
07	NG	Bi K	Jump to K if Bi < B0
BOOLEAN UNIT			
10	BXi	Xj	Transmit Xj to Xi
11	BXi	Xj*Xk	Logical Product of Xj and Xk to Xi
12	BXi	Xj + Xk	Logical sum of Xj and Xk to Xi
13	BXi	Xj - Xk	Logical difference of Xj and Xk to Xi
14	BXi	- Xk	Transmit comp. of Xk to Xi
15	BXi	- Xk*Xj	Logical product of Xj and Xk comp. to Xi
16	BXi	- Xk + Xj	Logical sum of Xj and Xk comp. to Xi
17	BXi	- Xk - Xj	Logical difference of Xj and Xk comp. to Xi
SHIFT UNIT			
20	LXi	jk	Left shift Xi, jk places
21	AXi	jk	Arithmetic right shift Xi, jk places
22	LXi	Bj Xk	Left shift Xk nominally Bj places to Xi
23	AXi	Bj Xk	Arithmetic right shift Xk nominally Bj places to Xi
24	NXi	Bj Xk	Normalize Xk in Xi and Bj
25	ZXi	Bj Xk	Round and normalize Xk in Xi and Bj
26	UXi	Bj Xk	Unpack Xk to Xi and Bj
27	PXi	Bj Xk	Pack Xi from Xk and Bj
43	MXi	jk	Form mask in Xi, jk bits
ADD UNIT			
30	FXi	Xj + Xk	Floating sum of Xj and Xk to Xi
31	FXi	Xj - Xk	Floating difference Xj and Xk to Xi
32	DXi	Xj + Xk	Floating DP sum of Xj and Xk to Xi
33	DXi	Xj - Xk	Floating DP difference of Xj and Xk to Xi
34	RXi	Xj + Xk	Round floating sum of Xj and Xk to Xi
35	RXi	Xj - Xk	Round floating difference of Xj and Xk to Xi

Octal Op. Code	Mnemonic	Address	Comments
LONG ADD UNIT			
36	IXi	Xj + Xk	Integer sum of Xj and Xk to Xi
37	IXi	Xj - Xk	Integer difference of Xj and Xk to Xi
MULTIPLY UNIT			
40	FXi	Xj * Xk	Floating product of Xj and Xk to Xi
41	RXi	Xj * Xk	Round floating product of Xj and Xk to Xi
42	DXi	Xj * Xk	Floating DP product of Xj and Xk to Xi
DIVIDE UNIT			
44	FXi	Xj / Xk	Floating divide Xj by Xk to Xi
45	RXi	Xj / Xk	Round floating divide Xj by Xk to Xi
46	NO		No operation
47	CXi	Xk	Counts in Xi the number of 1's in Xk
INCREMENT UNIT			
50	SAi	Aj + K	Set Ai to Aj + K
50	SAi	Aj - K	Set Ai to Aj + comp. of K
51	SAi	Bj + K	Set Ai to Bj + K
51	SAi	Bj - K	Set Ai to Bj + comp. of K
52	SAi	Xj + K	Set Ai to Xj + K
52	SAi	Xj - K	Set Ai to Xj + comp. of K
53	SAi	Xj + Bk	Set Ai to Xj + Bk
54	SAi	Aj + Bk	Set Ai to Aj + Bk
55	SAi	Aj - Bk	Set Ai to Aj - Bk
56	SAi	Bj + Bk	Set Ai to Bj + Bk
57	SAi	Bj - Bk	Set Ai to Bj - Bk
60	SBi	Aj + K	Set Bi to Aj + K
60	SBi	Aj - K	Set B to Aj + comp. of K
61	SBi	Bj + K	Set Bi to Bj + K
61	SBi	Bj - K	Set Bi to Bj + comp. of K
62	SBi	Xj + K	Set Bi to Xj + K
62	SBi	Xj - K	Set Bi to Xj + comp. of K
63	SBi	Xj + Bk	Set Bi to Xj + Bk
64	SBi	Aj + Bk	Set Bi to Aj + Bk
65	SBi	Aj - Bk	Set Bi to Aj - Bk
66	SBi	Bj + Bk	Set Bi to Bj + Bk
67	SBi	Bj - Bk	Set Bi to Bj - Bk
70	SXi	Aj + K	Set Xi to Aj + K
70	SXi	Aj - K	Set Xi to Aj + comp. of K
71	SXi	Bj + K	Set Xi to Bj + K
71	SXi	Bj - K	Set Xi to Bj + comp. of K
72	SXi	Xj + K	Set Xi to Xj + K
72	SXi	Xj - K	Set Xi to Xj + comp. of K
73	SXi	Xj + Bk	Set Xi to Xj + Bk
74	SXi	Aj + Bk	Set Xi to Aj + Bk
75	SXi	Aj - Bk	Set Xi to Aj - Bk
76	SXi	Bj + Bk	Set Xi to Bj + Bk
77	SXi	Bj - Bk	Set Xi to Bj - Bk

TABLE 2
PERIPHERAL PROCESSOR
OPERATION CODES

Octal Op. Code	Mnemonic	Address	Comments
00	PSN		Pass
01	LJM	m,d	Long jump to m + (d)
02	RJM	m,d	Return jump to m + (d)
03	UJN	d	Unconditional jump d
04	ZJN	d	Zero jump d
05	NJN	d	Nonzero jump d
06	PJN	d	Plus jump d
07	MJN	d	Minus jump d
10	SHN	d	Shift d
11	LMN	d	Logical difference d
12	LPN	d	Logical product d
13	SCN	d	Selective clear d
14	LDN	d	Load d
15	LCN	d	Load complement d
16	ADN	d	Add d
17	SBN	d	Subtract d
20	LDC	dm	Load dm
21	ADC	dm	Add dm
22	LPC	dm	Logical product dm
23	LMC	dm	Logical difference dm
24			Pass
25			Pass
26	EXN		Exchange jump
27	RPN		Read program address
30	LDD	d	Load (d)
31	ADD	d	Add (d)
32	SBD	d	Subtract (d)
33	LMD	d	Logical difference (d)
34	STD	d	Store (d)
35	RAD	d	Replace add (d)
36	AOD	d	Replace add one (d)
37	SOD	d	Replace subtract one (d)
40	LDI	d	Load ((d))
41	ADI	d	Add ((d))
42	SBI	d	Subtract ((d))
43	LMI	d	Logical difference ((d))
44	STI	d	Store ((d))
45	RAI	d	Replace add ((d))
46	AOI	d	Replace add one ((d))
47	SOI	d	Replace subtract one ((d))
50	LDM	m,d	Load (m + (d))
51	ADM	m,d	Add (m + (d))

Octal Op. Code	Mnemonic	Address	Comments
52	SBM	m,d	Subtract (m + (d))
53	LMM	m,d	Logical difference (m + (d))
54	STM	m,d	Store (m + (d))
55	RAM	m,d	Replace add (m + (d))
56	AOM	m,d	Replace add one (m + (d))
57	SOM	m,d	Replace subtract one (m + (d))
60	CRD	d	Central read from (A) to d
61	CRM	m,d	Central read (d) words from (A) to m
62	CWD	d	Central write to (A) from d
63	CWM	m,d	Central write (d) words to (A) from m
64	AJM	m,d	Jump to m if channel d active
65	IJM	m,d	Jump to m if channel d inactive
66	FJM	m,d	Jump to m if channel d full
67	EJM	m,d	Jump to m if channel d empty
70	IAN	d	Input to A from channel d
71	IAM	m,d	Input (A) words to m from channel d
72	OAN	d	Output from A on channel d
73	OAM	m,d	Output (A) words from m on channel d
74	ACN	d	Activate channel d
75	DCN	d	Disconnect channel d
76	FAN	d	Function (A) on channel d
77	FNC	m,d	Function m on channel d

Notation

d	Implies d itself
(d)	Implies the contents of d
((d))	Implies the contents of the location specified by d
m	Implies m itself used as an address
m + (d)	The contents of d are added to m to form an operand (jump address)
(m + (d))	The contents of d are added to m to form the address of the operand
dm	Implies an 18-bit quantity with d as the upper 6 bits and m as the lower 12 bits

APPENDIX 1

FLOYD PRODUCTION LANGUAGE PROGRAM

Under certain circumstances (see 9.0) program FEA checks the syntax of expressions by means of a set of productions for recognizing valid FORTRAN expressions. These productions, as introduced by Floyd (10) and modified by Evans (8), constitute a language, referred to as Floyd Production Language in this thesis, for manipulating symbols in a last-in-first-out stack.

The language consists of statements of the form:

LO: $S_m.....S_1 \rightarrow S'_n.....S'_1$ *GO

where LO is an optional label definition

$S_m.....S_1$ is a string of symbols called the comparison string

$S'_n.....S'_1$ is an optional string of symbols called the replacement string

* is an optional input string operator

GO is a destination label

This statement is executed in the following way:

(i) The comparison string is tested against the stack contents by aligning S_1 with the top of the stack. If a match does not occur, control proceeds to the next statement. Otherwise if a replacement string is present, the matched symbols in the stack are replaced by the symbols of the replacement string. However, if no replacement string is present, the contents of the stack are not changed.

(ii) If * is present, the next input string symbol is inserted into the stack.

(iii) Control proceeds to the statement labelled GO.

The Floyd Production Language program below is stored in coded form within program FEA which simulates its execution. A number of symbols with predefined meanings are used in this program and they are:

- i represents a constant or variable of any type other than LOGICAL
- ℓ represents a LOGICAL constant or variable
- w represents a LOGICAL function
- x represents a function of any type other than LOGICAL
- y represents a parametric function reference (see note XV of 9.1)
- α represents the logical operator .NOT.
- β represents any logical operator other than .NOT.
- γ represents any relational operator
- ↑ represents the exponentiation operator
- $ will match any symbol in the stack
- ' is used to represent both the start and end of the input string

The program is commenced with ' in the stack.

```
                '          *E4
E4:             i          *IO
                ℓ          *T2
                $           EO
LO:             (          *EO
                i          *I1
                ℓ          *I2
                $           X2
EO:             +          *MO
                -          *MO
```

M0:	α			*L0
	(			*E0
	i			*I0
	ℓ			*T2
X2:	x			*X3
	w			*X3
	y			*X1
X1:	x(y,	→	x(	*E0
	x(y)	→	i	*I4
	w(y,	→	w(	*E0
	w(y)	→	ℓ	*I5
	$			ERROR EXIT
X3:	(			*E0
	$			ERROR EXIT
I1:	αiβ	→	g	*M0
	αi)	→	d)	B0
	αi'	→	d'	B0
	$			I3
I2:	αℓβ	→	c	*M0
	αℓ)	→	b)	L1
	αℓ'	→	b'	L1
T2:	ℓβ	→	c	*M0
	cℓ)	→	b)	L1
	cℓ'	→	b'	L1
	ℓ$	→	T$	T1
I0:	iβ	→	g	*M0
	gi$	→	d$	B0

I3:	i$	→	T$	T1
T1:	f↑t$	→	f$	F1
	t$	→	f$	F1
F1:	f↑			*M3
	m*f$	→	m$	M1
	m/f$	→	m$	M1
	f$	→	m$	M1
	$			ERROR EXIT
M1:	m*			*M3
	m/			*M3
	e+m$	→	e$	E1
	e-m$	→	eS	E1
	+m$	→	e$	E1
	-m$	→	e$	E1
	$			ERROR EXIT
E1:	'e'			SUCCESS EXIT - see note 1
	g(e)	→	d	*BO
	α(e)	→	d	*BO
	x(e)	→	i	*I4
	w(e)	→	ℓ	*I5
	(e)			*E2
	e+			*M3
	e-			*M3
	eγ			*E3
	eγe$	→	a$	RO
	x(e,	→	x(	*EO
	w(e,	→	w(	*EO
	$			ERROR EXIT

```
E2:     (e)β  →   g     *MO
        (e)$  →   t$     T1
R0:      'a'              SUCCESS EXIT - see note 2
        x(a)  →   i     *I4
        w(a)  →   ℓ     *I5
        x(a,  →   x(    *E0
        w(a,  →   w(    *E0
         (a)  →   a     *R0
        αa$   →   b$     L1
        ca$   →   b$     L1
         aβ   →   c     *M0
          $               ERROR EXIT
L1:      'b'              SUCCESS EXIT - see note 3
        cb$   →   c$     L1
        α(b)  →   b     *L1
        x(b)  →   i     *I4
        w(b)  →   ℓ     *I5
        x(b,  →   x(    *E0
        w(b,  →   w(    *E0
         (b)  →   b     *L1
         bβ   →   c     *M0
          $               ERROR EXIT
B0:      gd$   →   d$     B0
B1:      dβ   →   g     *M0
        α(d)  →   d     *B0
        x(d)  →   i     *I4
        w(d)  →   ℓ     *I5
        x(d,  →   x(    *E0
```

	w(d,	→	w(	*E0
	(d)	→	d	*B0
	'd'			SUCCESS EXIT - see note 4
	d/	→	m/	*M3
	d*	→	m*	*M3
	d+	→	e+	*M3
	d-	→	e-	*M3
	$			ERROR EXIT
E3:	+			*M3
	-			*M3
M3:	(			*E3
	i	→	t	*T1
	ℓ	→	t	*T1
	x			*X3
	w			*X3
	$			ERROR EXIT
I4:	αi$			I1
	eγi$	→	eγt$	T1
	$			I0
I5:	αℓ$			I2
	eαℓ$	→	eγt$	T1
	$			T2

Notes:

The input string has been successfully reduced to a valid

(1) arithmetic expression

(2) relational expression

(3) logical expression

(4) masking expression.

APPENDIX 2

THE SOURCE LANGUAGE

Statement type	Processed by:
ASSIGN s TO m	MSA
BACKSPACE i	MSA
BLOCK DATA	GSR/DSA
BUFFER IN (u,p)(A,B)	MSA
BUFFER OUT (u,p)(A,B)	MSA
CALL name	FEA
CALL name (p_1,.....,p_n)	FEA
COMMON/I_1/$list_1$/I_2/$list_2$/..../I_n/$list_n$	DSA
COMPLEX list	DSA
n CONTINUE	GSR
DATA I_1/$list_1$/,I_2/$list_2$/,.....I_n/$list_n$/	DSA
DECODE (c,n,v)list	MSA
DIMENSION list	DSA
DO n i = m_1,m_2,m_3	MSA
DO n i = m_1,m_2	MSA
DOUBLE list	DSA
DOUBLE PRECISION list	DSA
ENCODE (c,n,v)list	MSA
END	GSR
END name	GSR
ENDFILE i	MSA
ENTRY name	MSA
EQUIVALENCE (A,B),(A1,B1),...	DSA
EXTERNAL list	DSA

Statement Type	Processed by:
FORMAT ($spec_1$,$spec_2$,...)	GSR/MSA
FUNCTION name (p_1,....,p_n)	GSR/DSA
type FUNCTION name (p_1,....,p_n)	GSR/DSA
GOTO n	MSA
GOTO m	MSA
GOTO (n_1,....,n_m),i	MSA
IF(a)n_1,n_2,n_3	GSR/FEA/MSA
IF(ℓ)n_1,n_2	GSR/FEA/MSA
IF(ℓ)s	GSR/FEA
IF(ENDFILE i)n_1,n_2	GSR/MSA
IF(EOF,i)n_1,n_2	GSR/MSA
IF(IOCHECK,i)n_1,n_2	GSR/MSA
IF(UNIT,i)n_1,n_2,n_3,n_4	GSR/MSA
INTEGER list	DSA
LOGICAL list	DSA
NAMELIST/name/list....	DSA
PAUSE	MSA
PAUSE n	MSA
PRINT n,list	MSA
PROGRAM name (f_1,....,f_n)	GSR/DSA
PUNCH n,list	MSA

Statement Type	Processed by:
READ n,list	MSA
READ (i,n)list	MSA
READ INPUT TAPE i,n,list	MSA
READ (i)list	MSA
READ TAPE i,list	MSA
REAL list	DSA
RETURN	GSR
REWIND i	MSA
SENSE LIGHT i	MSA
STOP	MSA
STOP n	MSA
SUBROUTINE name (p_1,....,p_n)	GSR/DSA
TYPE COMPLEX list	DSA
TYPE DOUBLE list	DSA
TYPE INTEGER list	DSA
TYPE LOGICAL list	DSA
TYPE REAL list	DSA
WRITE (i,n) list	MSA
WRITE OUTPUT TAPE i,n, list	MSA
WRITE (i) list	MSA
WRITE TAPE i, list	MSA
A = arithmetic expression	FEA
L = logical/relational expression	FEA
M = masking expression	FEA
name (p_1,p_2,...,p_n) = expression	FEA

APPENDIX 3

THE COMPILATION ERROR NUMBERS, CODES AND MESSAGES

ERROR NUMBER	TWO-LETTER CODE	ERROR MESSAGE TEXT
1	AC	ARGUMENT COUNT TOO HIGH
2	AE	ARITHMETIC STATEMENT FUNCTION CALLS ITSELF
3	AF	ARITHMETIC STATEMENT FUNCTION ERROR
4	AL	SYNTAX ERROR IN ARGUMENT LIST
5	AS	SYNTAX ERROR IN ASSIGNMENT STATEMENT
6	BC	SYNTAX ERROR IN BOOLEAN CONSTANT
7	CB	LABELLED COMMON BLOCKS EXCEED MAXIMUM OF 61
8	CD	VARIABLE DUPLICATED IN A COMMON REGION
9	CE	VARIABLES ASSIGNED TO COMMON ARE IMPROPERLY EQUIVALENCED
10	CL	SYNTAX ERROR IN CALL STATEMENT
11	CM	SYNTAX ERROR IN COMMON STATEMENT
12	CN	TOO MANY CONTINUATION CARDS
13	CO	COMMON STORAGE EXCEEDED
14	CT	CONTINUE STATEMENT IS MISSING A STATEMENT NUMBER
15	DA	DUPLICATE ARGUMENTS IN A FUNCTION DEFINITION STATEMENT
16	DB	ARRAY SIZE OUT OF RANGE
17	DC	SYNTAX ERROR IN A DECIMAL CONSTANT
18	DD	VARIABLE BEING DIMENSIONED HAS BEEN PREVIOUSLY DIMENSIONED
19	DF	DUPLICATE FUNCTION NAME
20	DI	DO TERMINATOR PREVIOUSLY DEFINED
21	DL	DECLARATIVE APPEARS AFTER FIRST EXECUTABLE STATEMENT

ERROR NUMBER	TWO-LETTER CODE	ERROR MESSAGE TEXT
22	DM	SYNTAX ERROR IN DIMENSION STATEMENT
23	DN	ILLEGAL DO TERMINATOR
24	DO	SYNTAX ERROR IN A DO STATEMENT
25	DP	MULTIPLY DEFINED STATEMENT NUMBER
26	DR	DATA RANGE ERROR
27	DS	UNDEFINED STATEMENT NUMBER IN A DO LOOP
28	DT	SYNTAX ERROR IN DATA STATEMENT
29	DU	ATTEMPT HAS BEEN MADE TO PRESTORE VALUE INTO BLANK COMMON
30	EC	CONTRADICTION IN EQUIVALENCE STATEMENT
31	EF	END OF FILE CARD ENCOUNTERED, END CARD ASSUMED
32	EM	SYNTAX ERROR IN INDICATED EXPONENTIATION
33	EQ	SYNTAX ERROR IN EQUIVALENCE STATEMENT
34	EX	SYNTAX ERROR IN EXPONENT
35	FA	FUNCTION HAS NO ARGUMENT
36	FL	SYNTAX ERROR IN EXTERNAL OR F-TYPE STATEMENT
37	FM	UNRECOGNISABLE STATEMENT
38	FN	NO STATEMENT NUMBER FOR FORMAT STATEMENT
39	FS	ERROR IN SPECIFICATION PORTION OF FORMAT STATEMENT
40	FT	SYNTAX ERROR IN FUNCTION TYPE STATEMENT
41	GF	FORMAT NUMBER REFERENCED BY CONTROL STATEMENT
42	GO	SYNTAX ERROR IN A GO TO STATEMENT

ERROR NUMBER	TWO-LETTER CODE	ERROR MESSAGE TEXT
43	IC	CHARACTER NOT IN FORTRAN CHARACTER SET
44	ID	IMPROPERLY NESTED DO LOOPS
45	IF	SYNTAX ERROR IN AN IF STATEMENT
46	IL	SYNTAX ERROR IN AN INDEXED LIST OF I/O STATEMENT
47	IN	ILLEGAL FUNCTION NAME
48	IO	ILLEGAL I/O DESIGNATOR
49	IT	ILLEGAL TRANSFER TO DO TERMINATOR
50	LN	NAMELIST ERROR
51	LP	EXPONENTIATION TO A LOGICAL POWER
52	LS	SYNTAX ERROR IN INPUT/OUTPUT LIST
53	MA	MISUSED SUBROUTINE ARGUMENT IN AN EQUIVALENCE STATEMENT
54	MO	MEMORY OVERFLOW, FIELD LENGTH TOO SHORT
55	MS	UNDEFINED STATEMENT NUMBER
56	NC	SUBROUTINE OR FUNCTION NAME CONFLICTS WITH A PRIOR USAGE
57	NM	IMPROPER HEADER CARD
58	NO	NO OBJECT CODE GENERATED
59	NP	NO PATH TO THIS STATEMENT
60	PM	FUNCTION PARAMETER MODE INCONSISTENCY
61	PN	UNBALANCED PARENTHESIS
62	PT	SYNTAX ERROR IN AN ENTRY STATEMENT
63	RN	SYNTAX ERROR IN A RETURN STATEMENT
64	SB	ERROR IN AN ARRAY SUBSCRIPT
65	SE	SYNTAX ERROR IN SENSE STATEMENT
66	SF	FIELD LENGTH OF ROUTINE BEING COMPILED EXCEEDS THE SPECIFIED FIELD LENGTH

ERROR NUMBER	TWO-LETTER CODE	ERROR MESSAGE TEXT
67	SM	SYNTAX ERROR IN STATEMENT NUMBER
68	SN	ILLEGAL CHARACTER IN STATEMENT NUMBER USAGE
69	SY	SYSTEMS ERROR IN FORTRAN COMPILER
70	TM	SUBROUTINE HAS MORE THAN 60 ARGUMENTS
71	TT	VARIABLE GIVEN CONFLICTING TYPES
72	TN	PROGRAM HAS MORE THAN 50 ARGUMENTS
73	TY	SYNTAX ERROR IN A TYPE STATEMENT
74	UA	REFERENCE MADE TO AN AS YET UNDIMENSIONED ARRAY
75	UE	LOGICAL UNIT NUMBER IS NOT AN INTEGER
76	VC	VARIABLE NAME CONFLICTS WITH A PRIOR USAGE
77	VD	ARRAY WHOSE DIMENSIONS ARE ARGUMENTS TO THE SUBROUTINE OR FUNCTION HAS BEEN MISUSED
78	XF	SYNTAX ERROR IN THE EXPRESSIONS BEING PROCESSED
79	ZY	SYSTEM ERROR - UNKNOWN TWO-LETTER CODE
80	DJ	SAME DO LOOP CONTROL VARIABLE USED MORE THAN ONCE IN A SET OF NESTED DO'S

REFERENCES

1 Arnold, J.S., Cosey, D.P., and McKinstry, R.H. Design of tightly-coupled multiprocessing programming. IBM Systems J. 13, 1 (1973), 60-87.

2 Chen, Y.E., and Epley, D.R. Memory requirements in a multiprocessing environment. J. ACM 19, 1 (Jan. 1972), 57-69.

3 Control Data Corporation. Control Data 6400/6500/6600 Computer Systems FORTRAN reference manual, publication no. 60174900.

4 Control Data Corporation. Control Data 6400/6500/6600 Computer Systems SCOPE reference manual, publication no. 60189400.

5 Dijkstra, E.W. Co-operating sequential processes. In Programming languages. Genuys, F. (Ed.), Academic Press, London, 1968, 43-112.

6 Dijkstra, E.W. Hierarchical ordering of sequential processes. In Operating systems techniques. Hoare, C.A.R. and Perrot, R.N. (Eds.), Academic Press, London, 1972, 72-93.

7 Ellis, C.A. Parallel compiling techniques. Proc. ACM 1971 Annual Conf., ACM, New York, 508-519.

8 Evans, A. An Algol 60 compiler. Proc. ACM 1963 National Conf., ACM, New York.

9 Firestone, R.M. Parallel programming : operational model and detection of parallelism. New York Uni., New York, 1971.

10 Floyd, R. A descriptive language for symbol manipulation. J. ACM 8, 4 (Oct. 1961), 579-584.

11 Gonzales, M.J. Jr., and Ramamoorthy, C.V. Parallel task execution in a decentralized system. IEEE Trans. Computers, C-21, 12 (Dec. 1972), 1310-1322.

12 Hill, J.C. Synchronizing processors with memory-content-generated interrupts. Comm. ACM 16, 6 (June 1973), 350-351.

13 Kuck, D.J. et al. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. IEEE Trans. Computers, C-21, 12 (Dec. 1972), 1293-1310.

14 Lamport, L. The parallel execution of DO loops. Comm. ACM 17, 2 (Feb. 1974), 83-93.

15 Lorin, H. Parallelism in hardware and software : real and apparent concurrency. Prentice-Hall Inc., Englewood Cliffs, N.J., 1972, 274-276, 311-317.

16 Ramamoorthy, C.V., Chandy, K.M., and Gonzales, M.J. Jr. Optimal scheduling strategies in a multiprocessor system. IEEE Trans. Computers, C-21, 2 (Feb. 1972), 137-146.

17 Shoshani, A., and Bernstein, A.J. Synchronisation in a parallel-accessed data base. Comm. ACM 12, 11 (Nov. 1969), 604-607.

18 Squire, J.S. A translation algorithm for a multi-processor computer. Unpublished paper, Dept. of Electrical Engineering, Uni. of Michigan (1963).

19 Stone, H.S. Problems of parallel computation. In complexity of sequential and parallel numerical algorithms. Traub, J.F. (Ed.), Academic Press, New York, 1973, 1-16.

20 Zelkowitz, M. Interrupt driven programming. Comm. ACM 14, 6 (June 1971), 417-418.