campillo box 5

# Forward into the '90s

Donald E. Knuth
Grand Wizard of TEX

TEX Users Group Conference, College Station, Texas, June 18–20, 1990

LRDC

The compiler "was beautifully designed, very poorly documented, and patched at the machine level so we couldn't recompile it or reassemble it."

– Betty Holberton
from Hist Comp 20 (98) 18.

cmr17 scaled 20000 for a 15 dpi "writes-light-grey" device
Printed with TEX and TEXrox on a Xerox 9790 at Texas A&M University

2226

# LARC

# SCIENTIFIC

# COMPILER

## INTERNAL SYSTEMS MANUAL

~~Goodwin~~ KNUTH

# LARC

# SCIENTIFIC

# COMPILER

## INTERNAL SYSTEMS MANUAL

The compiler described herein was developed by Computer Sciences Corporation, Palos Verdes, California, under contract with Remington Rand Univac, Division of Sperry Rand Corporation.

Acknowledgment is given below to those members of the CSC staff whose efforts in design and implementation resulted in the system described herein.

| | |
|---|---|
| Joel D. Erdwinn | (Phases VI and VIII) |
| David E. Ferguson | (Preliminary Design, Phase VII, and Sorting System) |
| Louis Gatt | (Project Director) |
| Reginald E. Martin | (Phase I and VII) |
| Jack P. Middlekauff | (FILE Program and I/O Routines) |
| Hayden T. Richards | (Phases IV, V, and VI) |
| David W. Roberts | (Phases II, III, and IX) |

# INTRODUCTION

The Larc Scientific Compiler (LSC) is herein described by prose as well as by detailed flow charts. The major function of each phase is explained. Following the explanation, flow charts for that phase will immediately follow.

The peripheral programs associated with LSC are also described (See LSCN). In that section a description of the input and output system available to the object program is contained in the form of prose and flow charts.

During implementation, special attention was given to describing the coding of LSC with comments. These comments appear in the parallel code edits of the entire LSC system and should be helpful in going through the compiler.

# TABLE OF CONTENTS


| | Page |
|---|---|
| General Flow of Control in LSC | 0.0 |
| General Flow of Data in LSC | 0.2.1 |
| Input and Output of LSC | 0.3 |
| LSCN | 0.4 |
| I. Main Control | 0.4 |
| II. The Communication Region | 0.4 |
| III. The Sorting System | 0.5 |
| IV. LSCN Diagnostic System | 0.5 |
| Sorting System Flow Charts | 0.7 |
| File Program | 0.16 |
| File Program Flow Charts | 0.24 |
| Object Program Input-Output Routines | 0.31 |
| Input-Output Routines Flow Charts | 0.34 |
| Phase I | |
| General Description | 1.0 |
| Sequence Counters | 1.4 |
| Item Formats | 1.8 |
| Identification Digits Assignment | 1.21 |
| Mode Word | 1.23 |
| Flow Charts | 1.28 |
| Phase II | |
| General Description | 2.0 |
| Item Formats | 2.3 |
| Flow Charts | 2.6 |

| | Page |
|---|---|
| Phase III | |
| General Description | 3.0 |
| Output to File 4 | 3.8 |
| Output to File 91 | 3.17 |
| Flow Charts | 3.19 |
| Phase IV | |
| General Description | 4.0 |
| Input and Output | 4.2 |
| Treatment of: | |
| FUNCTION and SUBROUTINE | 4.3 |
| LIBRARY CALLS | 4.3 |
| PARAMETERS | 4.3 |
| SIMPLE VARIABLES | 4.4 |
| DIMENSION | 4.5 |
| EQUIVALENCE and COMMON | 4.5 |
| CONSTANT | 4.7 |
| ARITHMETIC STATEMENT FUNCTIONS | 4.7 |
| EXECUTABLE STATEMENTS | 4.7 |
| Subroutines | 4.14 |
| Flow Charts | 4.19 |
| Phase V | |
| General Description | 5.0 |
| Input | 5.0 |
| a - Rename Variables and Constant Subscripts | 5.1 |
| b - Number Subscript Terms | 5.2 |
| c - File Loops by Level | 5.3 |
| d - Unpack Terms | 5.4 |
| e - Form Non-constant Counts | 5.4 |
| f - Form Non-constant Initial and Incremental Multiples | 5.4 |
| g - Form Subscripts | 5.5 |

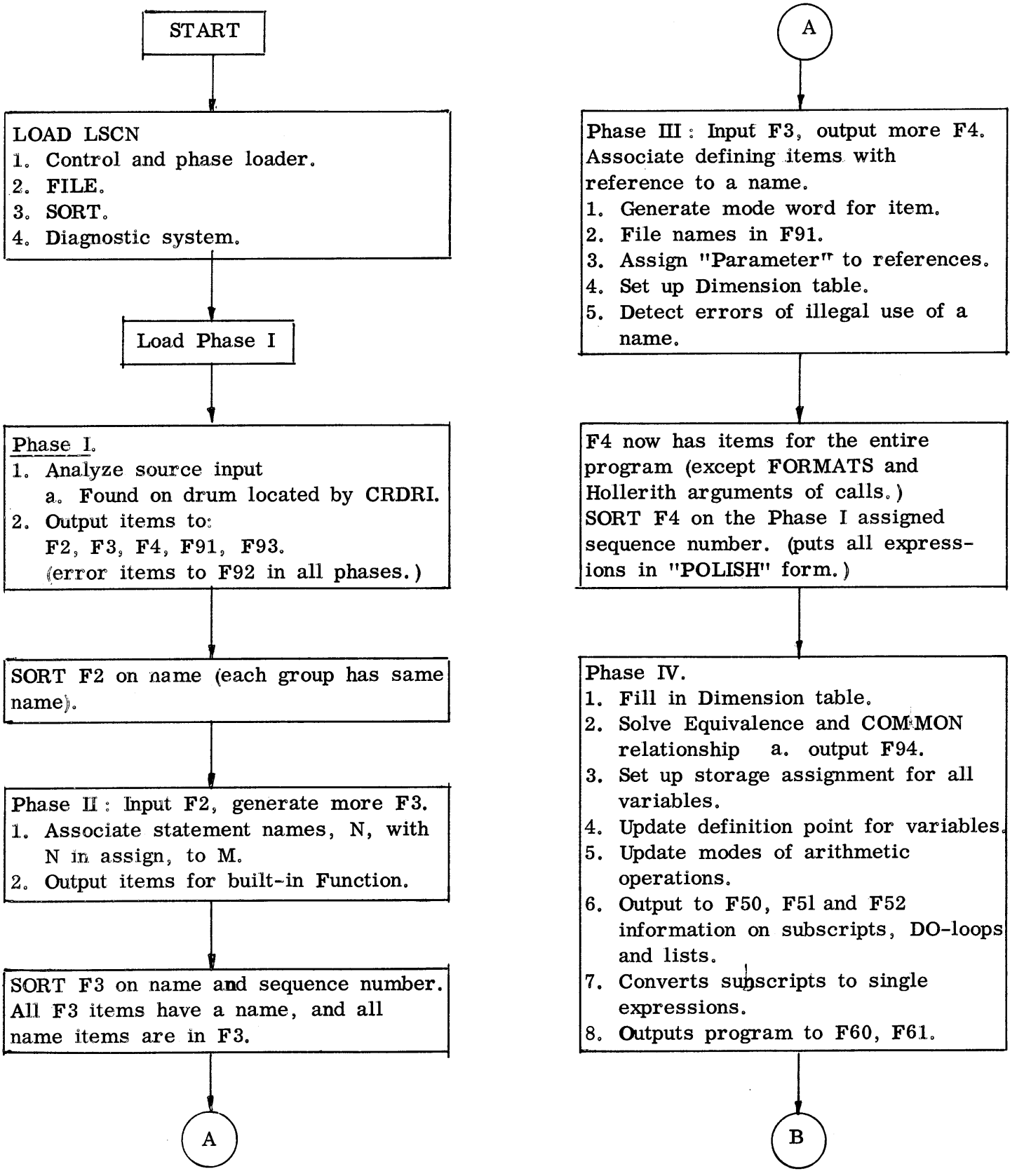
| | Page |
|---|---|
| Special Subroutines | 5.7 |
| Output to File 61 | 5.9 |
| Format of Phase V Internal Files | 5.12 |
| Flow Charts | 5.13 |
| **Phase VI** | |
| General Description | 6.0 |
| First Phase VI - Pass | 6.1 |
| Every Phase VI - Pass | 6.2 |
| Output | 6.3 |
| Flow Charts | 6.4 |
| **Phase VII** | |
| General Description | 7.0 |
| Generator | 7.1 |
| Successor Item Generation | 7.6 |
| Generation of Arithmetic Expressions | 7.7 |
| Examples of Object Code | 7.11 |
| Library Routines | 7.14 |
| Flow Charts | 7.15 |
| **Phase VIII** | |
| a - Backward Scan | 8.0 |
| b - Forward Scan | 8.2 |
| c - Build Flow Tables | 8.10 |
| d - Index Analysis | 8.11 |
| e - Produce F and S Commands | 8.13 |
| Item Formats | 8.15 |
| Flow Charts | 8.27 |

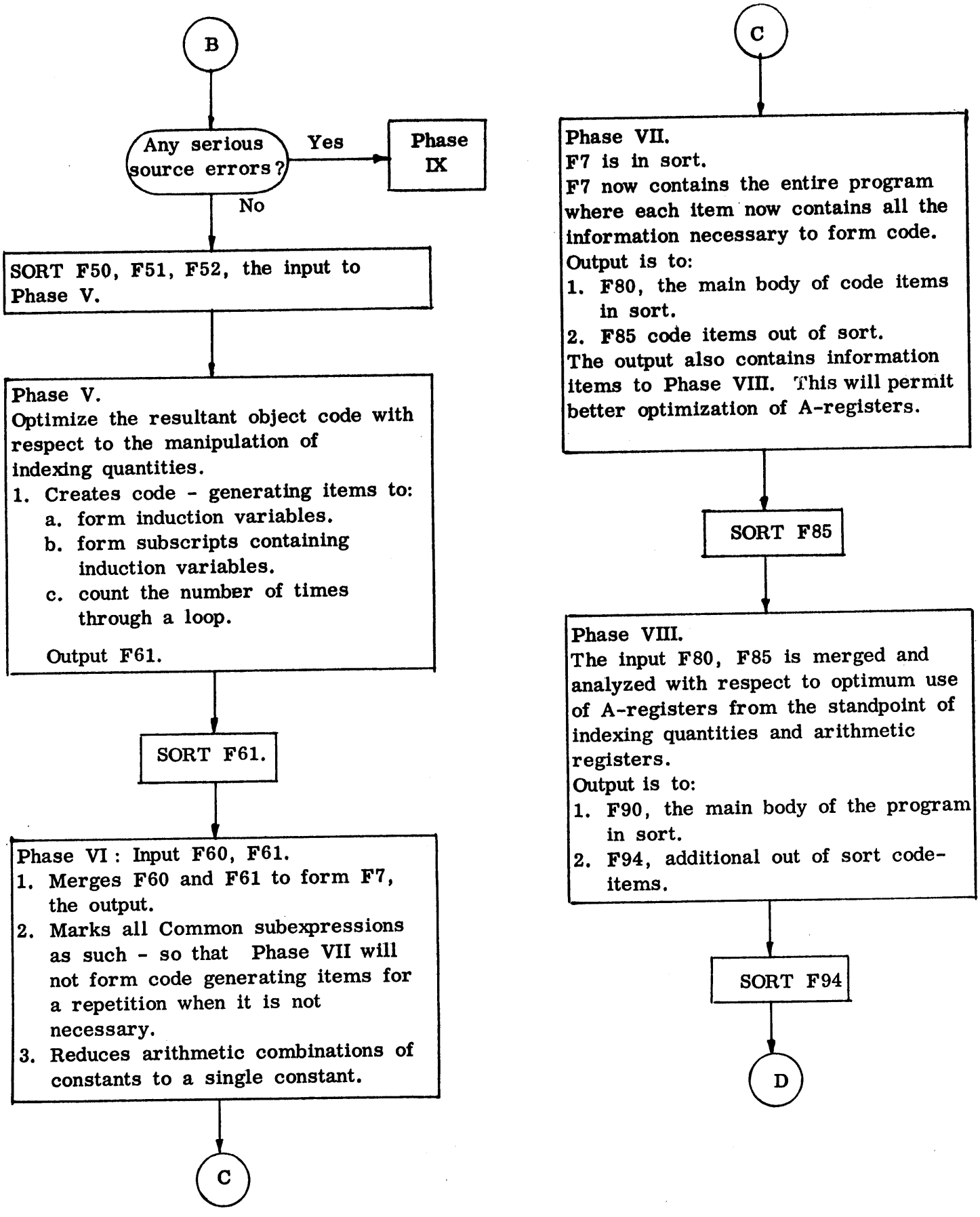
Page

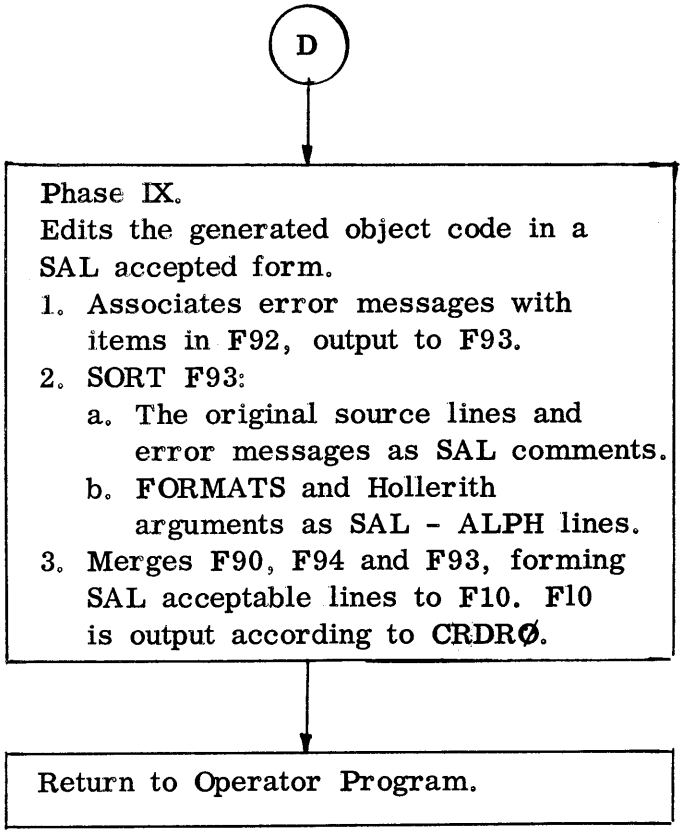
Phase IX


| | |
|---|---|
| General Description | 9.0 |
| a - Output Error Messages | 9.1 |
| b - Output of LSC | 9.2 |
| Name Generation | 9.4 |
| File 91 | 9.7 |
| File 92 | 9.8 |
| File 94 Items from Phase IV | 9.10 |
| File 90 and 94 Items from Phase VIII | 9.13 |
| Flow Charts | 9.19 |

# GENERAL FLOW OF CONTROL IN LSC.

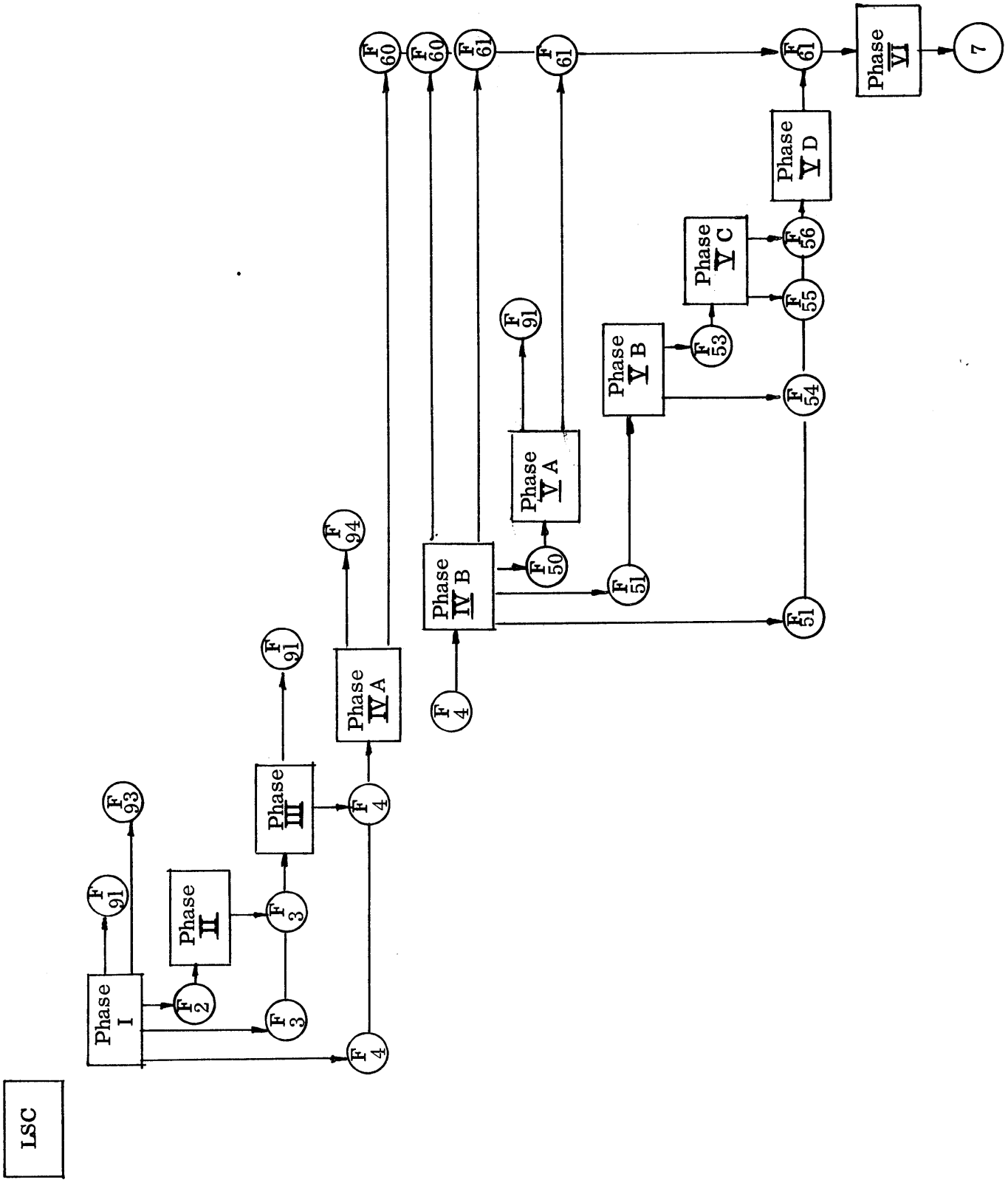



B
Any serious source errors?
Yes
Phase IX
No
SORT F50, F51, F52, the input to Phase V.
Phase V.
Optimize the resultant object code with respect to the manipulation of indexing quantities.
1. Creates code - generating items to:
a. form induction variables.
b. form subscripts containing induction variables.
c. count the number of times through a loop.
Output F61.
SORT F61.
Phase VI : Input F60, F61.
1. Merges F60 and F61 to form F7, the output.
2. Marks all Common subexpressions as such - so that Phase VII will not form code generating items for a repetition when it is not necessary.
3. Reduces arithmetic combinations of constants to a single constant.
C
C
Phase VII.
F7 is in sort.
F7 now contains the entire program where each item now contains all the information necessary to form code.
Output is to:
1. F80, the main body of code items in sort.
2. F85 code items out of sort.
The output also contains information items to Phase VIII. This will permit better optimization of A-registers.
SORT F85
Phase VIII.
The input F80, F85 is merged and analyzed with respect to optimum use of A-registers from the standpoint of indexing quantities and arithmetic registers.
Output is to:
1. F90, the main body of the program in sort.
2. F94, additional out of sort code-items.
SORT F94
D

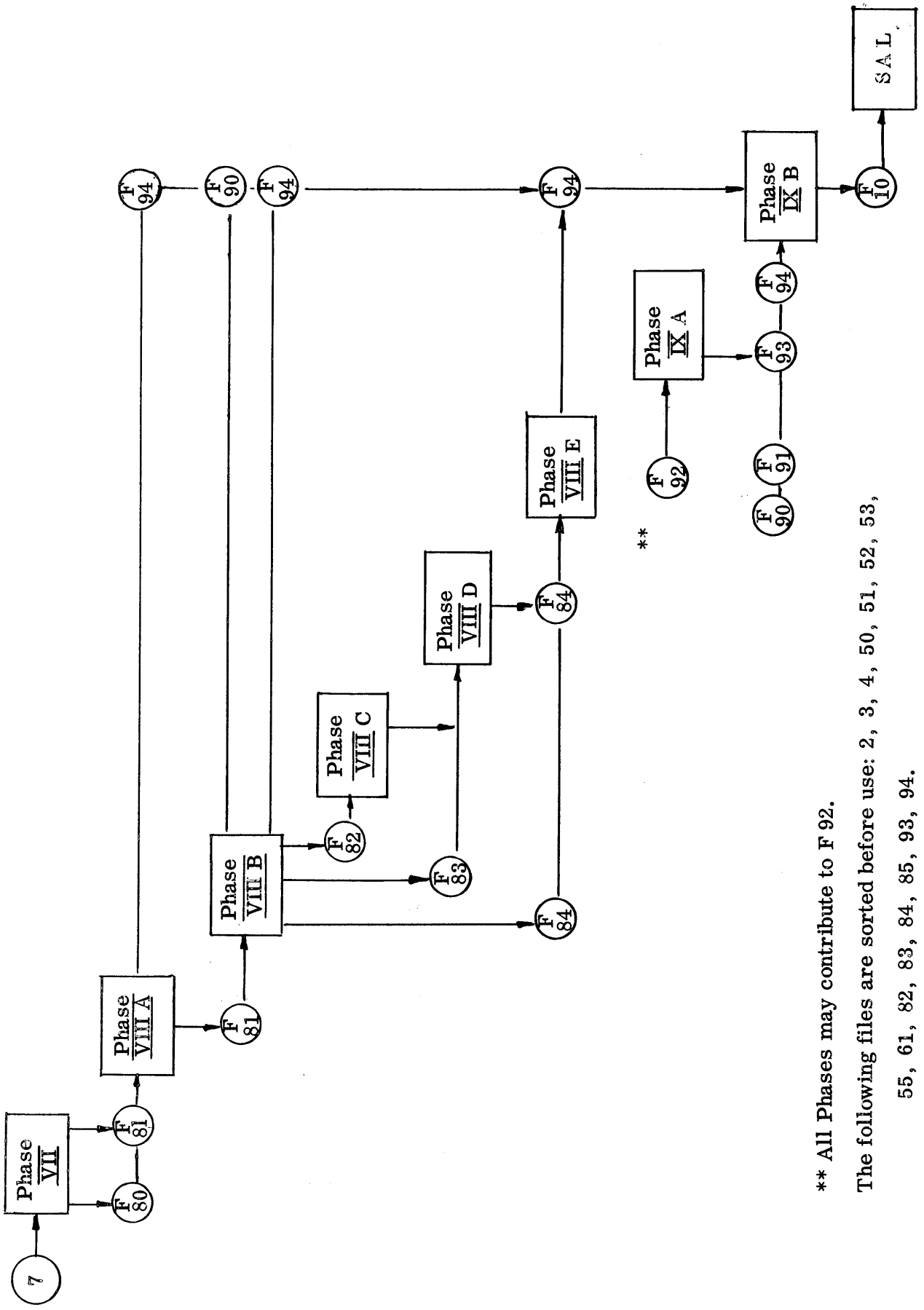

D
Phase IX.
Edits the generated object code in a SAL accepted form.
1. Associates error messages with items in F92, output to F93.
2. SORT F93:
a. The original source lines and error messages as SAL comments.
b. FORMATS and Hollerith arguments as SAL - ALPH lines.
3. Merges F90, F94 and F93, forming SAL acceptable lines to F10. F10 is output according to CRDRØ.
Return to Operator Program.

# LARC SCIENTIFIC COMPILER.

## DATA FLOW IN LSC.



DATA FLOW IN LSC. (continued)




** All Phases may contribute to F 92.
The following files are sorted before use: 2, 3, 4, 50, 51, 52, 53, 55, 61, 82, 83, 84, 85, 93, 94.

INPUT AND OUTPUT OF LSC.

The input described in the Programmer's Reference Manual, is given to LSC either on tape or on a specified drum. The Communication Region (CR) contains a flag, CRT, which is zero or non-zero.

1. If CRT $\neq 0$ input is on tape 11 in alphanumeric form, 2 lines per blockette.
2. If CRT $= 0$ input is on a drum. The starting sector, band, and drum numbers are specified in CRDRI.

During the operation of LSC, seven drums must be made available for temporary storage of generated files, and for the Sort-Merge program. The seven drums must have consecutive drum numbers, the first of which has drum number:

1+(CRDRF)

The input drum, specified by CRDRI, may be one of the seven temporary drums if it is equivalent to : (CRDRF) + 1 , (CRDRF) + 2., or (CRDRF) + 6.

The output of LSC is always output on a drum; however, it may also be output on tape or the on-line printer. The output drum is specified by CRDRO, by specifying the starting sector and band of a drum. CRDRO must not be any of the temporary drums. The output itself is more fully described in the Phase IX section of this report.

If CRHSP is non zero, the output is also generated for the on-line high speed printer or a tape. This choice is specified by setting CROUT to either zero or the appropriate tape servo number. For example:

CRHSP $\neq 0$

and CROUT = 21 output is sent to drum and on tape servo 21. When the output is generated for the high speed on-line printer or for tape, an additional line is written, which contains an internal sequence number as a SAL comment. The tape so generated may be used as input to SAL.

# LSCN

Compilation of a source program begins by bringing into memory that portion of LSC labeled LSCN. This will consist of:

1. The LSC Main Control and drum loader (Main Control)
2. The Communication region.
3. The sorting system
4. The diagnostic system
5. The FILE program

## I. Main Control

When LSCN is read into memory, a word in the communication region, CRDLSC, contains a define-drum summary order. This summary order is issued so that the phases of LSC may be read in relative to the definition contained in CRDLSC. LSCN, itself, may be on any other drum, or it may be loaded from tape. When the phases are read by LSCN, they are assumed to be on drum according to the method of loading defined by the loading program LSCLDR (i.e., each segment of a phase occupies an integral number of consecutive sectors, with the (n + 1) st segment beginning immediately after the $n^{th}$ segment).

When each phase is loaded into memory, control is transferred to it via:

```
TB      A1      Phase n
```

When each phase receives control, it displays in the 5-digit display register: ++n++ and sets up the diagnostic control in CRCRUD (see below - LSCN Diagnostic). Prior to the exit, the phase will display in the 5DD: --(n + 1) --. Upon returning from Phase IX, LSCN will return control to the master operating routine.

## II. The Communication Region

The Communication Region in LSC has various functions:

1. Permits altering LSC parameters to produce a different object code.
2. Allows communication from one phase of LSC to another.

The Communication Region is located 762 words from the start of LSC. The SAL parallel code edit of LSCN contains a detailed description of every item in this region; hence, the entire list is not repeated here. Throughout this report, however, various quantities in the Communication Region are mentioned and described; each symbol in the Communication Region begins with "CR".

## III. The Sorting System

Contained in LSC is a sorting system which is entered via:

```
TB    1    PRESRT
SK    K      Fn
```

Where K represents the length of the sort key in number of 2-digit pairs and Fn is the file to be sorted. The flow chart of this section is contained herein; however, it is worth pointing out a general characteristic of the system. First, every file that is to be sorted has been previously "Closed" or "Released" via the File program. A test is made in PRESRT to see if the entire file is in memory (i.e., entirely contained in its double buffer). If it is, then only an internal sort of the file is made and left in the buffer. If the file is on drum, it may still be small enough for only a simple high-speed internal sort. If it is less than 12,500 words, then the entire file is read in, sorted, and written back on drum. If the file is greater than 12,500 words, then a more complicated and more time-consuming process takes place. This is described in the Sort Merge flow charts. However, in every case, the same internal sort program is used.

The internal sort is a left-to-right radix sort where the base is 100; i.e., two digits constitutes one character.

## IV. LSCN Diagnostic System

Built into the control of LSC is a diagnostic system designed to simplify the obtaining of pertinent information should a difficulty arise while performing a compilation. This system includes A-register dumps, selective memory dumps, and dumps of sorted and unsorted files. LSC will produce the output on-line or on a tape for off-line printing according to the setting of CROUT: (See Communication Region).

There are two methods of taking action depending on whether:

1. LSC runs through to completion with incorrect results or
2. LSC fails to go to completion.

Case I

When LSC runs through completion with erroneous results, the files may be output to give all the necessary information. LSC will output all necessary files by setting:

CRFDMP = 0

CRFDMP is located at LSC + 768. If it is suspected that files produced by Phases M to N are in error, then only those files may be output by setting

CRFDMP = 000 000 000 0 N M

Case II

If LSC fails to complete a compilation, the operator should cause the computer to transfer control to:

CRCRUD = LSC + 794

This will cause a selected dump of memory to be output according to the phase currently in operation as well as all important files. If LSC is in an infinite loop, the operator should first bring the computer to a stop and force an A-register dump before transferring control to LSC + 794.

# LARC SCIENTIFIC COMPILER

## SORT and PRINT FILES



PRESRT

Start
No registers are saved.

Calling Sequence

| | | | |
|---|---|---|---|
| α | TB | 1 | PRESRT |
| α+1 | op | A | Fn |
| α+2 | return | | |

Save return.
SEXIT

A : 0 — = → CRFDMP$_1$ : ph.no (CRPH) — > → α+2

A : 0 — > → CRFDMP$_1$ : ph.no (CRPH) — ≤ → CRFDMP$_2$ : CRPH — < → α+2

A : 0 — ≥ → CRFDMP$_2$ : CRPH

Set up Presort call seq. for File, Fn.

NWFL : 0 — = → No sort, try print one line.

NWFL : 0 — > → Is file on drum? — Yes → CNMOVE

Is file on drum? — No → A : 0 — = → No sort, try print one line.

A : 0 — ≠ → Sort file in buffer. (no drum activity) → CNPRFL

No sort, try print one line. → CNPRFL

CNPRFL → α+2

CNMOVE

File is on drum.

op : SK — ≠ → op : SK

op : SK — = → SAVEMM

SAVEMM
Save on CRDRØ :
MIN(12500, (NWFL+2499)/2500) of memory.

NWFL : 12500 — > → PGSRT2

NWFL : 12500 — ≤ → READFL

READFL
Read entire file in memory.

A : 0 — ≠ → SORT entire file in memory. → Write entire file on drum.

A : 0 — = → Write entire file on drum.

Write entire file on drum. → CNPRFL

GPR → CNPRFL

CNPRFL → op : SK — ≠ → α+2

op : SK — = → Restore memory. → α+2

Symbols.

CRFDMP$_1$ = right most digit of flag word CRFDMP

CRFDMP$_2$ = second right most digit of CRFDMP

NWFL = no. of words in file, Fn

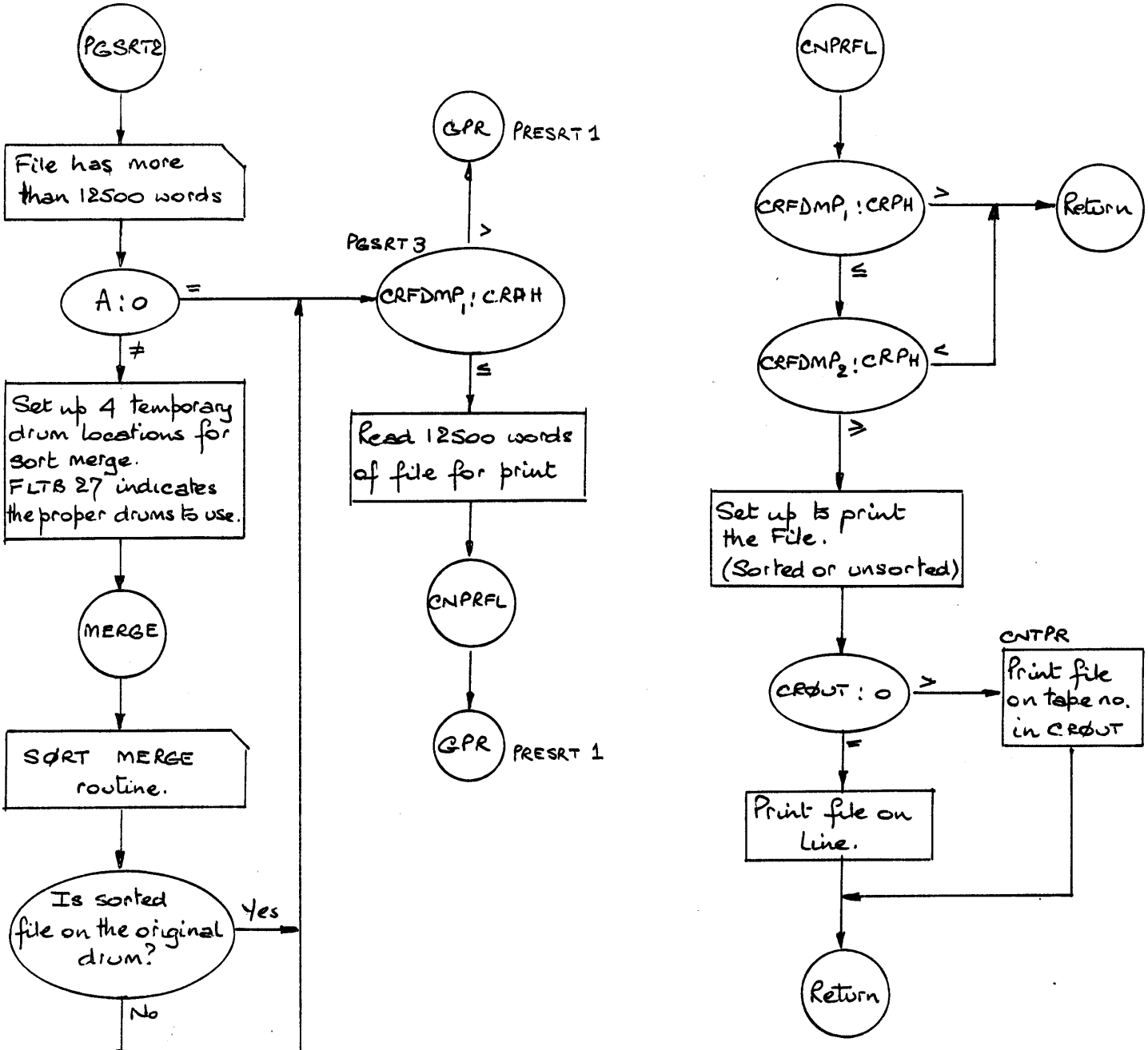
NIT = no. of items in file Fn

CRØUT = 0 or tape no. to use for off line printing.

# LARC SCIENTIFIC COMPILER

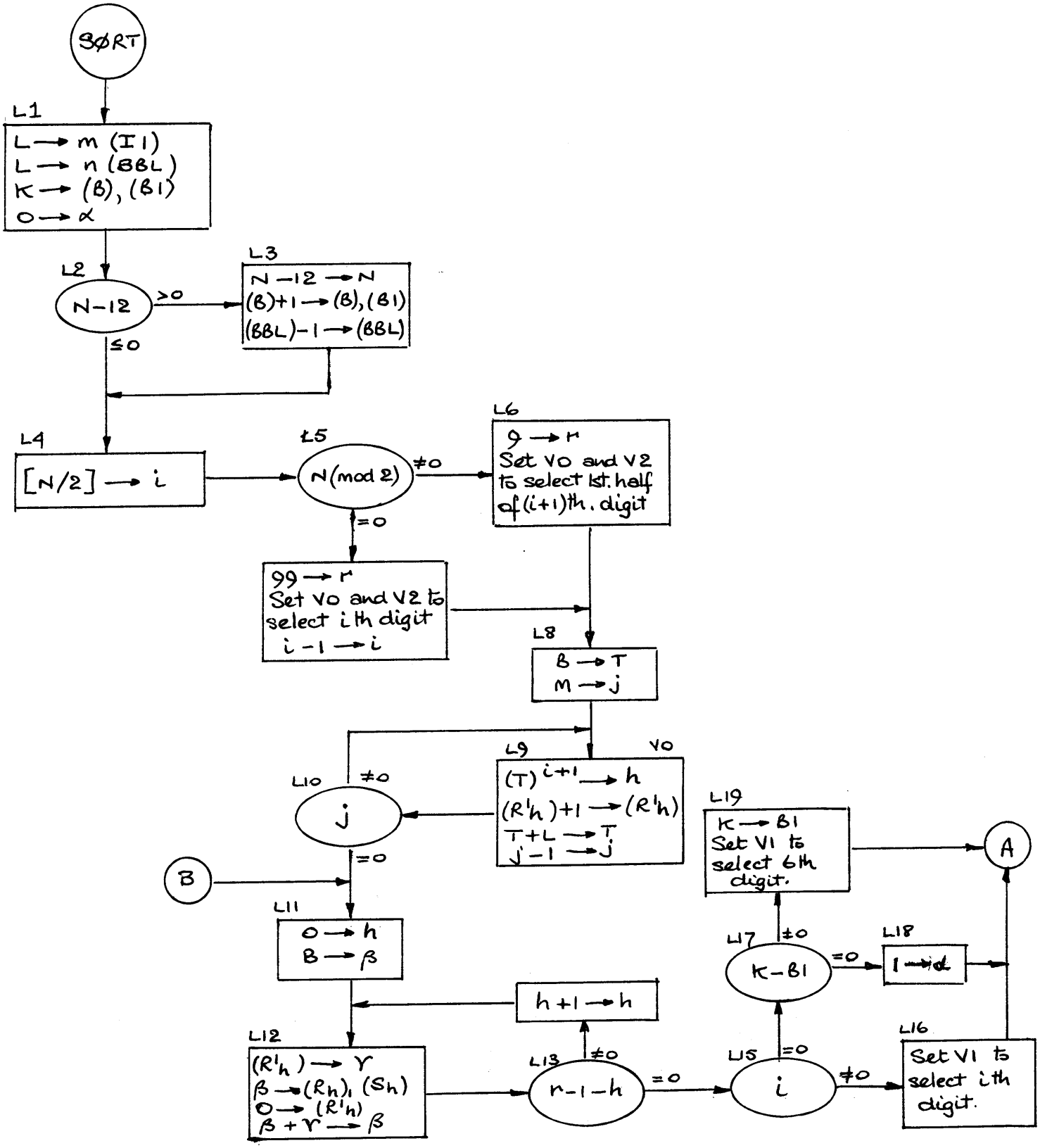
## SORT AND PRINT FILE.

PRESRT 2.



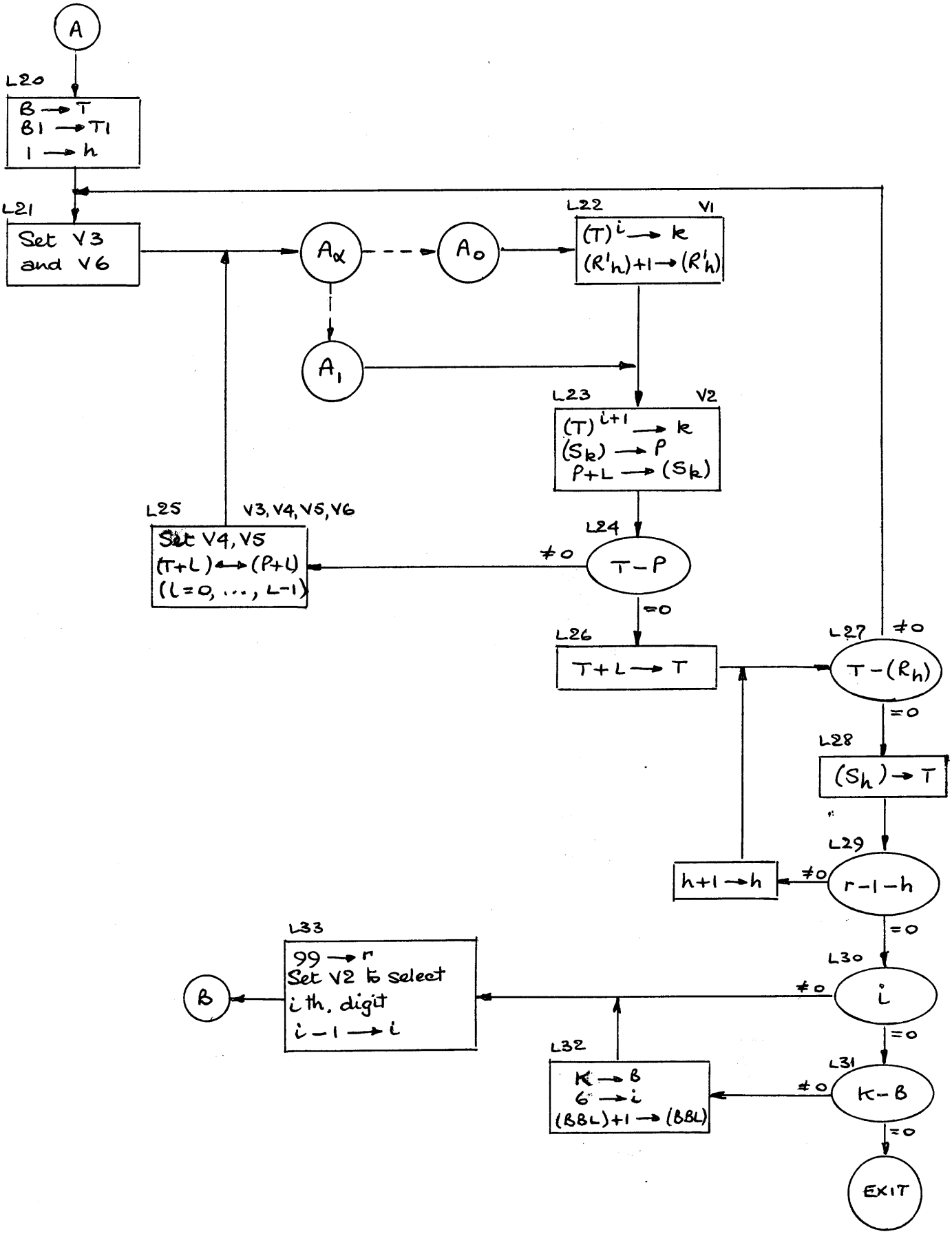
## INTERNAL SORT

Sort M items of length L on N digits starting at K.



## INTERNAL SORT (cont)

Sort M items of length L on N digits starting at K.



## FLOWCHART DEFINITIONS FOR PRESORT.

$L_o$ = Location of 1st. buffer.

$M_o$ = Location of 5th buffer = limit of string buffer.

k = Buffer length (in words).

f = Remaining length of file (initially length of file in words).

t = Item length (in words).

L = Next location in 1st 4 buffers (for constructing string).

M = Next location in 5th. buffer (for overflow items).

i = Output drum (3 or 4 in Presort).

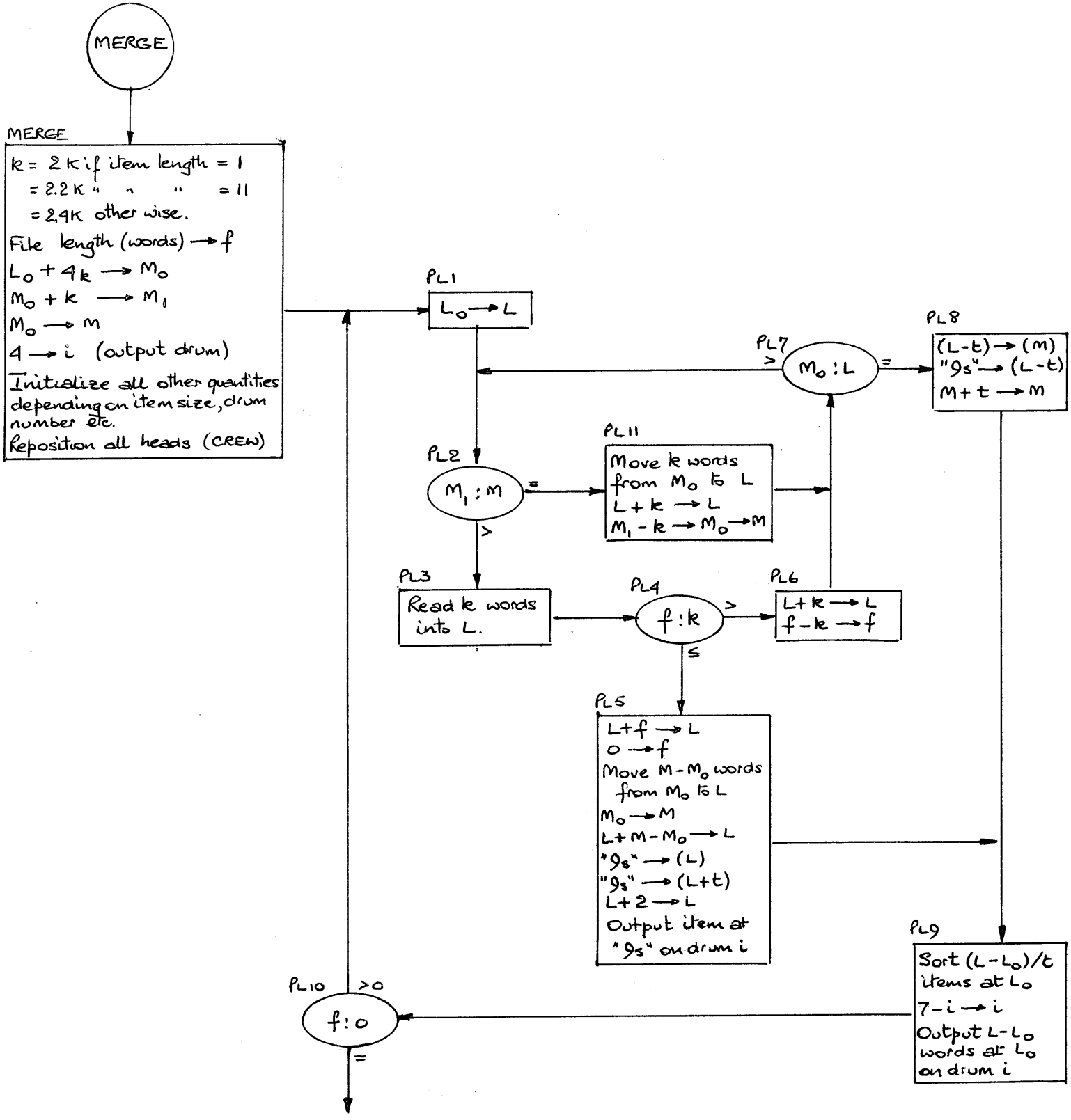
$M_1$ = Limit of overflow buffer.

## A REGISTER ASSIGNMENT.

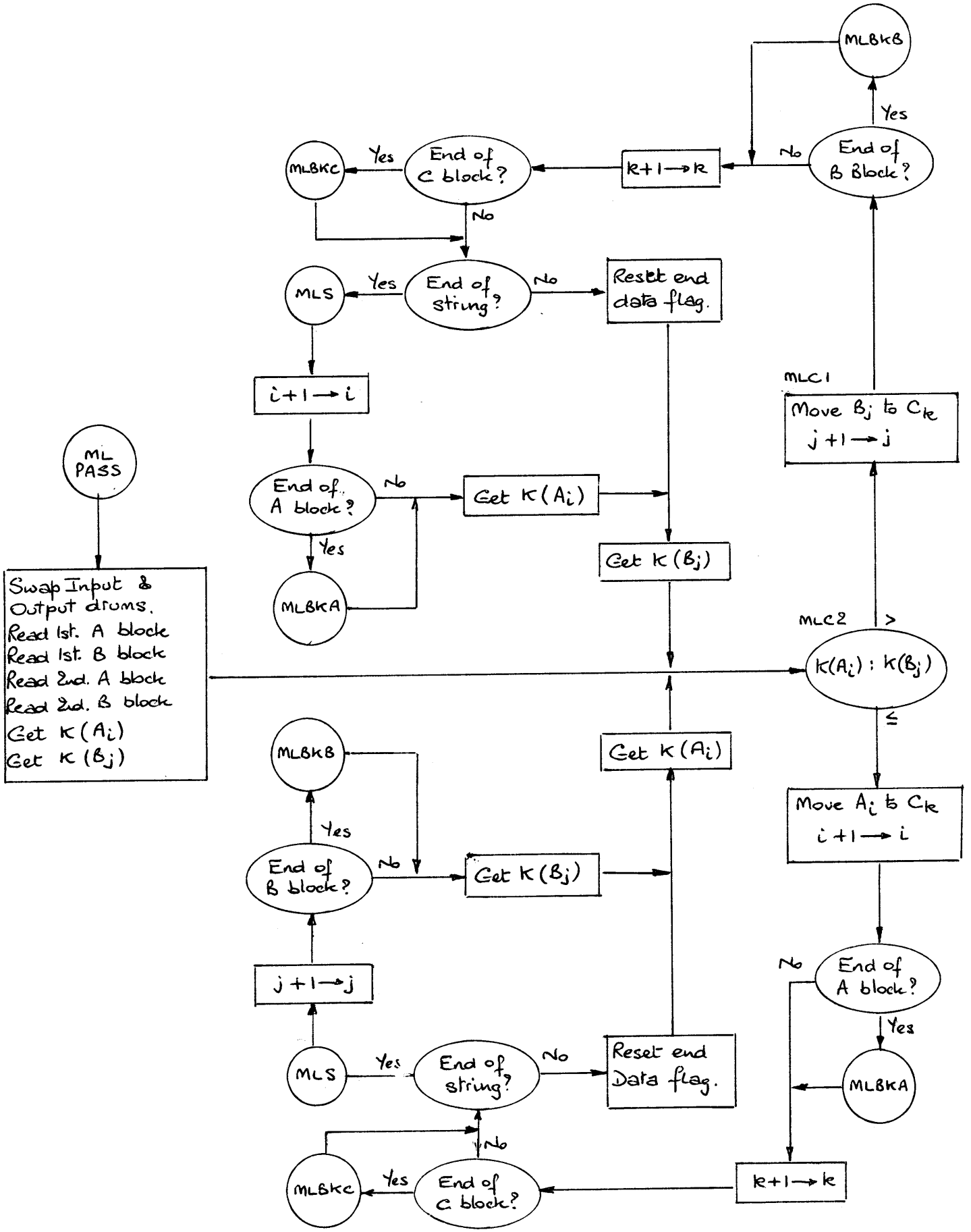
| | Presort | Merge | I/Ø |
|---|---|---|---|
| 16 | f | "9...9" | |
| 17 | k | $K(A_i)$ | |
| 18 | $M_1$ | $K^{-}(A_i)$ | |
| 19 | M | $K(B_j)$ | |
| 20 | $M_o$ | $K^{-}(B_j)$ | |
| 21 | L | i | |
| 22 | | j | *symbolic drum |
| 23 | | k | *number sectors to transfer |
| 24 | | | remaining sectors this band |
| 25 | | | used to construct IØ SUM ØRDS |
| 26 | | | memory LØC to read/write |

*note, 22 and 23 are saved by the I/Ø program.

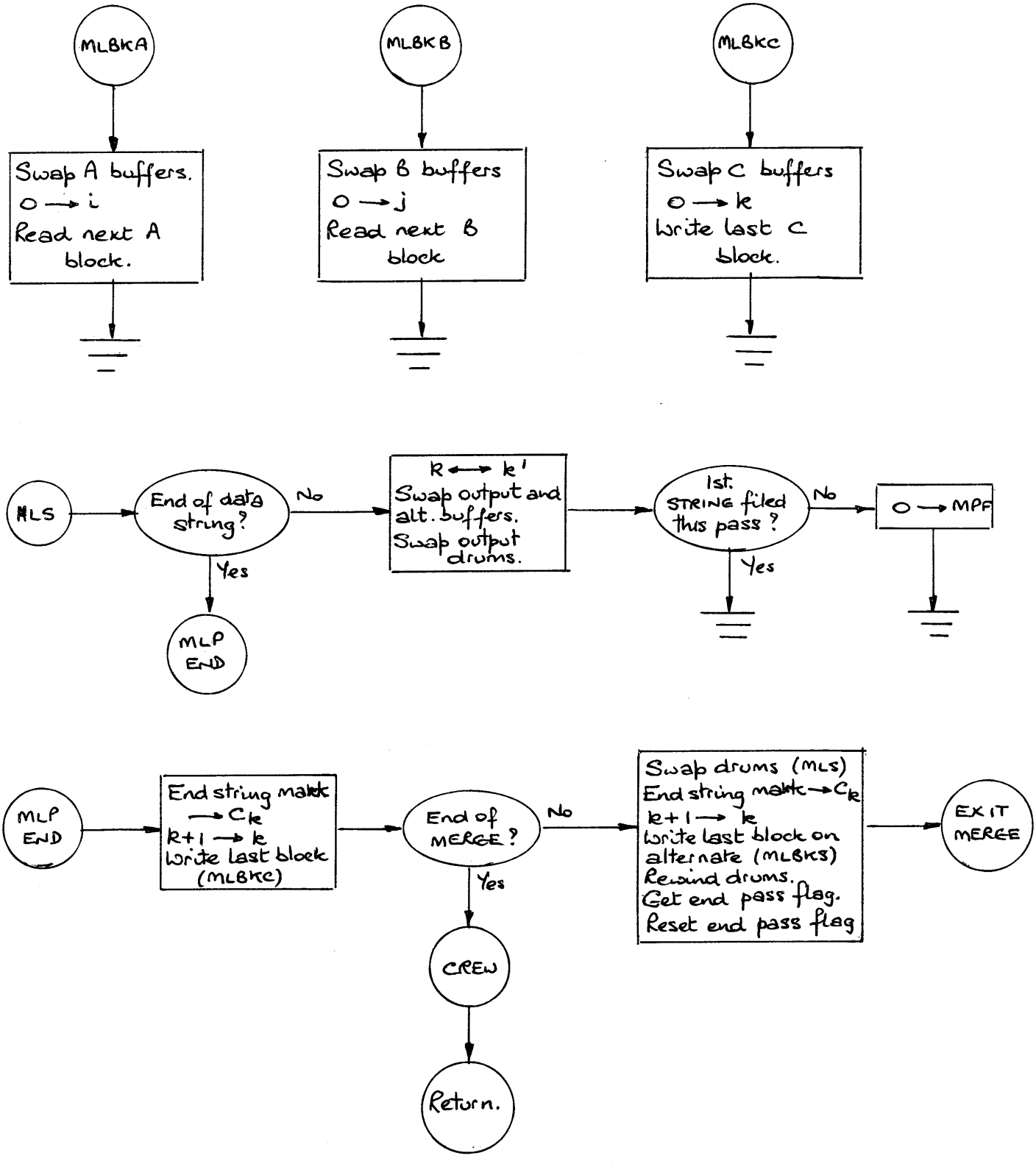
# PRESORT.


MERGE
MERGE
k = 2K if item length = 1
= 2.2K " " " = 11
= 2.4K other wise.
File length (words) → f
L0 + 4k → M0
M0 + k → M1
M0 → M
4 → i (output drum)
Initialize all other quantities depending on item size, drum number etc.
Reposition all heads (CREW)
PL1
L0 → L
PL2
M1 : M
=
>
PL11
Move k words from M0 to L
L + k → L
M1 - k → M0 → M
PL3
Read k words into L.
PL4
f : k
>
≤
PL6
L + k → L
f - k → f
PL5
L + f → L
0 → f
Move M - M0 words from M0 to L
M0 → M
L + M - M0 → L
"9s" → (L)
"9s" → (L+t)
L + 2 → L
Output item at "9s" on drum i
PL7
>
M0 : L
=
PL8
(L-t) → (M)
"9s" → (L-t)
M + t → M
PL9
Sort (L - L0)/t items at L0
7 - i → i
Output L - L0 words at L0 on drum i
PL10
>0
f : 0
=

# MERGE


ML PASS
Swap Input & Output drums.
Read 1st. A block
Read 1st. B block
Read 2nd. A block
Read 2nd. B block
Get K(Ai)
Get K(Bj)
MLBKB
Yes
End of B block?
No
Get K(Bj)
j+1→j
MLS
Yes
End of string?
No
Reset end Data flag.
MLBKC
Yes
End of C block?
No
k+1→k
MLBKA
Yes
End of A block?
No
Get K(Ai)
i+1→i
MLS
Yes
End of string?
No
Reset end data flag.
MLBKC
Yes
End of C block?
No
k+1→k
MLBKB
Yes
End of B Block?
No
MLC1
Move Bj to Ck
j+1→j
MLBKA
Yes
End of A block?
No
MLC2
K(Ai) : K(Bj)
>
≤
Move Ai to Ck
i+1→i

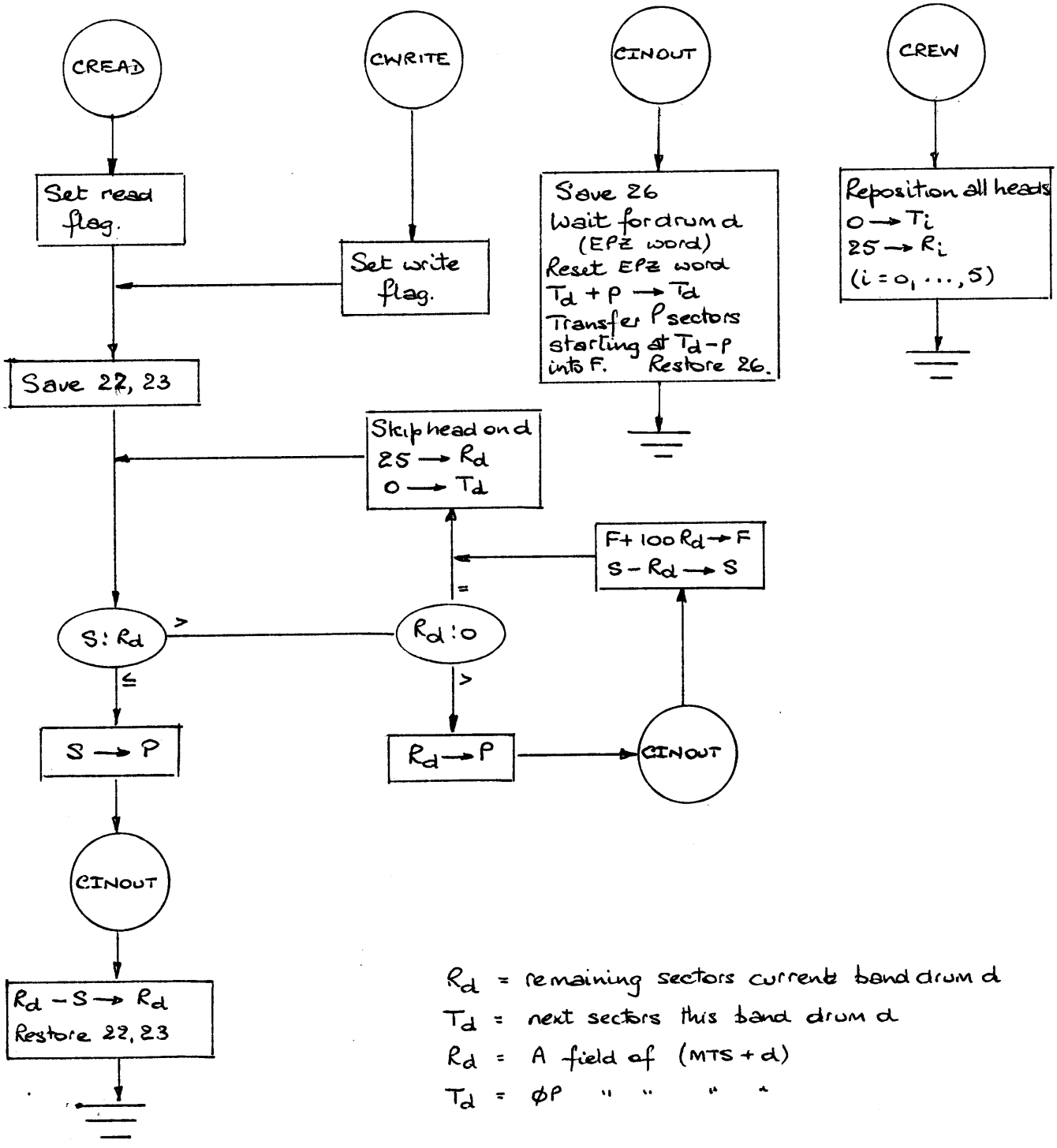
## MERGE (cont.)


MLBKA
Swap A buffers.
0 → i
Read next A block.
MLBKB
Swap B buffers
0 → j
Read next B block
MLBKC
Swap C buffers
0 → k
Write last C block.
MLS
End of data string?
No
Yes
MLP END
k ⟷ k'
Swap output and alt. buffers.
Swap output drums.
1st. STRING filed this pass?
No
Yes
0 → MPF
MLP END
End string mark → Ck
k+1 → k
Write last block (MLBKC)
End of MERGE?
No
Yes
Swap drums (MLS)
End string mark → Ck
k+1 → k
Write last block on alternate (MLBKS)
Rewind drums.
Get end pass flag.
Reset end pass flag
EXIT MERGE
CREW
Return.

# LARC SCIENTIFIC COMPILER

## I/φ ROUTINES for SORT-MERGE

Read into (or Write from) loc. F, S sectors.




$R_d$ = remaining sectors current band drum d

$T_d$ = next sectors this band drum d

$R_d$ = A field of (MTS + d)

$T_d$ = φP " " " "

# FILE

FILE is an LSC subroutine which stores items in files (writes) and retrieves items from files (reads). It is used by the main control program and by each phase of the compiler.

Different types of entrances to FILE specify opening, extending, marking, backing up, closing and releasing files, as well as reading and writing items singly or in groups. Description of these operations will be given later in this section.

For each file, the FILE subroutine keeps track of where in a buffer the next item is to be stored (if writing) or fetched from (if reading). In general, the buffers are intermediate storage areas which hold a block of items prior to and during writing on a drum or during and following reading from a drum. If short enough, however, files may be allowed to remain in their buffers without ever being written on a drum.

Each buffer is actually a double buffer. During writing, for example, items are collected in buffer 1, while the contents of buffer 2 are being written on a drum. When buffer 1 becomes full, the roles of buffers 1 and 2 are interchanged.

Information about each of the files is stored in a set of tables. A description of some of these tables follows.

Table 1 (origin, FLTB1) contains three consecutive words for each file:

OTTIIIDDBBPP
OOOOOOWWWWWW
OOOOOOZZZZZZ or OMMMMMZZZZZZ

T = relative table 2 origin for this file
I = number of words per item
D = logical drum number
B = band number of file origin

P = last permissible band number

W = item counter

Z = number of items in file

M = location of file in memory (zero if file is on drum)

The actual drum number is obtained by adding D to the contents of CRDRF (a cell in the communication region). Exceptions are files 1 and 10, whose actual drum numbers are found in CRDRI and CRDRO, respectively. The starting band and sector numbers of files 1 and 10 are likewise found in CRDRI and CRDRO. The format of each of these cells is:

BB Sector Band Drum

Word 3 of table 1 is used as an item counter during the writing of the file, and has the form indicated above only after the file has been closed or released.

Table 2 (origin of file F information: FLTB2+F) has the format

FORM 2235 PH R $V_1V_1V_2$ M1

PH = phase number

R = relative locator for tables in phase PH

$V_1V_100$ = L1 = capacity of buffer 1 in phase PH

$V_1V_100 - V_200$ = L2 = capacity of buffer 2 in phase PH

M1 = origin of buffer 1 in phase PH

M1 + $V_1V_100$ = M2 = origin of buffer 2 in phase PH

The tables with which R is concerned contain origins of special sequences for handling writing or reading of items or groups of items, and locations of instructions in these sequences which must have their addresses preset. These tables are numbers 5,6,7,9, 10,12, and 14.

If I, the item length, is not a factor of 100, space for I-1 additional words must be provided in front of buffer 1 when reading, and space for I-1 additional words must be provided following buffer 2 when writing.

Table 4 is a storage area reserved for index words, one for each file. The index word format is given here to indicate the notation used:

BB NNN I Delta

Table 8 consists of three words per file. These are initially zeros, and are replaced by drum summary orders when required. The location of the second of these three words is contained in the third word, which is an EPZ. Because of this, an absolute zero replaces the second word when the drum operation is completed.

Table 15 is a storage area reserved for the storage of buffer counters, one for each file. These have the format BB 0 C 0, where C = 0,2,4,etc. C is increased by

2 each time a drum read or write operation for the associated file is initiated. The buffer counters are used only by the marking and backing up operations of phase 1. To form the mark word, the buffer counter is combined with the variable portions of the index word. The format of the mark word is:

BB NNN C Delta

Another storage area, FLZ, contains transient information about a file. The contents of this area are as follows (cells not shown have general or temporary uses):

| Cell | Contents |
|---|---|
| FLZ + O | file number |
| + 1 | 3 X file number |
| + 2 | relative location of table 2 information |
| + 3 | I |
| + 4 | physical drum number |
| + 5 | starting band number |
| + 6 | P |
| + 7 | R |
| + 8 | M1 |
| + 9 | M2 |
| + 10 | L1 |
| + 11 | L2 |
| + 12 | L(from calling sequence) |
| + 13 | starting sector number |
| + 17 | current band number |
| + 24 | L2 |
| + 25 | L1 |
| + 27 | L1 + L2 |
| + 30 | I X 100000 |

FILE has four calling sequences. Two of these (numbers 2 and 4) are high-speed entries to special instruction sequences within FILE which handle the two basic operations - reading or writing a single item.

Calling sequence number 1:

```
TB  1  FILE
XX  F  L
```

XX = SK: open file F for writing

XX = AX: mark the current positions in phase 1 output files 2,3,4,91,93, store mark words in L on

XX = A: store in file F the n items starting in L (n must be given in register 2)

XX = AM: back up to positions marked in phase 1 output files 2,3,4,91,93 (the mark words must be supplied in L on)

XX = AU: close output file F

XX = AAX: extend output file F

XX = AA: release output file F

XX = NX: open file F for reading forward

XX = N: open file F for reading backward, starting at the end of the file (items not reversed)

XX = NNX: mark current position input file 1, store mark word in L

XX = NN: back up to the item preceding the position marked in input file 1 (the mark word must be supplied in L)

XX = MXR: read the next line (10 words) from input file 1, store in L on.

If part of a file is on a drum and part in a buffer, closing or releasing the file causes the contents of the buffer to be written onto the drum. If the file is entirely in memory, closing the file leaves the file in memory, and releasing the file causes it to be written onto a drum. Extending a file allows more items to be added to a file after it has been closed or released.

Calling sequence number 2:

```
TB  1  FILE + F
H   O  L
```

This causes one item, whose first word is in L, to be stored in file F.

Calling sequence number 3:

```
TB  1  RFILE
XX  F  L
mask (or 0)
```

XX = MXE: read the next n items from file F, store in L on (n must be given in register 2)

XX = MR: read the group of items (defined by the mask) from the file F, store in L on, supply index word in register 2

XX = M: read the next item from file F without counting it, store in L on,

The mask is a word of 1's and 0's (e.g., 111100001111) and is applied in MR entries

to the first word of each item to determine a group. All consecutive items in a file which have their 1-corresponding first word digits respectively equal constitute a group (with respect to the mask).

Upon return from an MR entry, register 2 contains an index word:

BB NNN I L

where NNN is the number of items in the group.

After an M entry has been used to read the nth item, the next reading of the same file will then again bring in the nth item, rather than the (n + 1) st item as in the normal case.

The normal return from the FILE subroutine following an entry via calling sequence number 3 or 4 is to the second instruction beyond the mask or zero word. The end-of-file return is to the first instruction following the mask or zero word. The end-of-file return is made following every entry after the end of file has been reached, unless an intervening NX or N entry has been made.

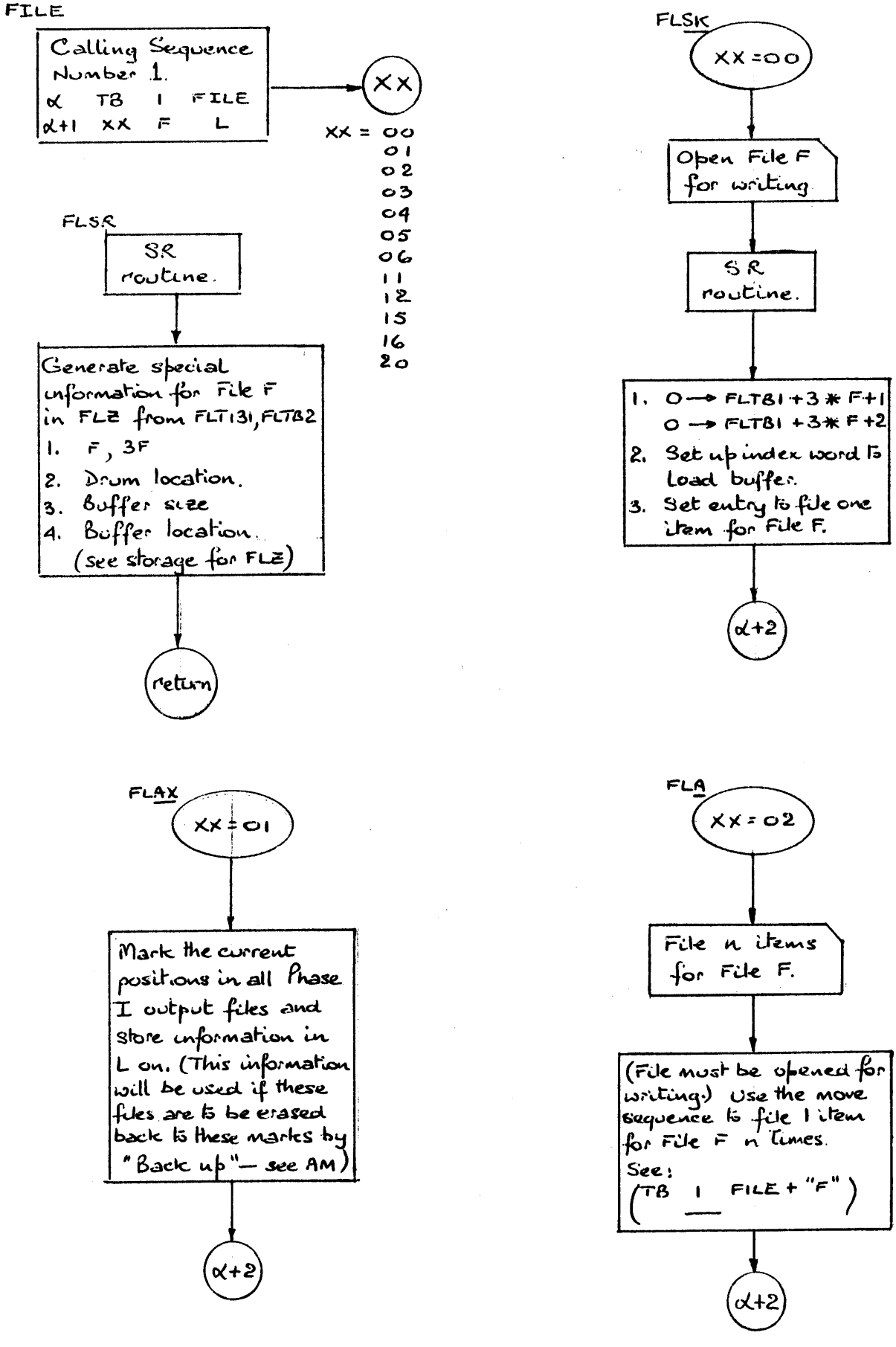
Upon a normal or end-of-file return following an MXE entry, the actual number of items read appears in register 2. An end-of-file return following an MR entry indicates that all the items in the file have been read previous to this entry.

All references to files are made symbolically. For example, F3 denotes file 3. The correspondences between symbol and absolute designations are given in the following table:

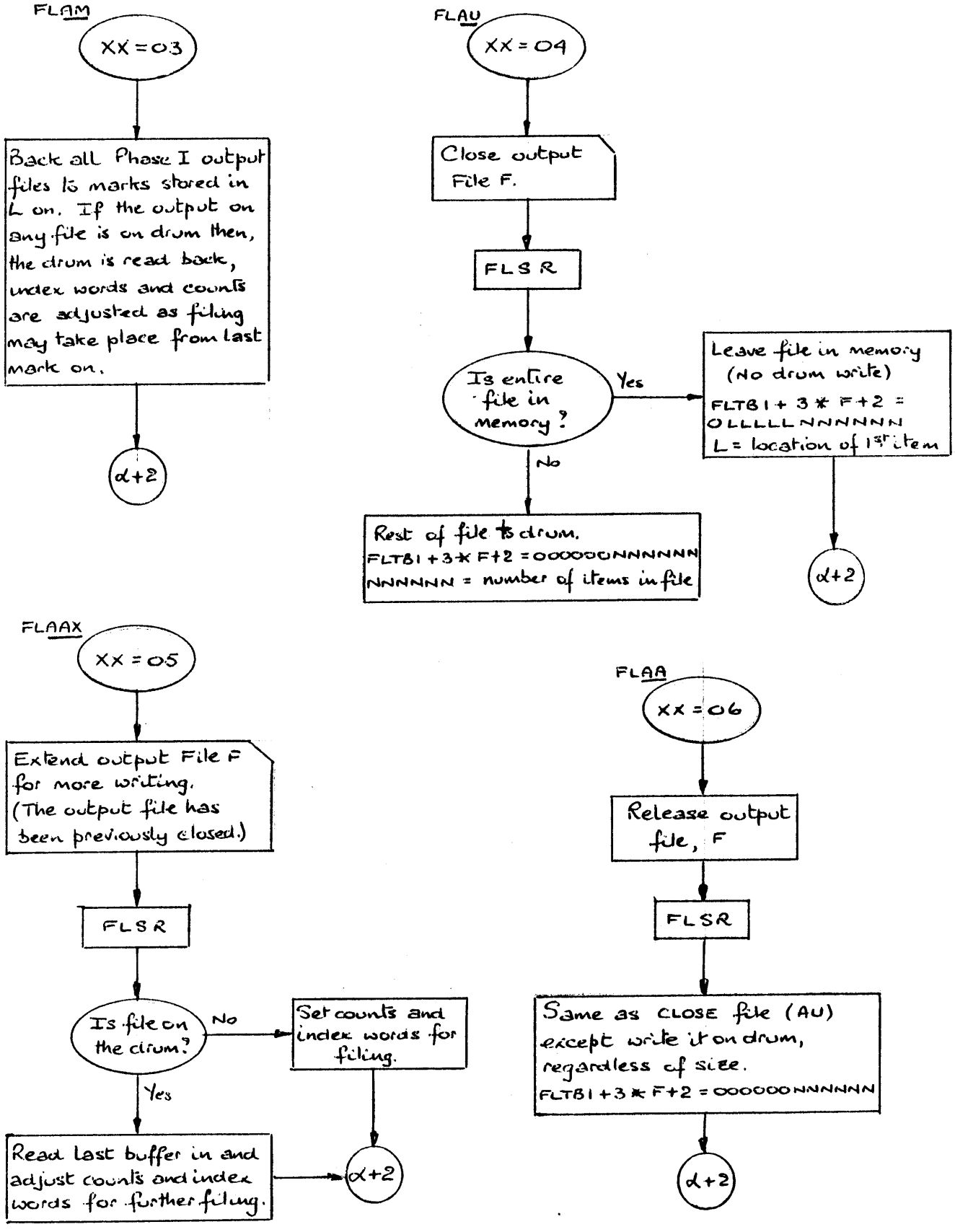
| Symbolic | Absolute |
|---|---|
| F1 | 3 |
| F2 | 4 |
| F3 | 5 |

| | |
|---|---|
| F4 | 6 |
| F5 (or F50) | 7 |
| F51 | 8 |
| F52 | 9 |
| F53 | 10 |
| F54 | 11 |
| F55 | 25 |
| F56 | 26 |
| F6 (or F60) | 12 |
| F61 | 13 |
| F7 (or F70) | 14 |
| F8 (or F80) | 15 |
| F81 | 16 |
| F82 | 17 |
| F83 | 18 |
| F84 | 19 |
| F85 | 28 |
| F9 (or F90) | 20 |
| F91 | 21 |
| F92 | 22 |
| F93 | 23 |
| F94 | 27 |
| F10 | 24 |

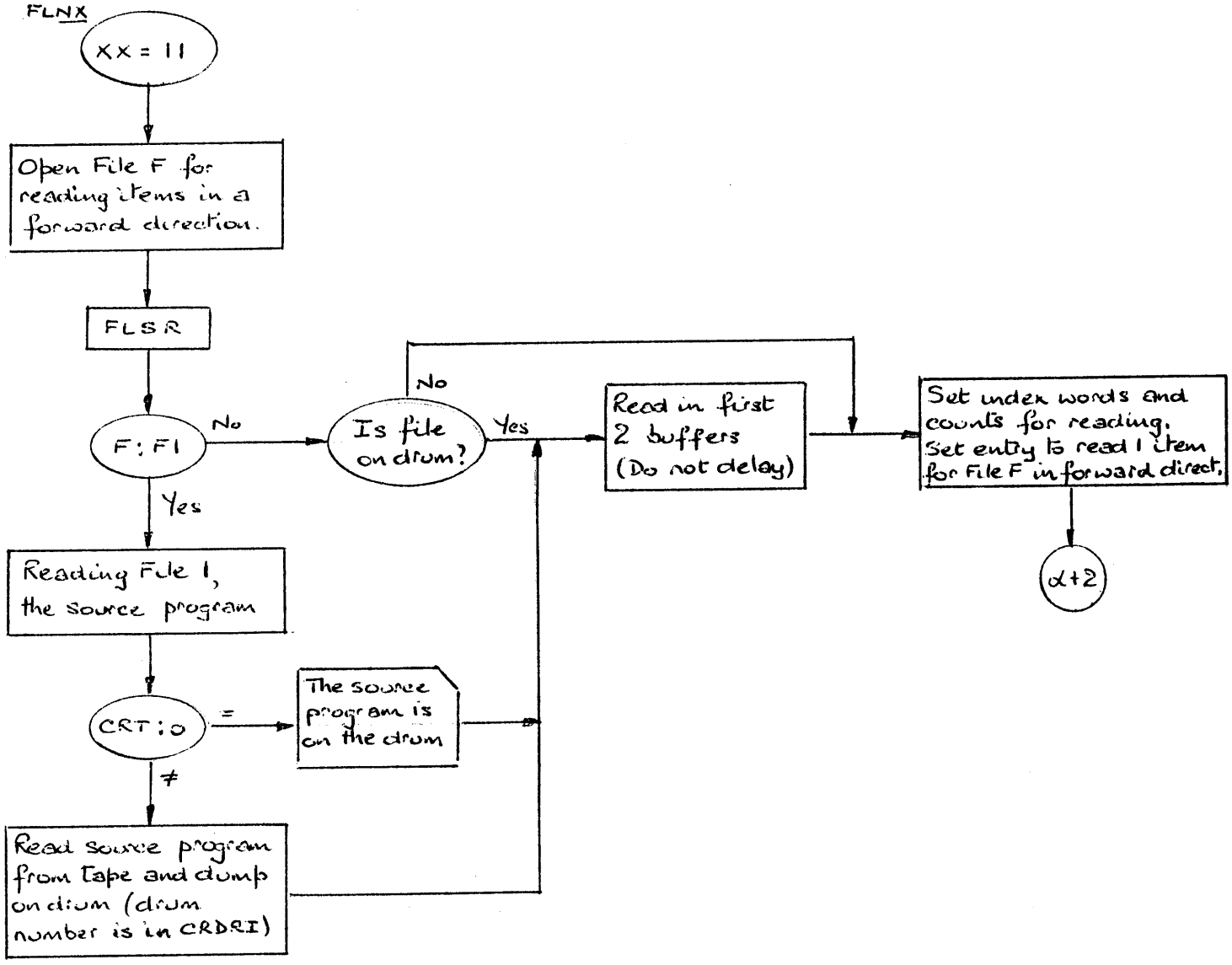
FILE ROUTINE (details in Parallel Code Edit)

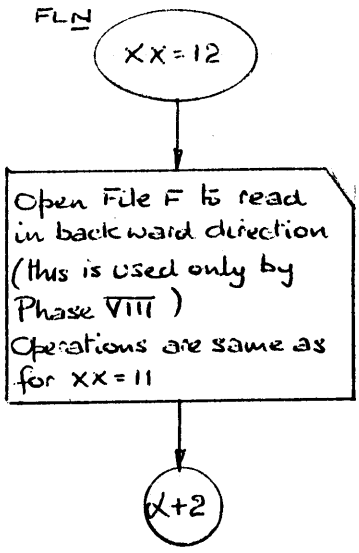


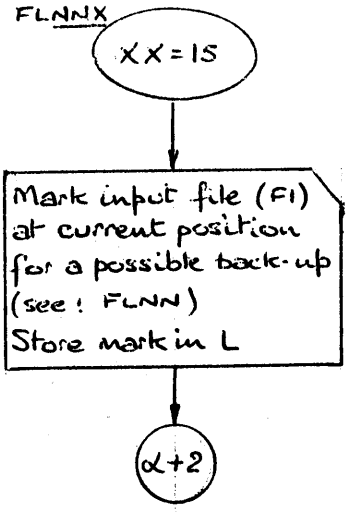
FILE ROUTINE (continued)

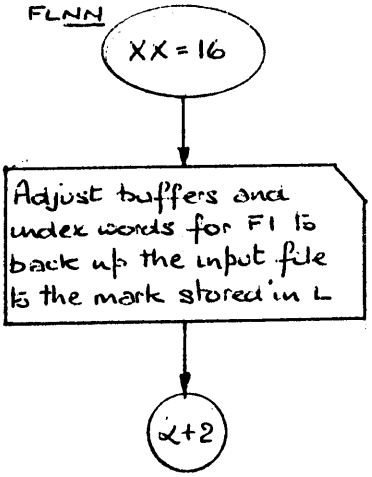


FILE ROUTINE (continued)

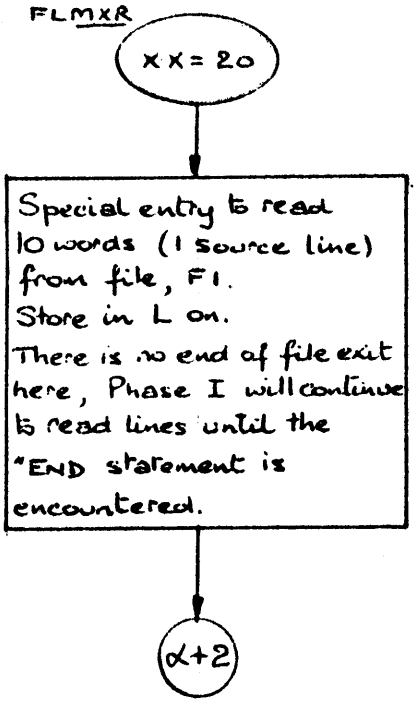


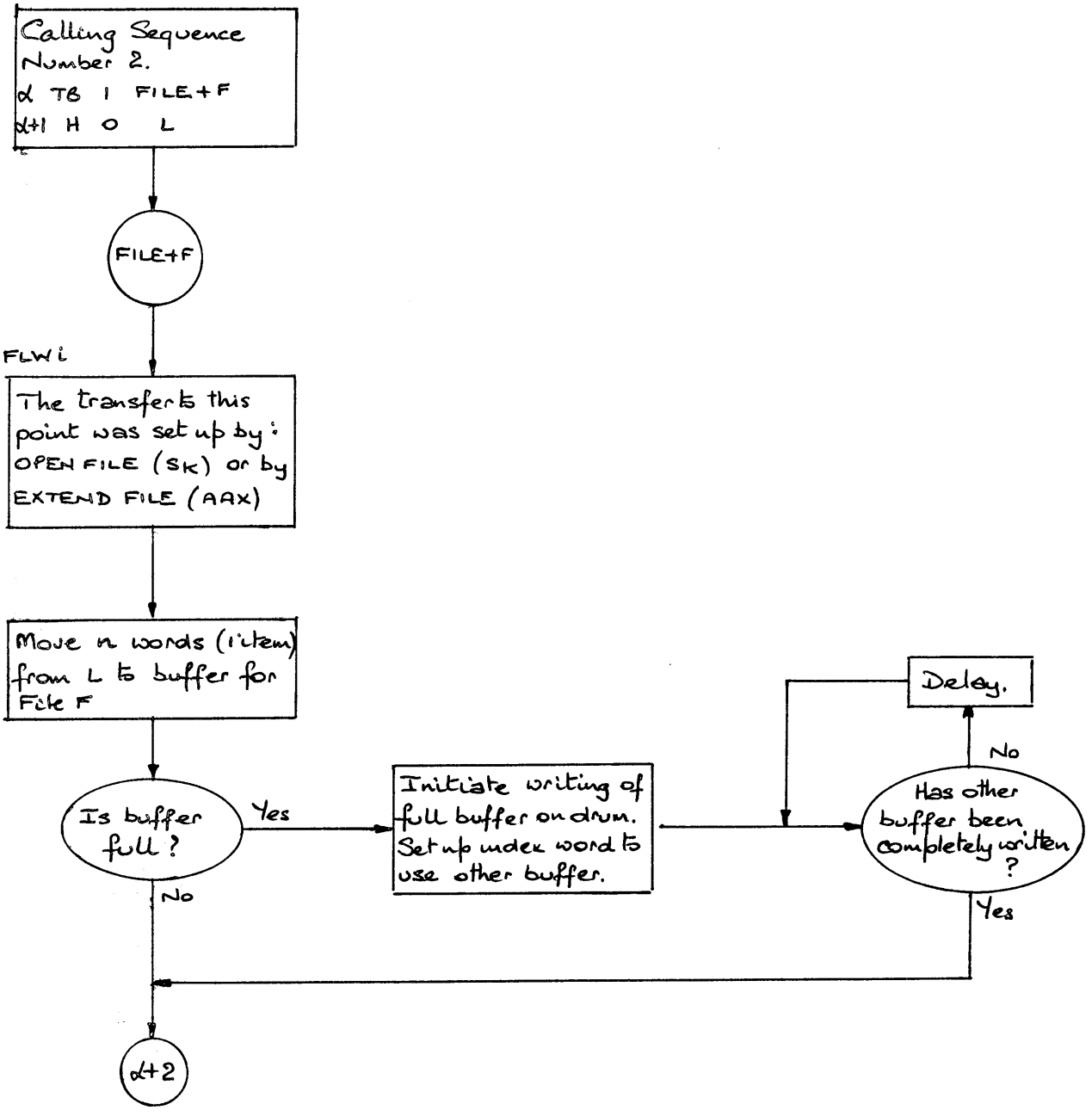



FILE ROUTINE (continued)



FILE ROUTINE (continued)



FILE ROUTINE (continued)

RFILE

Calling Sequence
Number 3.
α TB 1 RFILE
α+1 YY F L
α+2 mask (or zero)
α+3 possible End of File.
α+4 return

YY

YY = 21
22
23

FLMXE

YY = 21

Read the next n items from File F, using the read 1 item sequence for File F (see : RFILE+F)

Exit to α+3 if End of File was encountered during reading.

α+4

FLMR

YY = 22

Read the next "group" of items from File F.
A group of items are those consecutive items that have the same digits in the positions defined by the mask in α+2
(see: RFILE+F for buffering)

Exit to α+3 if End of File was encountered during reading.

α+4

FLM

YY = 23

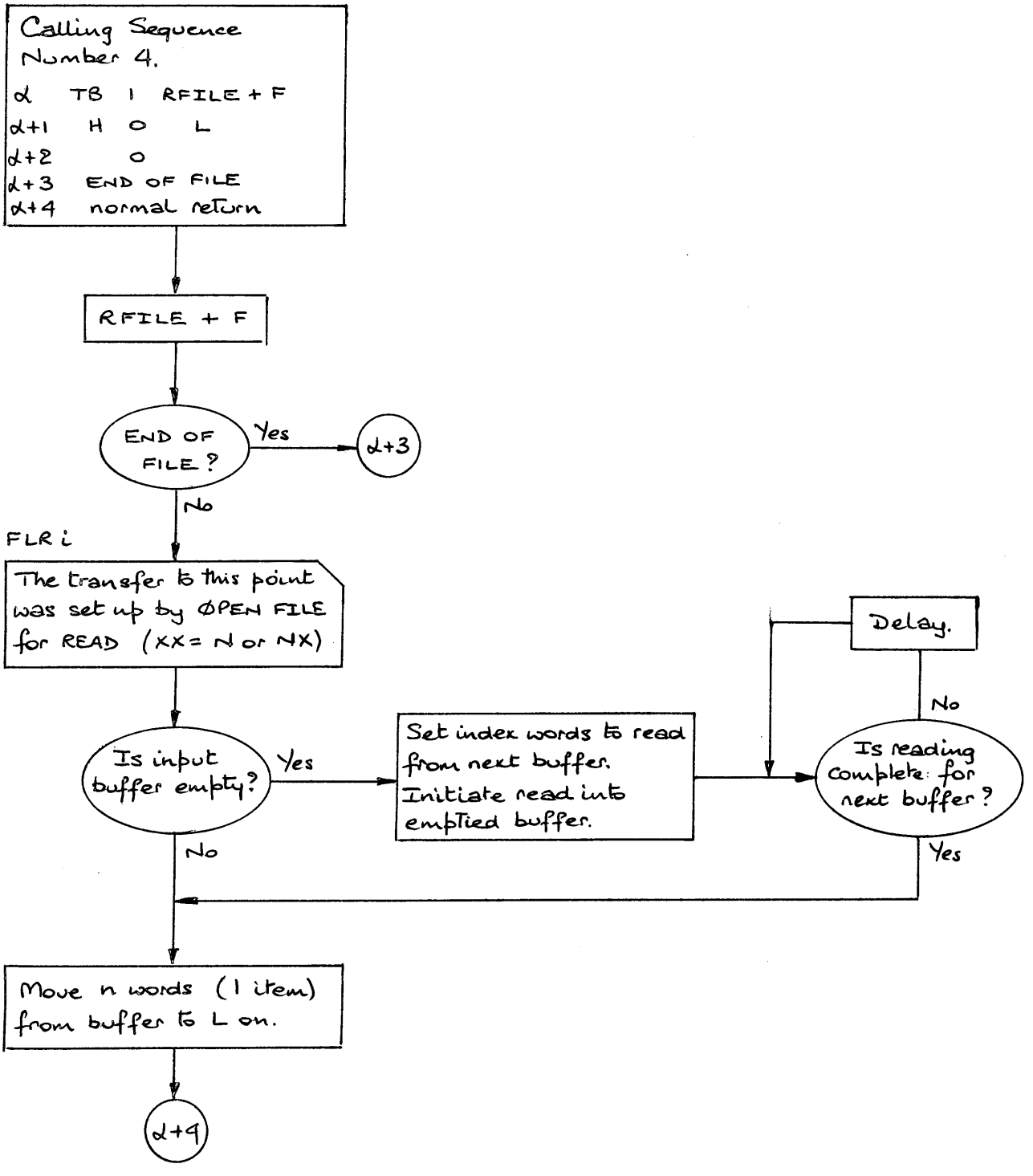
End of File ? Yes → α+3

No

Read the next item from File F into L on without advancing buffer locations or counts. (Allows peeking at next item and reading it later.

α+4

FILE ROUTINE (continued)



# INPUT-OUTPUT ROUTINES

A typical calling sequence to an input-output routine is:

```
TB     # 0      9RIPT
SK       0      (0 . 2 (2) )
SK       0      10L   0   9
```

The second word contains the location of the tape number (the tape number is in floating point), and the third word contains the format location. If the format is assigned, the third word becomes

```
SK       0      (10L)  0   6
```

Immediately following the calling sequence are the list instructions. A typical list for an input routine is:

```
A , B , (C(I), I = 1,10)
```

which might be represented by the following instructions:

```
S      1     A       0      5
SS     2     B       0      5
F      #13   (BB     10     1     1)
S      1     C       #13    5
BIT    #13   HERE-1
T            9FIN
```

Here B is double precision, the other numbers single precision.

The same list for an output routine would have the S and SS instructions replaced by F and FF instructions.

Before referring to the list, the input-output output routine causes the tracing mode to be entered. This is done in the subroutine 9LEV.

Reference is made to the list simply by transferring to its first instruction (by way of the subroutine 9NTR, which transfers to 2600, in which 9LEV has stored a transfer to the list origin). When an instruction with a tracing mode selector of 5 is reached, control returns to the input-output routine by way of location 2601 (in which 9LEV has stored a transfer to the subroutine 9XCH). A transfer to the next list instruction is also automatically stored in 2600.

9XCH, after ascertaining that the tracing of an instruction did indeed just occur (rather than the occurrence of an error, which would also cause a transfer to 2601), forms an untraced version of the traced instruction, places it in the cell 9XCHX, and executes it.

For example, the instruction:

```
S    1    A    0    5
```

would become

```
S   #1    A
```

Prior to the execution of this instruction (in fact, just before transferring to 9NTR), an input routine would place the current word from the input buffer in #1 (or #1 and #2, if double-precision transmission is a possibility). An output routine would store the contents of #1 (and perhaps #2) in a buffer after the execution of a fetch in 9XCHX.

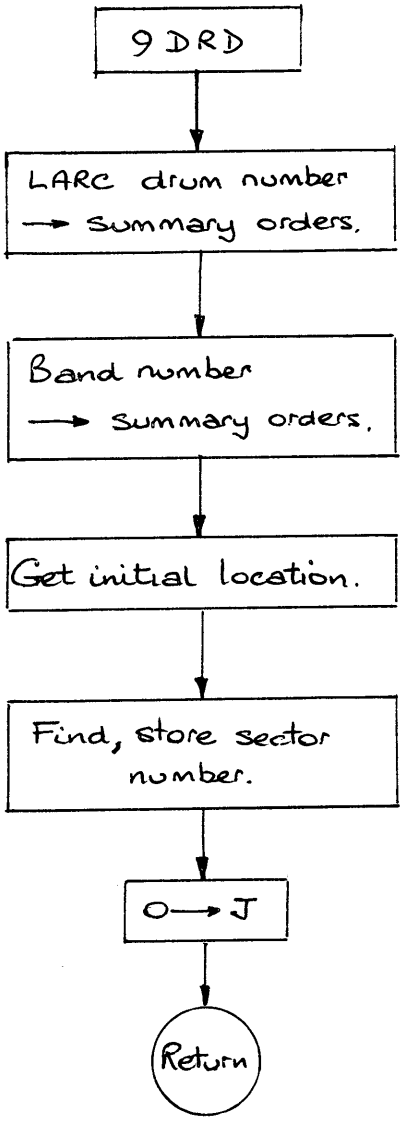
If a format is involved in the input-output operation, the scanning of the alphanumeric format is interpolated in the process described above. The subroutines WTG, GN, and GXB are used for this purpose.

Two 20-word buffers are used for alphanumeric tape input (READ; READ INPUT TAPE). While a tape block is being processed in one buffer, the next block is being read into the other buffer. However, if the program reads more than one input tape, the second one referred to must operate with a single buffer.

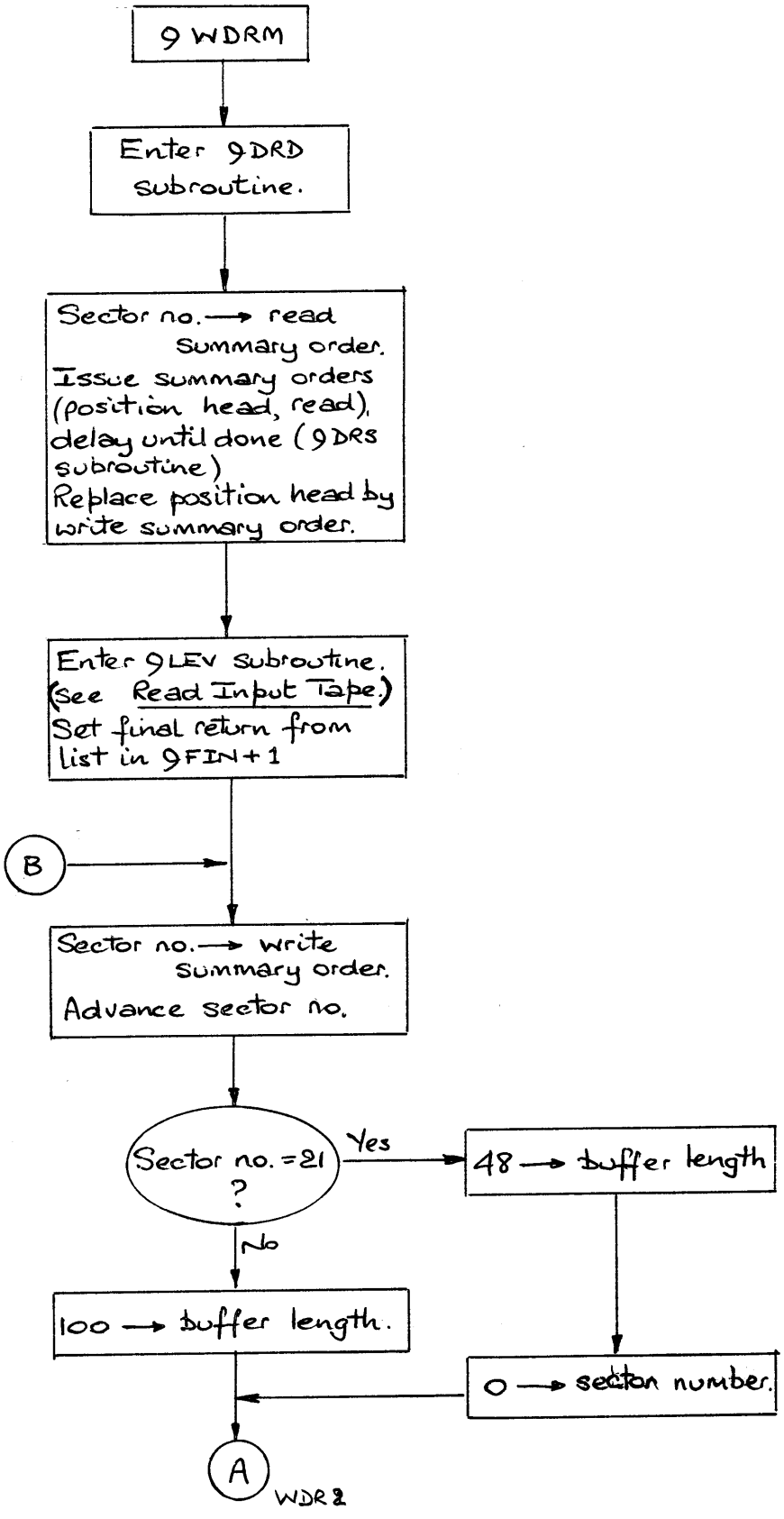
Two 20-word buffers are also used for alphanumeric tape output (PRINT; PUNCH; WRITE OUTPUT TAPE). A block is prepared in the first buffer and then moved to the second buffer for writing on tape, during which the preparation of the next block in the first buffer can proceed.

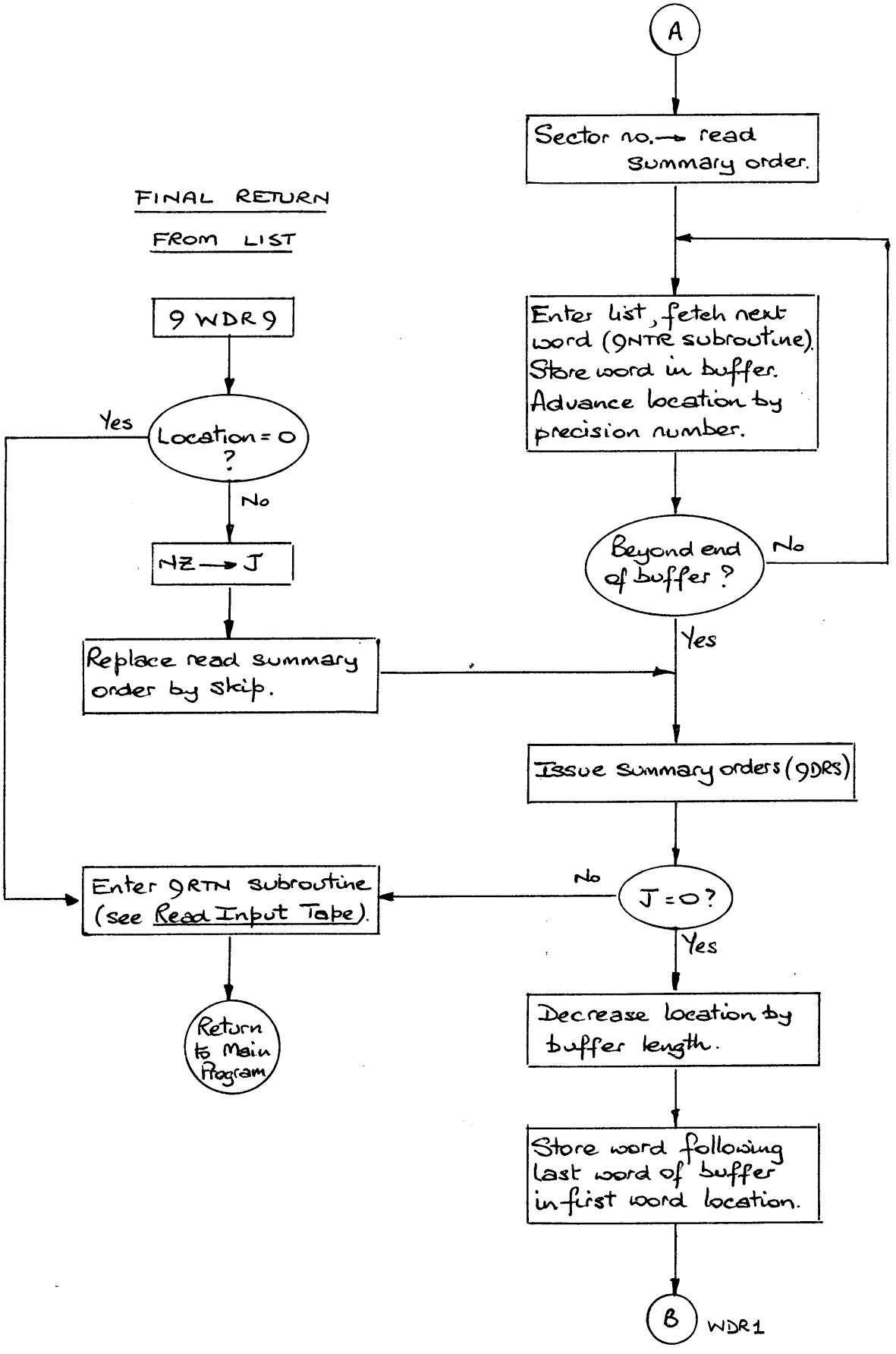
The remaining input-output operations use single buffers (with the exception of REWIND, REWIND INTERLOCK, and alphanumeric END FILE, which do not require buffers). WRITE DRUM and READ DRUM share a single 100-word buffer. WRITE TAPE, END FILE (decimal mode), and READ TAPE also share a single buffer whose origin (9STB) and length (9TBL) are given in the main program. (9TBL is a constant supplied from the communication region of LSC via CRTREC.)

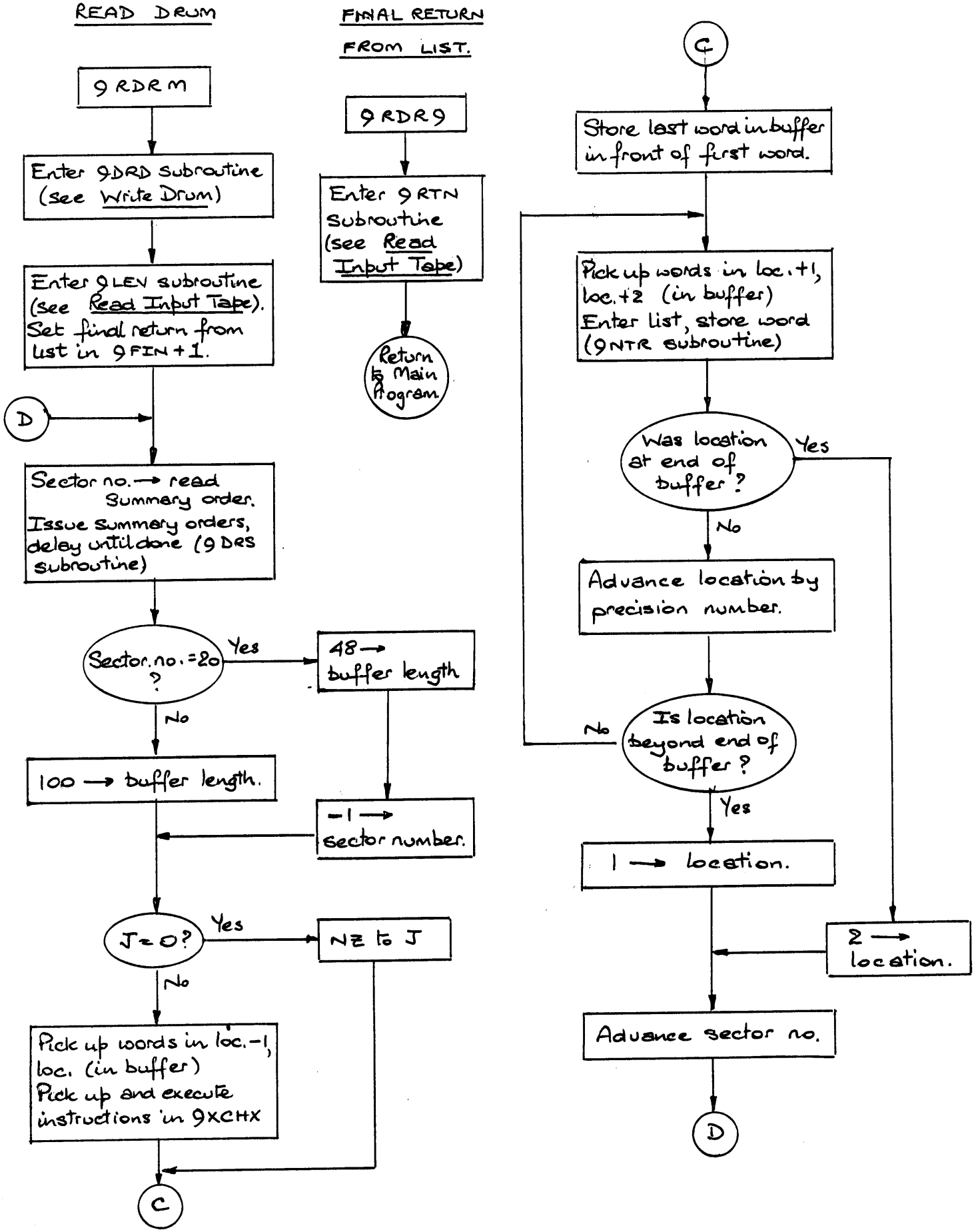
More details on the above program are found in the attached flow charts and the code listings of these programs.


9 DRD
LARC drum number → summary orders.
Band number → summary orders.
Get initial location.
Find, store sector number.
0 → J
Return

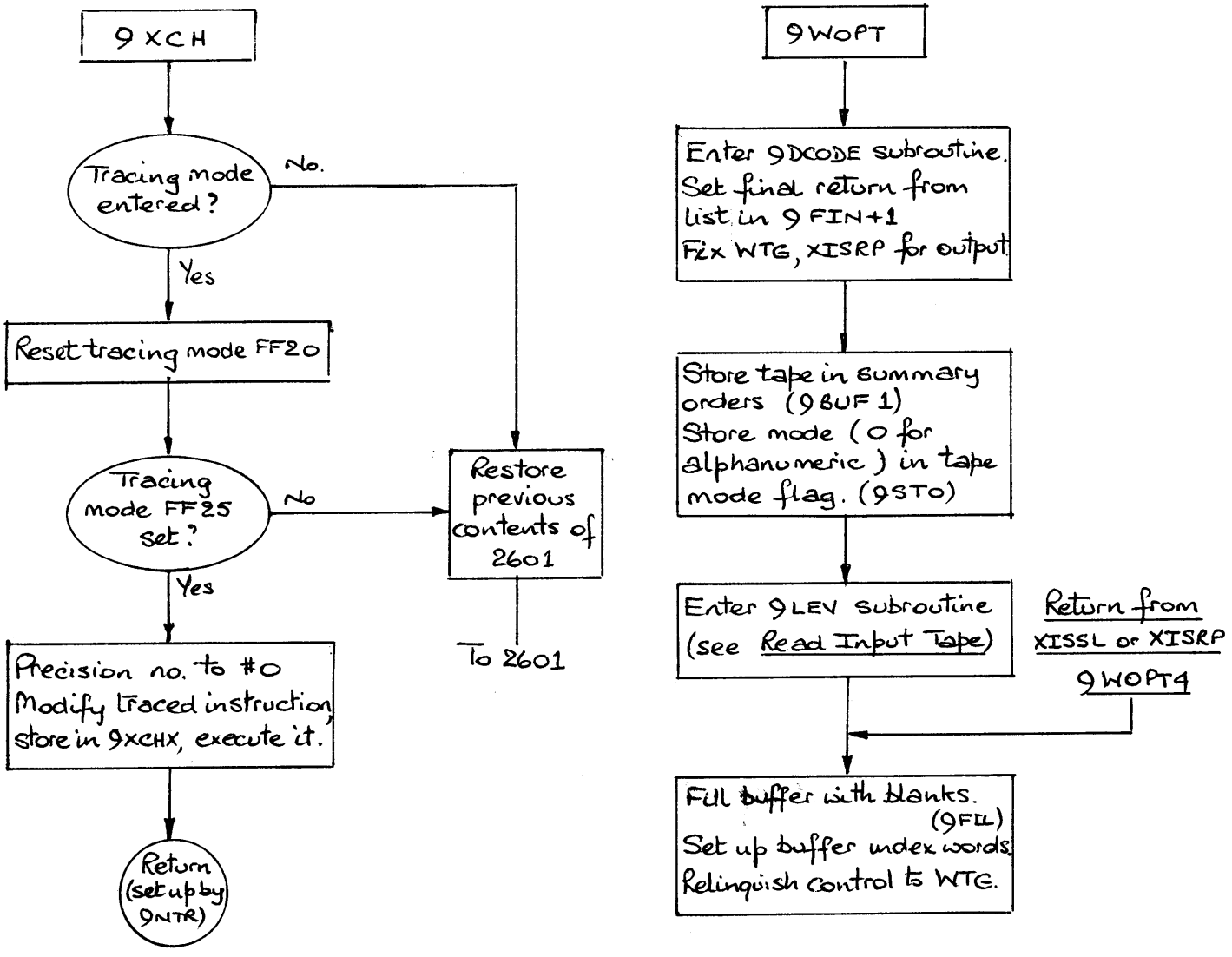
## WRITE DRUM


9 WDRM
Enter 9 DRD subroutine.
Sector no. → read summary order.
Issue summary orders (position head, read), delay until done (9 DRS subroutine)
Replace position head by write summary order.
Enter 9 LEV subroutine. (See Read Input Tape.)
Set final return from list in 9 FIN + 1
B
Sector no. → write summary order.
Advance sector no.
Sector no. = 21 ?
Yes
48 → buffer length
No
100 → buffer length.
0 → sector number.
A
WDR 2

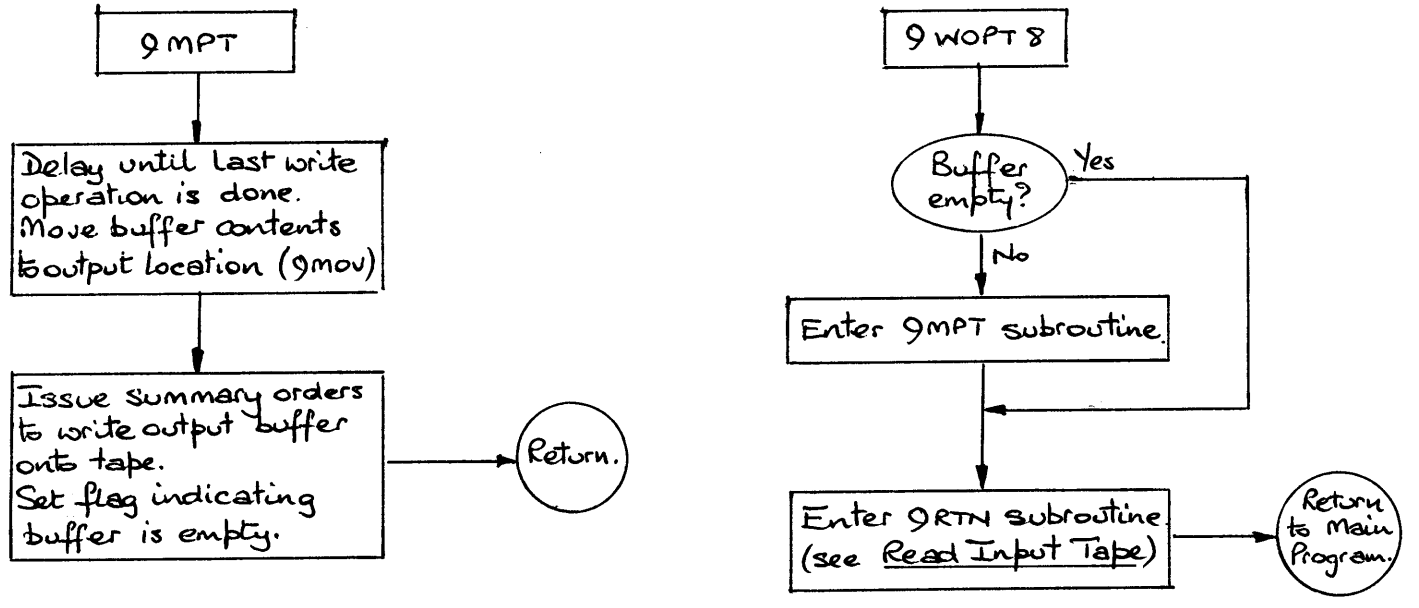

A
Sector no.→ read summary order.
Enter list, fetch next word (9NTR subroutine). Store word in buffer. Advance location by precision number.
Beyond end of buffer ?
No
Yes
Issue summary orders (9DRS)
J = 0 ?
No
Yes
Decrease location by buffer length.
Store word following last word of buffer in first word location.
B
WDR1
FINAL RETURN FROM LIST
9 WDR 9
Location = 0 ?
Yes
No
NZ → J
Replace read summary order by skip.
Enter 9RTN subroutine (see Read Input Tape).
Return to Main Program


READ DRUM
9RDRM
Enter 9DRD subroutine (see Write Drum)
Enter 9LEV subroutine (see Read Input Tape). Set final return from list in 9FIN+1.
D
Sector no. → read Summary order. Issue summary orders, delay until done (9DRS subroutine)
Sector no. = 20 ?
Yes
48 → buffer length
No
100 → buffer length.
-1 → sector number.
J = 0?
Yes
NZ to J
No
Pick up words in loc.-1, loc. (in buffer) Pick up and execute instructions in 9XCHX
C
FINAL RETURN FROM LIST.
9RDR9
Enter 9RTN subroutine (see Read Input Tape)
Return to Main Program
C
Store last word in buffer in front of first word.
Pick up words in loc.+1, loc.+2 (in buffer) Enter list, store word (9NTR subroutine)
Was location at end of buffer?
Yes
No
Advance location by precision number.
Is location beyond end of buffer?
No
Yes
1 → location.
2 → location.
Advance sector no.
D

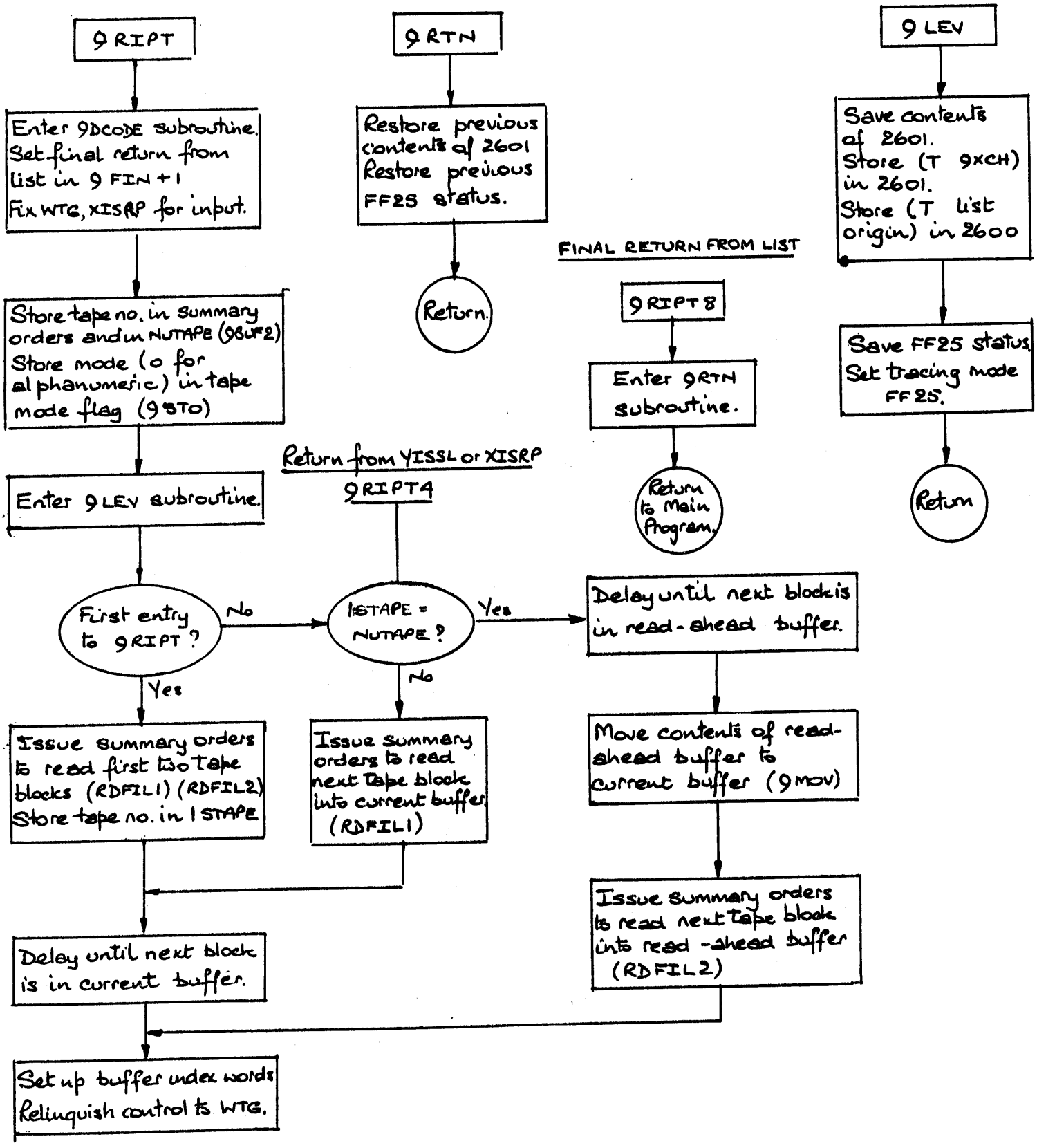
## WRITE OUTPUT TAPE, PRINT, PUNCH




## FINAL RETURN FROM LIST.



# READ INPUT TAPE, READ.



ROUTINES USED by 9RIPT & 9WOPT

9DCODE

Set up format index words.

0 → scale factor, parenthesis level, repeat flag.

Set flag indicating buffer is empty.

Return

XISRP

Subtract 1 from parenthesis level; >0 ?

No → Restore parenthesis level, format index words to values at initial left parenthesis. → TO XISSL 1 (output) or RDTEST (input)

Yes

Subtract 1 from repeat ctr; >0 ?

No → 20 WTG

Yes

Restore parenthesis level, format index words to values at last left parenthesis. → 20 WTG.

Scan Format.

WTG.

Enter GN subroutine.

GN: obtain number (or 0) in NUM and #3, following non-blank character in #4

| Character | Transfer to | |
|---|---|---|
| ( | XISLP | |
| (If not (, store NZ in repeat flag) | | |
| ) | XISRP | |
| P | XISP | |
| X | XISX | |
| E | XISE, YISEI | X for output, Y for input. |
| F | XISF, YISF | |
| I | XISI, YISEI | |
| H | XISH, YISH | |
| A | XISA, YISA | |
| / | XISSL, YISSL | |
| other | WTG. | |

XISSLP

Add 1 to parenthesis level.

Repetition number → repeat ctr.

Save format index words for repeat.

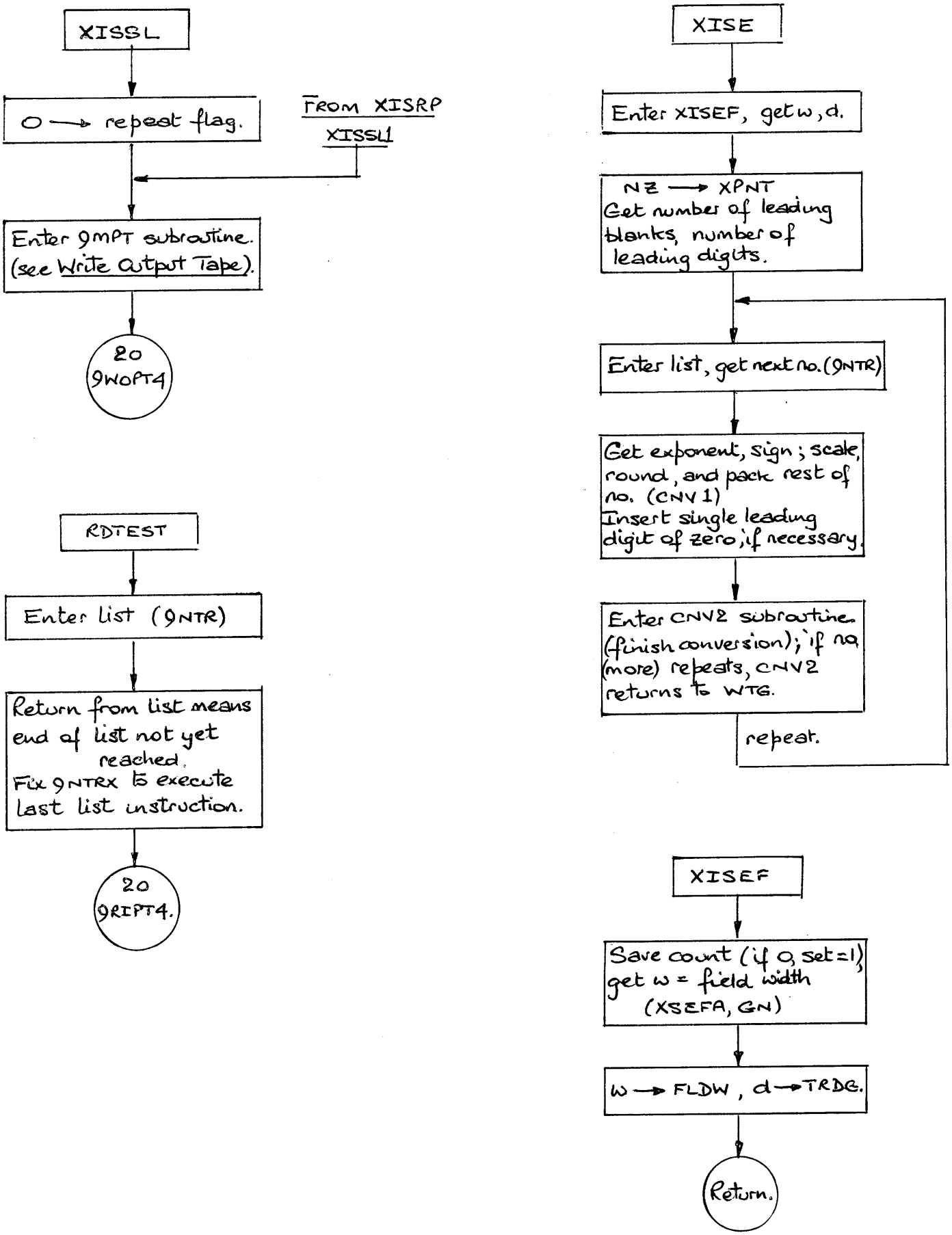
Are repeat ctr. and repeat flag both zero ?

No → 20 WTG

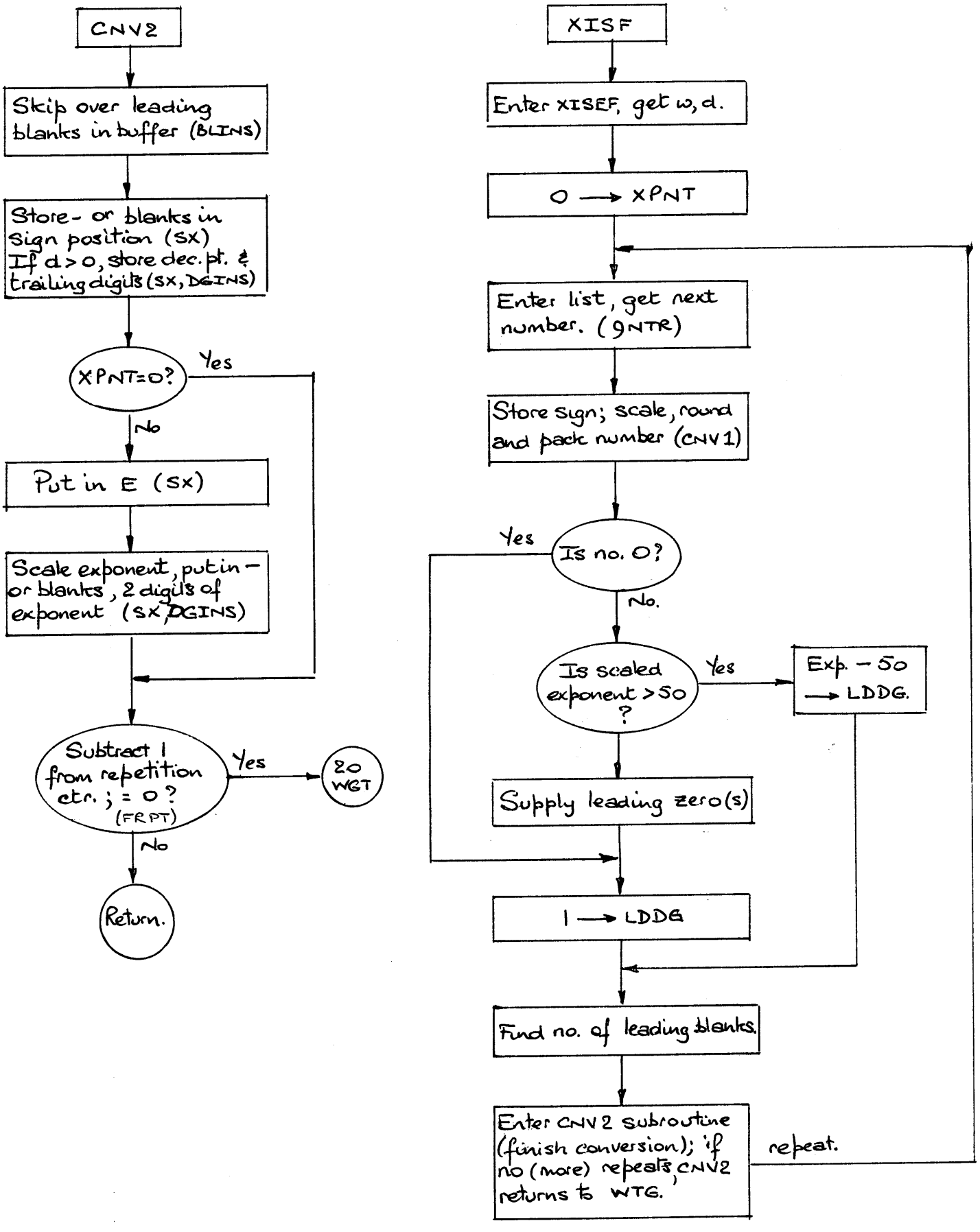
Yes

Save format index words and parenthesis level for recycling thru format. → 20 WTG.

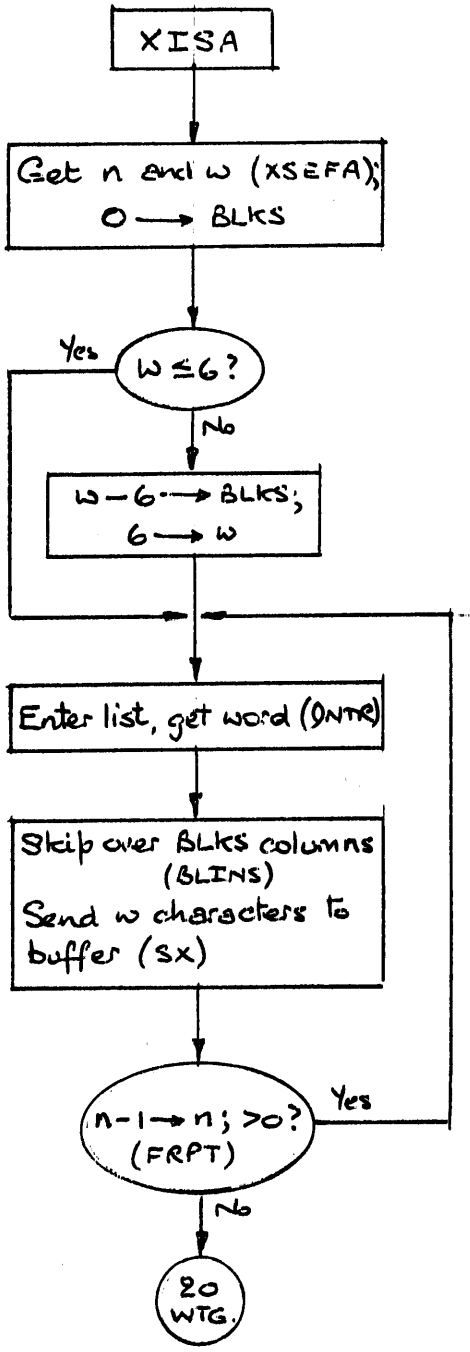
## ROUTINES USED by 9RIPT & 9WOPT (cont.)

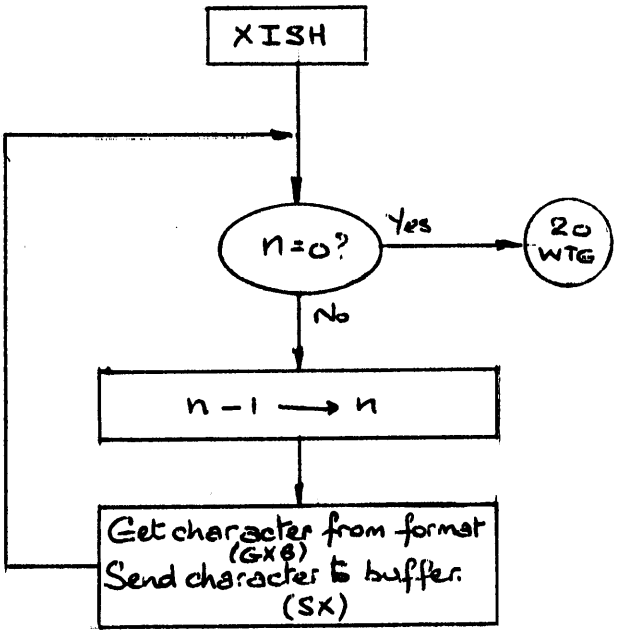


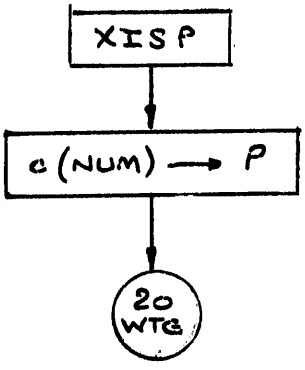
## ROUTINES USED by 9RIPT & 9WOPT (cont).

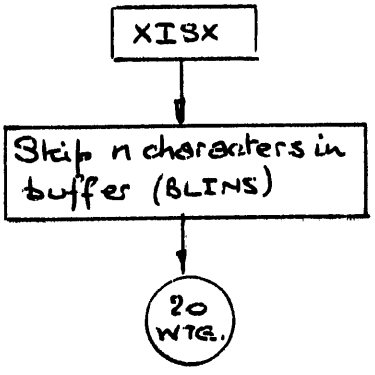


## ROUTINES USED by 9RIPT & 9WOPT (cont.)



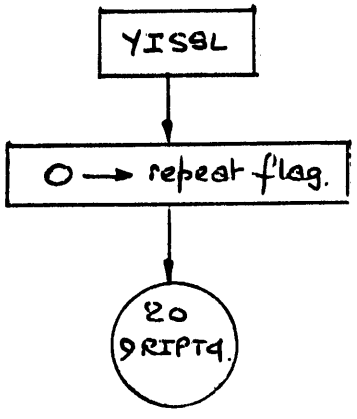



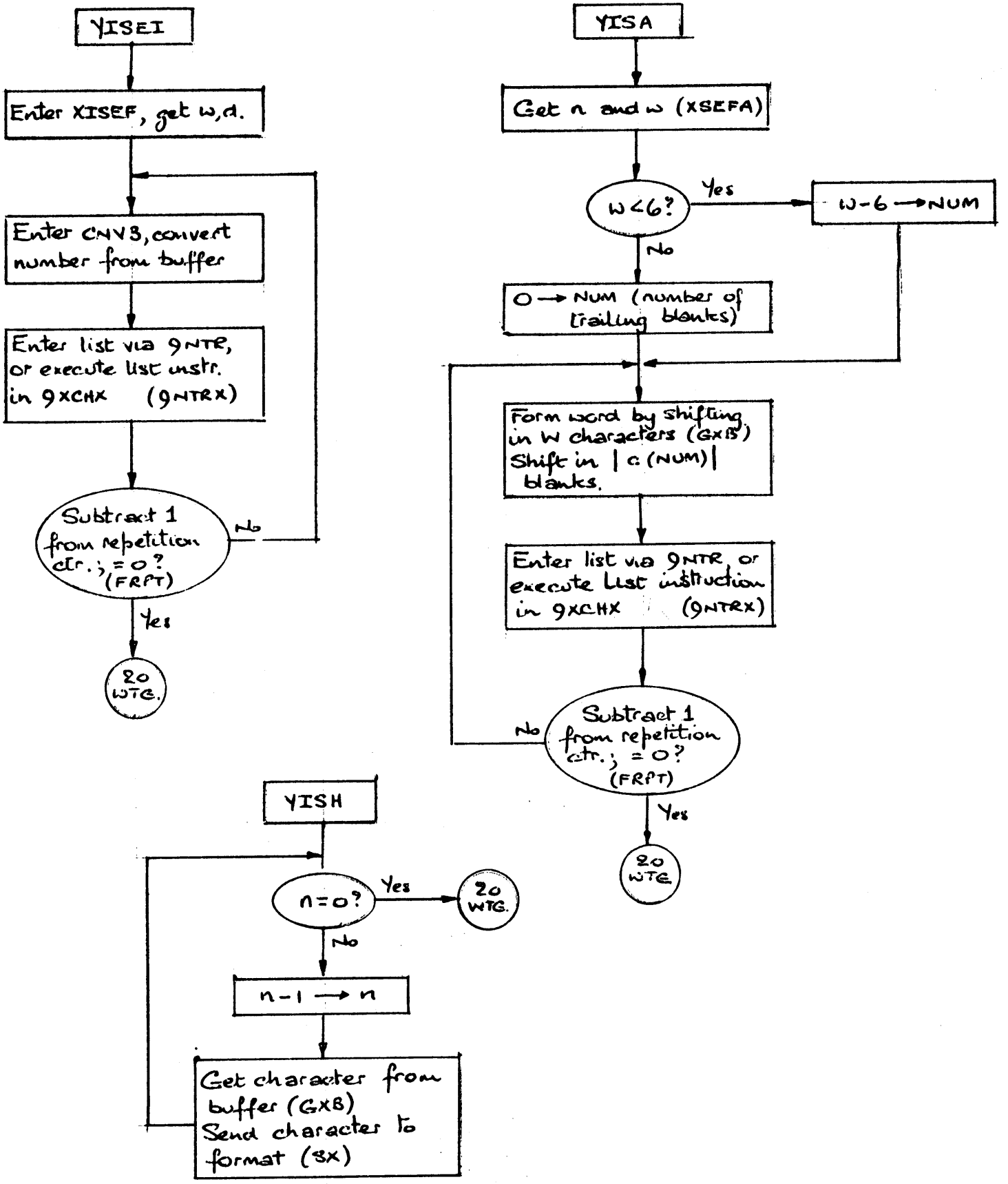

XISI

Flow chart for XISI is the same as for XISF, except:

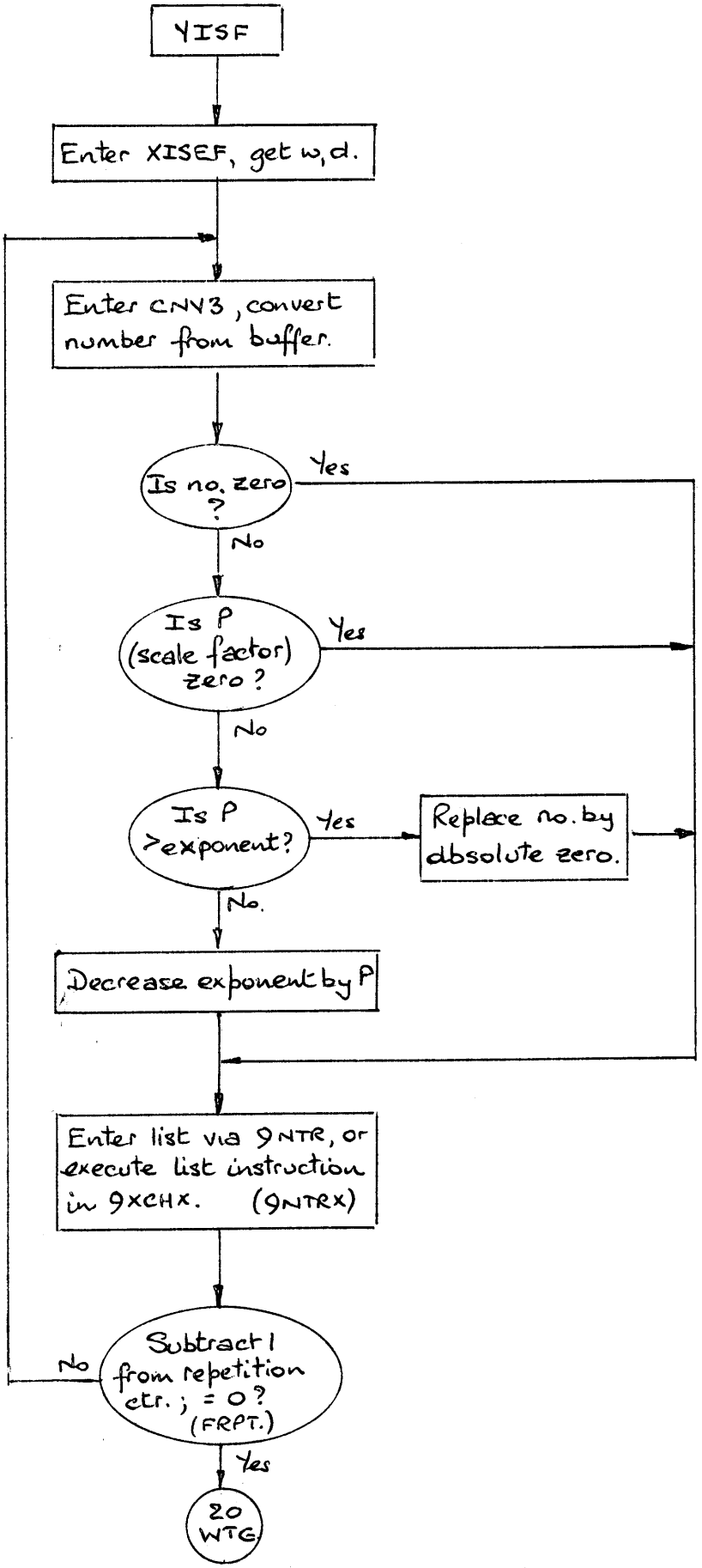
1. No scaling is done.
2. The exponent is assumed to be > 50 for non-zero numbers.



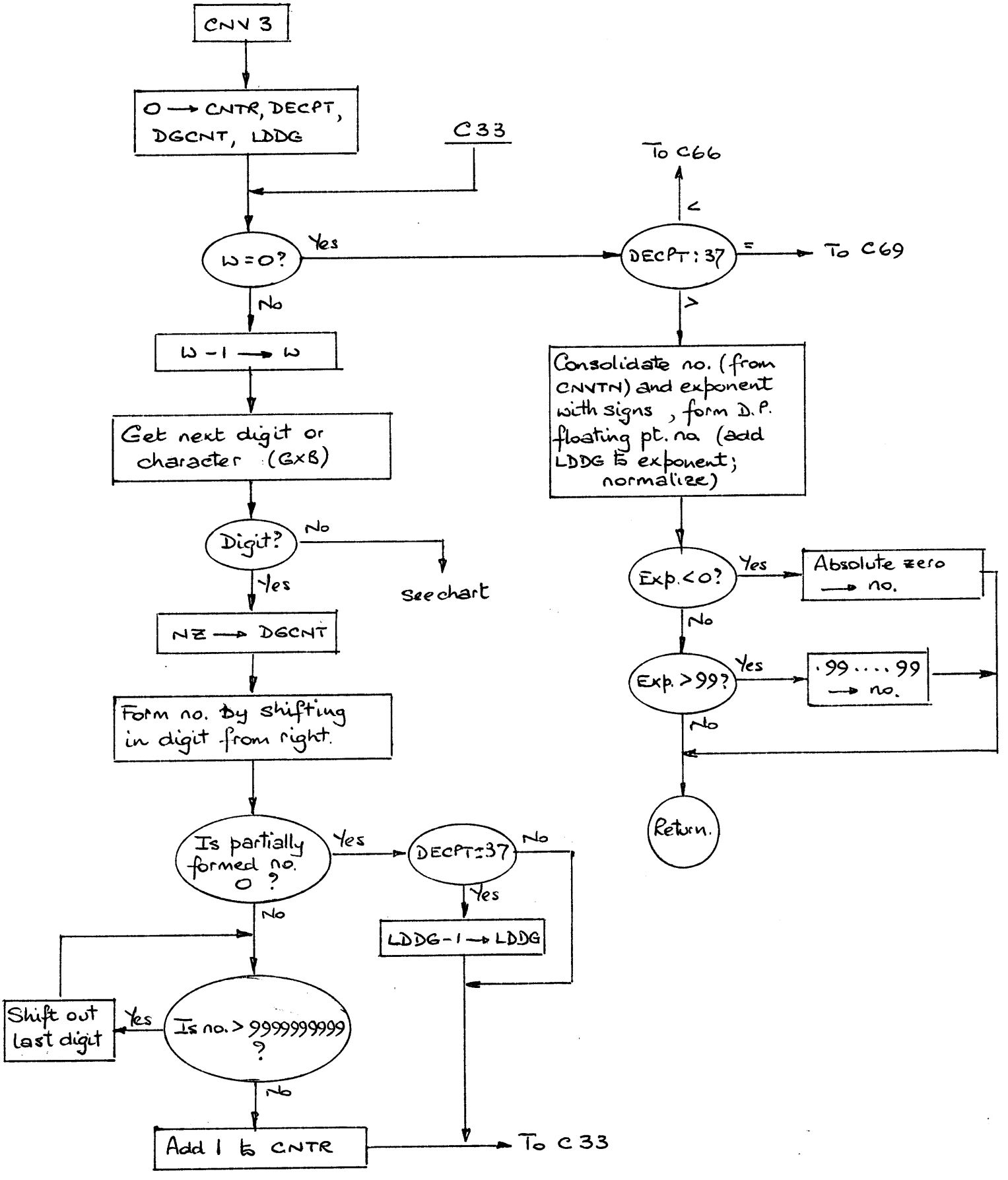
ROUTINES USED by 9RIPT & 9WOPT (cont.)



## ROUTINES USED by 9RIPT & 9WOPT (cont.)



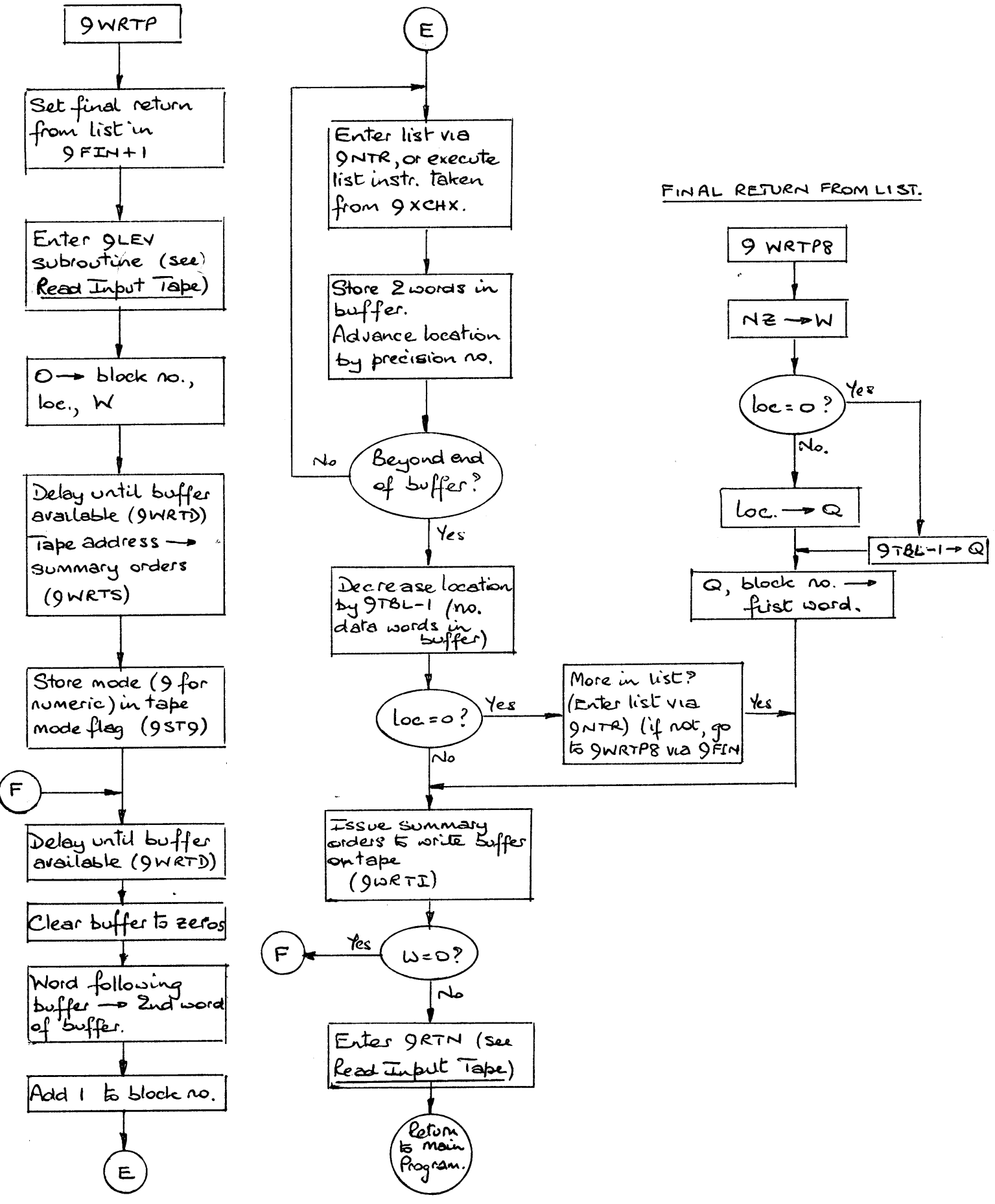
## ROUTINES USED by 9RIPT & 9WOPT (cont.)



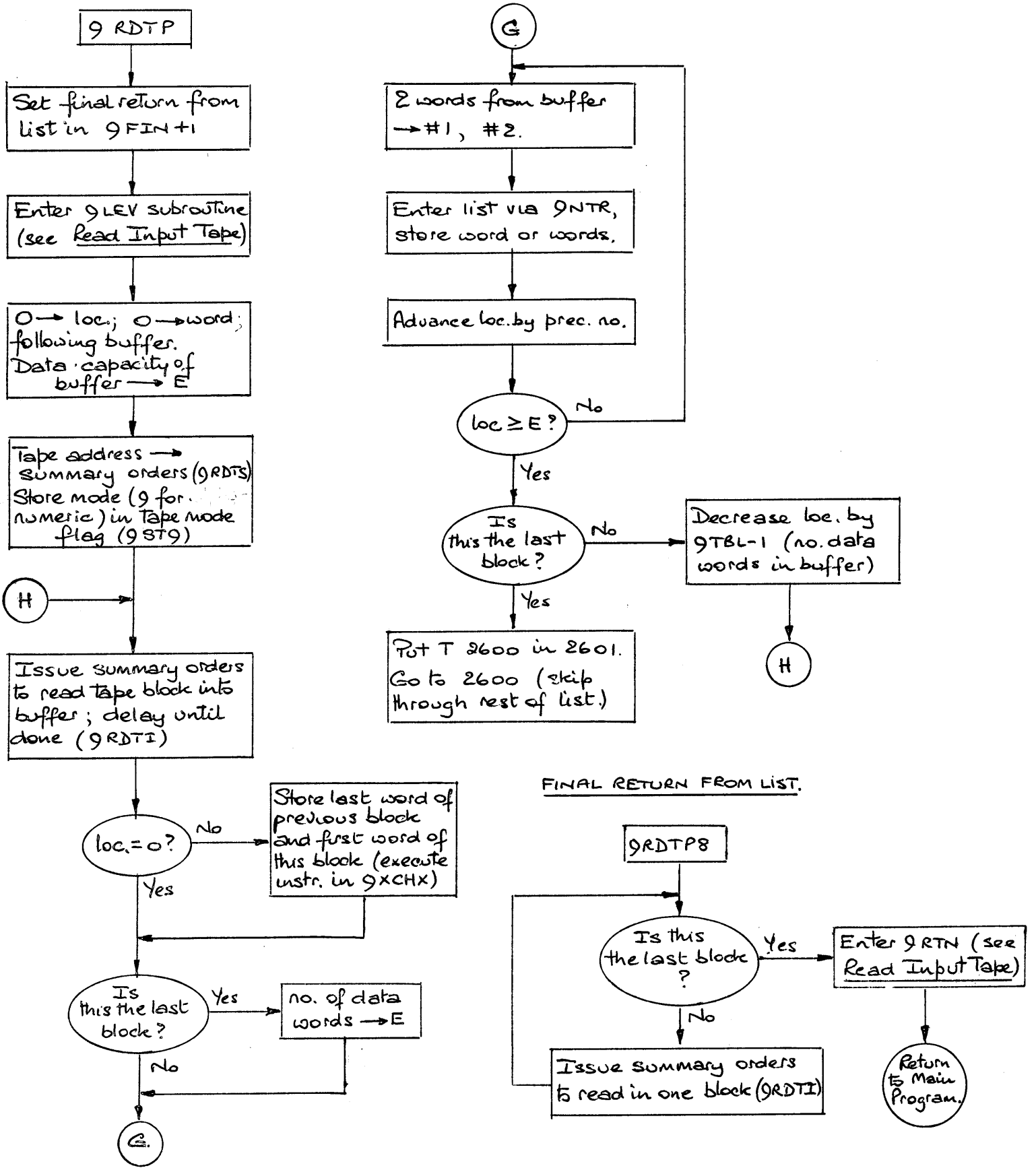
## ROUTINES USED by 9RIPT & 9WOPT (cont.)

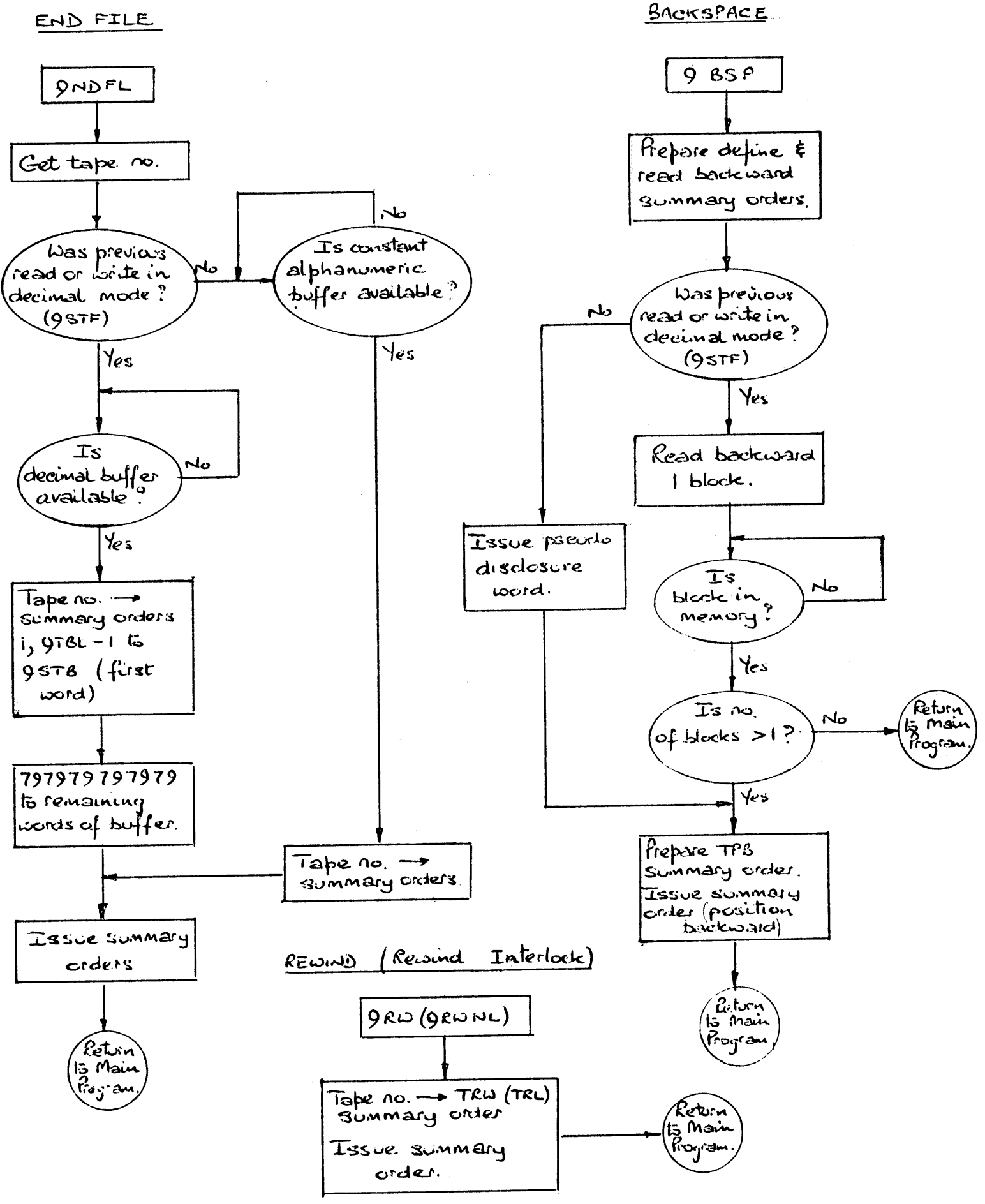
| Char. \ DECPT | 0 | 37 | > 37 |
|---|---|---|---|
| Blank. | If CNTR = 0, → C33<br>If CNTR ≠ 0, → C39 | C39<br>Treat blank as a zero; → C33 | → C39 |
| — | If DECNT = 0, make sign of no. —; → C33<br>If DECNT ≠ 0, make exp. —; → C66 | Make exp. —;<br>→ C69 | Make exp. —;<br>→ C33 |
| + | If DECNT = 0, → C33<br>If DECNT ≠ 0, → C66 | C69<br>Store no. in CNVTN;<br>make DECPT > 37; → C33 | → C33 |
| . | 37 → DECPT;<br>C(CNTR) → LDDG;<br>→ C33 | ——— | ——— |
| E | C66<br>CNTR − TRDG → LDDG<br>(if neg., set = 0); → C69 | → C69 | → C33 |

# WRITE TAPE


9WRTP
Set final return from list in 9FIN+1
Enter 9LEV subroutine (see Read Input Tape)
0 → block no., loc., W
Delay until buffer available (9WRTD)
Tape address → summary orders (9WRTS)
Store mode (9 for numeric) in tape mode flag (9ST9)
F
Delay until buffer available (9WRTD)
Clear buffer to zeros
Word following buffer → 2nd word of buffer.
Add 1 to block no.
E
E
Enter list via 9NTR, or execute list instr. taken from 9XCHX.
Store 2 words in buffer. Advance location by precision no.
Beyond end of buffer?
No
Yes
Decrease location by 9TBL-1 (no. data words in buffer)
loc=0?
Yes
No
More in list? (Enter list via 9NTR) (if not, go to 9WRTP8 via 9FIN
Yes
Issue summary orders to write buffer on tape (9WRTI)
W=0?
Yes
F
No
Enter 9RTN (see Read Input Tape)
Return to main Program.
FINAL RETURN FROM LIST.
9WRTP8
NΣ → W
loc=0?
Yes
No.
loc. → Q
9TBL-1 → Q
Q, block no. → list word.

# READ TAPE


9 RDTP
Set final return from list in 9FIN+1
Enter 9LEV subroutine (see Read Input Tape)
0 → loc.; 0 → word following buffer. Data capacity of buffer → E
Tape address → summary orders (9RDTS) Store mode (9 for numeric) in Tape mode flag (9ST9)
H
Issue summary orders to read Tape block into buffer; delay until done (9RDTI)
loc.=0?
No
Store last word of previous block and first word of this block (execute instr. in 9XCHX)
Yes
Is this the last block?
Yes
no. of data words → E
No
G.
G
Σ words from buffer → #1, #2.
Enter list via 9NTR, store word or words.
Advance loc. by prec. no.
loc ≥ E?
No
Yes
Is this the last block?
No
Decrease loc. by 9TBL-1 (no. data words in buffer)
H
Yes
Put T 2600 in 2601. Go to 2600 (skip through rest of list.)
FINAL RETURN FROM LIST.
9RDTP8
Is this the last block?
Yes
Enter 9RTN (see Read Input Tape)
No
Issue summary orders to read in one block (9RDTI)
Return to Main Program.


END FILE
9NDFL
Get tape no.
Was previous read or write in decimal mode? (9STF)
No
Is constant alphanumeric buffer available?
No
Yes
Yes
Is decimal buffer available?
No
Yes
Tape no. → summary orders i, 9TBL-1 to 9STB (first word)
7979797979 to remaining words of buffer.
Tape no. → summary orders
Issue summary orders
Return to Main Program.
BACKSPACE
9 BSP
Prepare define & read backward summary orders.
Was previous read or write in decimal mode? (9STF)
No
Yes
Read backward 1 block.
Issue pseudo disclosure word.
Is block in memory?
No
Yes
Is no. of blocks >1?
No
Return to Main Program.
Yes
Prepare TPB summary order. Issue summary order (position backward)
Return to Main Program.
REWIND (Rewind Interlock)
9RW (9RWNL)
Tape no. → TRW (TRL) summary order
Issue summary order.
Return to Main Program.

# PHASE I

The main function of Phase I is to analyze the initial source program and to output standard units of information (items), describing the input. These items are listed below with details of their construction. The source program is read from the input drum via the FILE routine. If the word in the Communication region indicates, the input line is converted to the LSC internal character code prior to analyzing the statement. SAL code is not analyzed except for recording the label of a SAL instruction.

The analysis of the source statement is sufficient for Phase I to output an item for every meaningful feature of the input and such that no other phase need refer to the source input again. The items output by Phase I are placed in different files according to their type and the original source lines (in the LSC internal character code) are filed for editing when compiling is complete. The items generated are filed by the FILE routine while the scan is in process. Hence, no restrictions are made regarding the number of lines (continuations) used to represent a single statement except that space on the drums must exist for the file being formed.

Phase I is broken down into many parts, some of which are described below; flowcharts of these sections are included. Some statements receive more thorough analysis in Phase I than others. In general, Phase I does not make an analysis of the program as a whole. In particular:

1. No distinction is made between subscripts and arguments.

2. The arithmetic mode implied by the name of a variable is ignored.

3. The appearance of an induction variable (explained later) in arithmetic expressions or subscripts is ignored.

4. Information contained in one expression or statement is not carried along when processing later statements, except in a DO statement (see 2. below).

The analysis of Phase I is best summarized by noting the list of items which it generates. The output of Phase I has been designed in such a manner as to facilitate the analysis of Phase II, III, and IV. On the other hand, the functions of Phases II, III, and IV make it unnecessary for Phase I to keep lengthy tables describing variables and searching for information about a variable before outputting items.

Some areas of Phase I deserve special attention:

1. Arithmetic Scan

   During Phase I, an arithmetic expression is scanned from left to right and items are assigned a subsequence number in such a way as to cause the expression to appear in the Polish (Lukasiewicz) notation in later phases. The major sequence number (the lead five digits) is the same for all items in the expression. The included flowchart contains details on how the subsequence number (SS) is established.

2. Determination of a "DO range"

   A DO statement is the only statement which causes Phase I to store inter-statement information. Whenever a statement:

   $$\text{DO } \mu,\ I = b,c,d$$

   is encountered, the quantities: I, b, c, and d are output immediately. The statement sequence number, S, associated with these items is stored (hereafter referred to as DS) along with the referred to statement name, $\mu$, and with I. The range is determined when a statement named $\mu$ is encountered; at this time a second item for I is output (a "20" type item). Prior to the arrival of the statement $\mu$ (or to another DO statement) DS, as described above, is associated with all items output by Phase I that have DS indicated in the item.

3. The occurrence of a "list" in the source program (input-output list or "list" = expression, etc.) causes Phase I, in some cases, to output items similar to those of a DO. For example:

```
( ( ( A ( I, J, K), I = 1, 10), J = 1, 5), K = 1, 10) = expression
```

will generate:

```
1.    (Expression evaluation)
      DO β, K = 1, 10
      DO β, J = 1, 5
      DO β, I = 1, 10
```

4. Analyzing SAL statements:

A. An "S" in the first column of a statement denotes SAL coding, all names interpreted by LSC must be legal LSC symbols.

B. Output special continue item for all names of SAL code.

C. Output a special continue item for the first statement in a block of SAL coding (ID = 11) to F4.

```
W1  Seq. , O , 11
W2         O
W3  DS , L , P
```

D. The induction variable in a DO loop is made available to SAL programs.

E. Any exits from a block of SAL coding to another block of SAL coding or to an LSC statement must be indicated in the following statement:

SAL EXITS ($N_1$, $N_2$, . . . $N_k$)
where $N_i$ is the symbol to which a transfer is made
outside of the following block of SAL coding.

F. Phase I will recognize the LABEL line and a DIRECTORY; in general these items remain unprocessed and are assumed to be acceptable SAL items.

APPENDIX - A

Sequence Counters

During Phase I, a set of sequence counters is maintained in order that every piece of information extracted from the source program (an item) can have a number associated with it. The sequence counters (ST, SD, SE, etc.) each have a range of numbers exclusive of the others. The number itself will be an integer less than $10^5$.

1. ST is the sequence counter for the following statements:

   INTEGER
   FLOATING
   DOUBLE PRECISION
   INTEGER FUNCTION
   FLOATING FUNCTION
   DOUBLE PRECISION FUNCTION
   PARAMETER

2. SC, SF, and SA are used for:

   CONSTANT
   FORMAT
   Arithmetic functions, respectively

3. SD is the sequence counter for DIMENSION statement items.

   Example: DIMENSION A (I, B), B(7)
   DIMENSION C (M)

Assuming that the current value of the DIMENSION counter is SD, the first statement, as a whole, has SD as the sequence number. (This is for printing purposes). Items A, I and B have sequence number SD + 1. B and 7 have SD + 2. The second DIMENSION statement has sequence number SD + 3. Items C and M have sequence number SD + 4. The next DIMENSION statement encountered would have sequence number SD + 5, etc. Any error items get the same sequence number as the item which was in error.

4. SE is the sequence counter for EQUIVALENCE and COMMON statement items.

Example:

```
EQUIVALENCE ( A (I), B(5) ), (C, D)
COMMON D, E, F
```

Assuming that the current value of the sequence counter is SE, the following assignments of sequence numbers will be made:

| Item | Sequence Number | Subsequence Number |
|---|---|---|
| EQUIVALENCE Statement | SE | 0 |
| A, I, B, 5 | SE + 1 | 1,2,3,4 |
| C, D | SE + 2 | 1,2 |
| COMMON Statement | SE + 3 | 0 |
| D | SE + 4 | 0 |
| E | SE + 5 | 0 |
| F | SE + 6 | 0 |

Note: Any error items get the same sequence number as the item which was in error.

5. S is the sequence counter for the items in the remainder of the statements.

In general, the entire sequencing number (10 digits) is made up of two parts, the Sequence number described above and a Subsequence number, which gives the order (or desired order) of the items within a group of related items. Variables in a list get numbered

sequentially. Variables in an expression all receive the same sequence number, but receive subsequence numbers in such a way as to transform the expression into a parenthesis free notation when the items are sorted on the sequencing number. Statements in the print file have sequence numbers which differ by at least 1. Error items always have a sequence number equal to the sequence number of the item which caused the error.

6. SB is used for the FUNCTION and SUBROUTINE statement.

Sequencing of Arithmetic Statements

S is the sequence counter used for arithmetic statements. Let n be the value of the sequence counter S associated with an arithmetic statement:

| S | Statement Name | Statement |
|---|---|---|
| n | JOE | X(I), Y = A + B |

Sequence and subsequence numbers are assigned in the following manner:

| | Item | Sequence S | Subsequence SS |
|---|---|---|---|
| | line | n | 0 |
| | JOE | n + 1 | 0 |
| | X | n + 2 | 0 |
| * | X | | 2 |
| | : | n + 3 | 1 |
| | I | | 100 |
| * | Y | n + 2 | 0 |
| | Y | n + 4 | 0 |
| | + | | 49 |
| | A | n + 2 | 50 |
| | B | | 100 |

* Arithmetic MODE Item

Note that the sequencing numbers of the list items are greater than any of the expression items.

Sequencing of Arithmetic Functions

SA is the sequence counter used for arithmetic functions. Let n be the value of the sequence counter SA associated with the arithmetic function:

| SA | Statement Name | Statement |
|---|---|---|
| n | SAM | FIRSTF (A, B) = 2A + B**2 |

Sequence and subsequence numbers are assigned in the following manner:

| Item | Sequence | Subsequence |
|---|---|---|
| line | n | 0 |
| SAM | S | 0 |
| FIRST | n + 1 | 0 |
| FIRST | n + 2 | 0 (expression mode item) |
| A | n + 1 | 1 |
| B | " | 2 |
| 2 | n + 2 | 100 |
| * | " | 99 |
| A | " | 150 |
| + | " | 98 |
| B | " | 200 |
| ** | " | 199 |
| 2 | " | 250 |

Note: The name "SAM" was used here to show that a continue item will be output for it with sequence number S (the executable statement counter). Normally a statement name is not given, since reference to the arithmetic function will be FIRST( x,y) or FIRSTF(x,y).

PHASE I APPENDIX B

ITEM FORMATS

The following formats represent items which are output by Phase I, II, and III. Outputs from Phase II and III are included here for completeness. The File 2 items are on the left, File 3 in the center, and File 4 on the right side of each page.

PARAMETER $V_1 = A_1$ , $V_2 = A_2$ , . . . $V_p = A_p$

| | | | | | |
|---|---|---|---|---|---|
| $V_j$ | | | ST | 0 | 01 |
| ST | 0 | 01 | 012 | 0000 | 0ffff |
| $A_j$ | | | $A_j$ | | |

$A_j$ is truncated to a floating point integer

CONSTANT X = numeric

| | | | | | |
|---|---|---|---|---|---|
| | | | SC | 1 | 45 |
| | | | 0b2 | 000 0 | 0000 |
| | | | high order part of numeric | | |
| X | | | SC | 0 | 38 |
| SC | 0 | 38 | 0b1 | 0000 | 0ffff |
| low order part of numeric | | | low order part of numeric | | |

INTEGER $V_1, \ldots, V_p$ (b = 1)

FLOATING $V_1, \ldots, V_p$ (b = 2)

DOUBLE PRECISION $V_1, \ldots V_p$ (b = 3)

| | | |
|---|---|---|
| $V_j$ | | |
| ST | 0 | 06 |
| 0b1 | 0000 | 00000 |

INTEGER FUNCTION $V_1, \ldots, V_p$ (b = 1)

FLOATING FUNCTION $V_1, \ldots, V_p$ (b = 2)

DOUBLE PRECISION FUNCTION $V_1, \ldots, V_p$ (b = 3)

$V_j$

| ST | 0 | 07 |
|---|---|---|
| 0b4 | 0000 | 0000 |

The mode digit "b" is assigned to all references to $V_j$ by Phase III when the reference items are sent to File 4.

SUBROUTINE N ( $M_1, \ldots, M_p$ )

N

| SB | 0 | 32 |
|---|---|---|
| 0 | | 0 p |

| SB | 0 | 32 |
|---|---|---|
| 0 0 4 | 0000 | 0ffff |
| 0 | | 0 p |

M

| SB | j | 33 |
|---|---|---|
| 0 | | 0 |

| SB | j | 33 |
|---|---|---|
| 9 b c | 0000 | 0ffff |
| 0 | | 0 |

FUNCTION N ( $M_1, \ldots, M_p$ )

N

| SB | 0 | 34 |
|---|---|---|
| 0 | | 0 p |

| SB | 0 | 34 |
|---|---|---|
| 0 0 4 | 0000 | 0ffff |
| 0 | | 0 p |

M

| SB | 0 | 42 |
|---|---|---|
| 0 | | 0 |

| SB | 0 | 42 |
|---|---|---|
| 9 b c | 0000 | 0ffff |
| 0 | | 0 |

RETURN

```
S                0          49
6 00     0014          00000
0 ———————————————————————— 0
```

ARITHMETIC STATEMENT FUNCTION  N F ( $M_1$ , . . . , $M_p$ ) = . . .

```
N
SA          0          23
0 ———————————————— 0 p
```

```
SA          0          23
5 6 4    0 000       0ffff
0 ———————————————————— 0p
```

```
Mj
SA          j          24
0 ———————————————— 0
```

(j) replaces the (uffff) digits of references to ($M_j$) appearing on the right hand side of this statement. (SA +1$^j$ ) is the sequence of items in the arithmetic expression on the right.

CALL N ( M, . . . , $M_p$ )

```
N
S           1          30
DS          L          p
```

```
S           1          30
0 b 4     0000       0ffff
  DS        L          P
```

(LIB request item initiated by Phase III).

```
          0          5          44
0 ———————————————————————— 0
          N
```

If ( $M_j$) is a single symbol:

```
Mj
S          ss          31
DS         L           P
```

```
S          ss          31
abc     d 000       0ffff
c=1 0 ———————————————— DP
c=2  aj
c=3  DS    L         P
```

If ( $M_j$ ) is a Hollerith argument:

File 93

| | | | |
|---|---|---|---|
| w1 | SF 0 03 | S | ss 37 |
| w2 | ALPH | 000 | 0000 0ffff |
| w3-11 | characters in the Hollerith argument. | SF | 0———0 |

If ( $M_j$ ) is an arithmetic expression, output Polish string for arithmetic.

DIMENSION $V(A_1, \ldots, A_p), V^1(A^1_1, \ldots A^1_p), \ldots$

$V^i$

| | | |
|---|---|---|
| SD | 0 | 02 |
| 0 | ——— | 0p |

| | | |
|---|---|---|
| SD | 0 | 02 |
| a b 1p | e e e | 0ffff |
| 0 | ——— | 0p |

If a dimension is specified as a constant:

| | | |
|---|---|---|
| SC | j | 45 |
| 012 | 0000 | 00000 |
| $A_j$ | | |

If a dimension is specified as a parameter:

$A_j$

| | | |
|---|---|---|
| SD | j | 03 |
| 0 | ——— | 0 |

| | | |
|---|---|---|
| SD | j | 45 |
| 012 | 0000 | 00000 |
| 0 | ——— | 0 |

EQUIVALENCE $(V_1(A^1_1, \ldots, A^{d1}_p), \ldots, V_n(A^1_n, \ldots, A^{dn}_n)), \ldots$

$V_j$

| | | |
|---|---|---|
| SE | i | 04 |
| 0 | ——— | 0 |

| | | |
|---|---|---|
| SE | j | 04 |
| a b l d | eee | 0ffff |
| 0 | ——— | 0 |

a = 2 if not common

a = 3 if common

For each ( $A_j^k$ ) specified as a constant:

| SE | j | 45 |
|---|---|---|
| 0 1 2 | 0000 | 00000 |

$A_j^k$

For each ( $A_j^k$ ) specified as a parameter:

$A_j^k$

| SE | j | 03 |
|---|---|---|
| 0 | | 0 |

| SE | j | 45 |
|---|---|---|
| 0 1 2 | 0000 | 00000 |

$A_j^k$

COMMON $V_1, \ldots V_p$

$V_j$

| (SE+j) | 0 | 05 |
|---|---|---|
| 0 | | 0 |

| (SE+j) | 0 | 05 |
|---|---|---|
| a b l | deee | 0ffff |
| 0 | | 0 |

a = 1 if not equivalent

a = 3 if equivalent

ARITHMETIC STATEMENTS

V in an expression

| V | | |
|---|---|---|
| S | ss | 14 |
| 0 | | 0 |

| S | ss | 14 |
|---|---|---|
| a b l | 0000 | 0ffff |
| 0000 | AAA | DP |

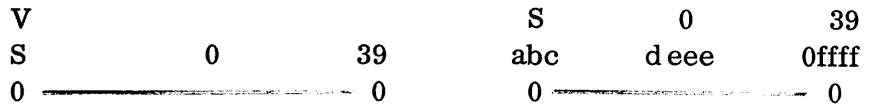
AAA = the subprogram argument number if V is an argument of a subprogram that is being compiled.

Store: V =

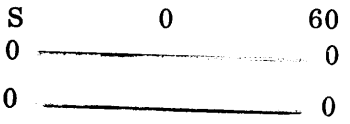
| V | | |
|---|---|---|
| S | 0 | 21 |
| 0 | | 0 |

| S | 0 | 21 |
|---|---|---|
| a b l | deee | 0ffff |
| 0000 | AAA | DP |

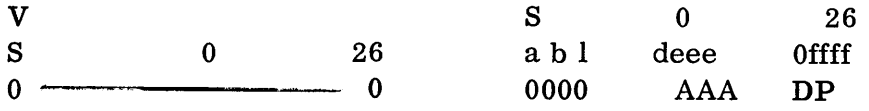
Mode of expression item




List Item



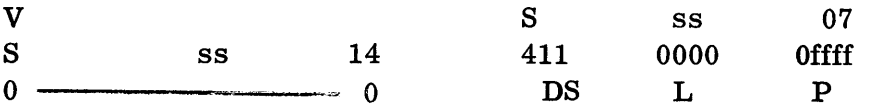

V in an input list




V in an output list



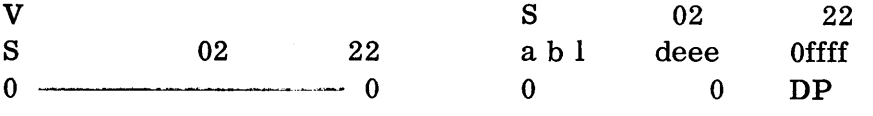

If V is a reference to the induction variable within the DO loop, Phase III changes the item to reflect this type of reference (e.g.)




V (subscript) in an expression




V (subscript) STORE: V ( ) =



Store expression item

| | | | | | |
|---|---|---|---|---|---|
| V | | | S | 0 | 39 |
| S | 0 | 39 | abl | deee | 0ffff |
| 0 | 0 | | 0 | 0 | |

V (subscript) in an input list

| | | | | | |
|---|---|---|---|---|---|
| V | | | S | 02 | 27 |
| S | 02 | 27 | a b l | deee | 0ffff |
| 0 | 0 | | 0 | 0 | DP |

V (subscript) in an output list

| | | | | | |
|---|---|---|---|---|---|
| V | | | S | 02 | 29 |
| S | 02 | 29 | a b l | deee | 0ffff |
| 0 | 0 | | 0 | 0 | DP |

DO L I = $N_1$ , $N_2$ , $N_3$ (DO loop)

( - - - , I = $N_1$ , $N_2$ , $N_3$) (List loop)

implied name n

| | | | | | |
|---|---|---|---|---|---|
| I | | | S | 1 | 19 |
| S | 1 | 19 | a b c | 0000 | 0ffff |
| DS | L | P | O | S(e) | 00 |

| | | | | | |
|---|---|---|---|---|---|
| I | | | S(e) | 0 | 20 |
| S(e) | 0 | 20 | 4 1 1 | 0000 | 0ffff |
| DS | L | P | O | S(e) | 00 |

This item is output from Phase I when statement L is encountered. S(e) is greater than any value of S assigned to items in statement L.

If ($N_j$) is specified as a variable:

| | | | | | |
|---|---|---|---|---|---|
| | $N_j$ | | | S | j+1 | 46 |
| S | j+1 | 46 | | 411 | 0000 | 0ffff |
| DS | L | P | | DS | L | P |

If ($N_j$) is specified as a constant:

| | | |
|---|---|---|
| S | j+1 | 45 |
| 412 | 0000 | 0ffff |
| $N_j$ | | |

CONTINUE - executable statement

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| name | | | name | | | S | 0 | 08 |
| S | 0 | 08 | (S - 4) | S | 08 | 003 | 0000 | 0ffff |
| DS | L | P | DS | L | P | DS | L | P |

(if unreferenced do not send to File 4)

GO TO L

| | | | | | |
|---|---|---|---|---|---|
| | L | | | S | 1 | 16 |
| | S | 1 | 36 | 003 d | 000 | 0ffff |
| | DS | L | P | S c o n t | 0——— | ———0 |

(if not a drop-out)

| | | |
|---|---|---|
| S | 1 | 36 |
| 003 | 0000 | 00000 |
| 0 ——— | | ———0 |

(if a drop-out)

GO TO ( $N_1$, . . . $N_k$ ) , M (computed GO TO)

| | | |
|---|---|---|
| $N_j$ | | |
| S | j+1 | 16 |
| DS | L | P |

| | | |
|---|---|---|
| S | j+1 | 16 |
| 003 | d000 | 0ffff |
| S(c) | 0 ——— | 0 |

S(c) = sequence of statement name

| | | |
|---|---|---|
| M | | |
| S | 1 | 14 |
| 0000 | AAA | DP |

| | | |
|---|---|---|
| S | 1 | 4 |
| 003 | d000 | 0ffff |
| 0000 | AAA | DP |

GO TO M, ( $N_1$, . . . , $N_k$) (assigned GO TO)

| | | |
|---|---|---|
| M | | |
| S | 1 | 17 |
| DS | L | P |

| | | |
|---|---|---|
| S | 1 | 17 |
| 003 | 1000 | 0ffff |
| DS | L | P |

| | | |
|---|---|---|
| $N_j$ | | |
| S | j+1 | 18 |
| M | | |

| | | |
|---|---|---|
| M | | |
| S | S(c) | 18 |
| DS | L | P |

ASSIGN N, TO M

| | | |
|---|---|---|
| N | | |
| S | 1 | 12 |
| M | | |

| | | |
|---|---|---|
| N | | |
| S | 1 | 12 |
| M | | |

| | | |
|---|---|---|
| M | | |
| (S-2) | S(c) | 13 |
| DS | L | P |

| | | |
|---|---|---|
| M | | |
| S | 2 | 13 |
| DS | L | P |

| | | |
|---|---|---|
| S | 1 | 12 |
| 003 | 1000 | 0ffff |
| M | | |

| | | |
|---|---|---|
| S | 2 | 13 |
| abl | 1000 | 0ffff |
| 0000 | AAA | DP |

CONTINUE - (FORMAT statement)

| name | | | name | | |
|---|---|---|---|---|---|
| S | 0 | 10 | (S - 3) | S | 10 |
| DS | L | P | DS | L | P |

READ INPUT TAPE I, N, list

READ INPUT TAPE I, N, list

WRITE OUTPUT TAPE I, N, list

READ PAPER TAPE I, N, list

PUNCH PAPER TAPE I, N, list

TYPEWRITE I, N, list

READ TAPE I, list

WRITE TAPE I, list

REWIND I

REWIND INTERLOCK I

BACKSPACE I

ENDFILE I

READ N, list

PRINT N, list

PUNCH N, list

READ DRUM I, J, list

WRITE DRUM I, J, list

DEFINE LARC DRUM I, J, K, L

POSITION HEAD I, J

ADVANCE HEAD I

BACK HEAD I

READ LARC DRUM I, J, list

WRITE LARC DRUM I, J, list

Reference to FORMAT "N"

```
N
S          ss          15
0----------------------0
```

```
S          ss          15
005   d    000         0ffff
0----------------------0
```

For those values of ( I, J )specified symbolically:

```
name
S          ss          14
0----------------------0
```

```
S          ss          14
a b l      0 000       0ffff
0000       AAA         DP
```

CONTINUE - SAL label

```
name                   name
S        0      11     (S-4)     S      11
DS       L      P       DS       L      P
```

Beginning of SAL block

```
S          0           11
0----------------------0
 DS        L           P
```

SAL EXITS ($V_1$ , . . . , $V_k$)

```
 Vj
S          j           35
 DS        L           P
```

IF ( - - - - - ) $N_1$, $N_2$, $N_3$

If ($N_j$) is missing

```
S          j           36
003        0000        0000
0----------------------0
```

If ($N_j$) is stated:

```
    N
     j
    S          j          36        S          j          16
    DS         L          P         003        d000       0ffff
                                    S(c)

                                    (if not drop-out)

                                    S          j          36
                                    003        0000       00000
                                    0----------------------0

                                    (if drop-out)
```

SENSE LIGHT I

IF (SENSE LIGHT I) $N_1$, $N_2$

IF ( SENSE SWITCH I) $N_1$, $N_2$

IF DIVIDE CHECK $N_1$, $N_2$

IF ACCUMULATOR OVERFLOW $N_1$, $N_2$

IF QUOTIENT OVERFLOW $N_1$, $N_2$

If ( I ) is specified numerically:

```
                                    S          1          45
                                    012        0000       00000
                                    I
```

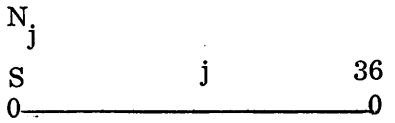
If ( I ) is specified symbolically:

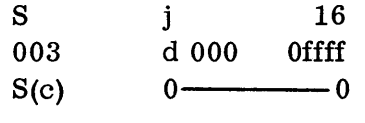
```
    I
    S          1          03        S          1          45
    0----------------------0        012        0000       00000
                                    PARAMETER ,           I
```

If ($N_j$) is missing:

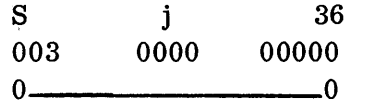
```
                                    S          j          36
                                    003        0000       00000
                                    0----------------------0
```

If ($N_j$) is stated:




(if not drop-out)




(if drop out)

SUCCESSOR item (ID = 03)

(See Phase III)

Ignore item (ID = 50)

(See Phase III)

Adjacent register flag item

| | | | |
|---|---|---|---|
| (When one expression must be in a register adjacent to another, as in the relational - IF statements. | S | SS<br>O<br>O | 48 |

TO and FROM items

ID = 40 ID = 43

(See Phase III)

Summary of Identification Digits Assignment (Assigned by Phase I)

The preceding items are all identified by a two digit number, the ID. File 2 and File 3 items have their ID in the right-most digits of the second word (W2); File 4 has the ID in the first word (w1).

| ID | DEFINITION |
|---|---|
| 01 | parameter |
| 02 | dimensioned variable |
| 03 | name referencing a parameter |
| 04 | equivalenced variable |
| 05 | common |
| 06 | mode specifier (variable) |
| 07 | mode specifier (function) |
| 08 | executable continue |
| 09 | non-executable continue |
| 10 | format continue |
| 11 | SAL continue |
| 12 | assignor (assign A, |
| 13 | assignee to B) |
| 14 | unparenthesised variable reference |
| 15 | format reference |
| 16 | statement name reference |
| 17 | must be assigned variable |
| 18 | listed assignment |
| 19 | induction variable - begin loop |
| 20 | induction variable - end loop |
| 21 | stored variable |
| 22 | stored variable( |
| 23 | arithmetic function name |
| 24 | arithmetic function argument |
| 25 | variable( in arithmetic |
| 26 | input listed variable |
| 27 | input listed variable( |
| 28 | output listed variable( |
| 29 | output listed variable( |
| 30 | name of routine being "CALLED". |
| 31 | single name unsubscripted CALL argument |
| 32 | name of SUBROUTINE |
| 33 | SUBROUTINE argument |
| 34 | name of FUNCTION |
| 35 | SAL exit |
| 36 | drop out reference in "IF" |
| 37 | Hollerith argument reference |
| 38 | variable in CONSTANT |
| 39 | mode of store item preceding exp. |
| 40 | TO item |
| 41 | single name subscripted CALL argument |
| 42 | FUNCTION argument |
| 43 | FROM item |
| 44 | OPEN subroutine, library subroutine |
| 45 | literal appearance of numeric quantities and numeric assignments |
| 46 | limit of DO |

| ID | DEFINITION |
|---|---|
| 47 | Function reference ("F" was dropped. |
| 48 | adjacent register flag (precedes first expression) |
| 49 | arithmetic and LSC statement operators |
| 60 | beginning and end of list item |

MODE WORD:

Information about an item is passed from phase to phase by a mode word which is part of the item. Phase I assigned mode words to literal appearances of numeric quantities and to operations. Phase III assigns mode words to items which represent variables and functions. Mode words are further up-dated in Phase IV, V, and VI; it is then used by Phase VII to generate code items. The mode word is a full 12 - digit LARC word broken down as follows. This summary includes information supplied to the mode word by Phases I to VI.

a, b, c, d, e, e, e, u, f, f, f, f

| | | |
|---|---|---|
| a (storage assignment) | 0 | Ordinary |
| | 1 | Common |
| | 2 | Equivalence |
| | 3 | Common and Equivalence |
| | 4 | Induction Variable |
| | 5 | Arithmetic Function |
| | 6 | Dummy Argument of an Arithmetic Function |
| | 7 | Built in Function |
| | 8 | Library Function |
| | 9 | Dummy Argument of FUNCTION or SUBROUTINE |
| b)arithmetic mode) | 0 | Not a Variable, Function, or Constant |
| | 1 | Integer |
| | 2 | FLOATING |
| | 3 | Double Precision |
| | 4 | Integer in integer form |
| | 5 | B-Box word |

| c(class) | | |
|---|---|---|
| | 0 | Fast Register |
| | 1 | Variable |
| | 2 | Constant |
| | 3 | Statement name |
| | 4 | Function or Subroutine |
| | 5 | Format |
| | 6 | Operation or Punctuation |
| | 7 | Built in Function |
| | 8 | Marth routine |

d(quantifier):

If c = 0

d = number of fast registers (1 or 2)

If c = 1

d = dimensionality (0, 1, 2, . . . 9)

after Phase IV, d = 1 if reference is subscripted, otherwise 0.

If c = 2

d = 0

If c = 3

d = 0 if the statement reference is direct

d = 1 if the statement reference is assigned

If c = 4

d = 0

If c = 5

d = 0 if the format reference is direct

d = 1 if the format reference is assigned

If c = 6

d = number of operands for the operator (see definition of eee for the value of d in each case)

eee:

If c = 1 and d ≠ 0

eee is the dimension table reference for the variable.

If a = 9

after Phase IV eee is the argument number of the variable

If c = 4

eee is the number of arguments of the function.

If c = 6

eee represents the operation (the list below gives values for eee and the corresponding value for d.

Specifier digits eee when c = 6. (Operation or Punctuation)

| eee = | | | d = |
|---|---|---|---|
| | 0 | , | 2 |
| | 1 | + | 2 |
| | 2 | - | 2 |
| | 3 | * | 2 |
| | 4 | / | 2 |
| | 5 | * * | 2 |
| | 6 | : | 2 |
| | 7 | fixed multiply | 2 |
| | 8 | fixed add | 2 |
| | 14 | RETURN | 0 |
| | 15 | PAUSE | 0 |
| | 16 | STOP | 0 |
| | 17 | END FILE | 1 |
| | 18 | DEF LARC DR | 4 |
| | 19 | POS. HEAD | 2 |
| | 20 | ADV. HEAD | 1 |
| | 21 | BACK HEAD | 1 |
| | 22 | R. LARC DRUM | 2 |
| | 23 | W. LARC DRUM | 2 |
| | 24 | R. DRUM | 2 |
| | 25 | Write DRUM | 2 |
| | 26 | R. Paper Tape | 2 |
| | 27 | W. Paper Tape | 2 |
| | 28 | TYPEWRITE | 2 |
| | 29 | SENSE LIGHT | 1 |
| | 30 | ASSIGN | 2 |
| | 31 | RIT | 2 |
| | 32 | WOT | 2 |
| | 33 | READ | 1 |
| | 34 | PRINT | 1 |
| | 35 | PUNCH | 1 |

| | | | |
|---|---|---|---|
| 36 | IF SENSE LIGHT | 3 | |
| 37 | GO TO<br>(and ASSIGNED GO TO) | 1 | |
| 38 | IF SS | 2 | |
| 39 | IF > 0 | 1 | Followed by GO TO . . . |
| 40 | IF = 0 | 1 | (generated for IF POS. etc.) |
| 41 | IF < 0 | 1 | |
| 42 | IF > | 1 | |
| 43 | IF = | 1 | |
| 45 | COMPUTED<br>GO TO | 0 | K, the number of GO TO operands, is put into the M-portion of 3rd word of the operator item. |
| 49 | WT | 1 | |
| 50 | REW | 1 | |
| 51 | REW INT | 1 | |
| 52 | BACKSP | 1 | |
| 54 | IF ( ) | 3 | (FORTRAN IF statement) |
| 55 | IF ≥ 0 | 1 | |
| 56 | IF ≠ 0 | 1 | |
| 57 | IF < | 1 | Followed by a GO TO |
| 58 | IF > | 1 | (generated for IF NOT, etc.) |
| 59 | IF ≠ | 1 | |
| 60 | IF < | 1 | |
| 61 | RT | 1 | |

ufff:

If c = 1 and a ≠ 4

u = 0

fff = dictionary reference to name or dictionary reference to name and and constant (u = 1 if a = 9 and constant is negative or sequence of generated statement name.

If c = 1 and a = 4

uffff = new name of induction variable item

If c = 2 and b = 3

u = 0

ffff = dictionary reference to low order part of the constant

If c = 6

Phase VI may further use uffff for the mode words of operations as follows

u = 0: expression is not common

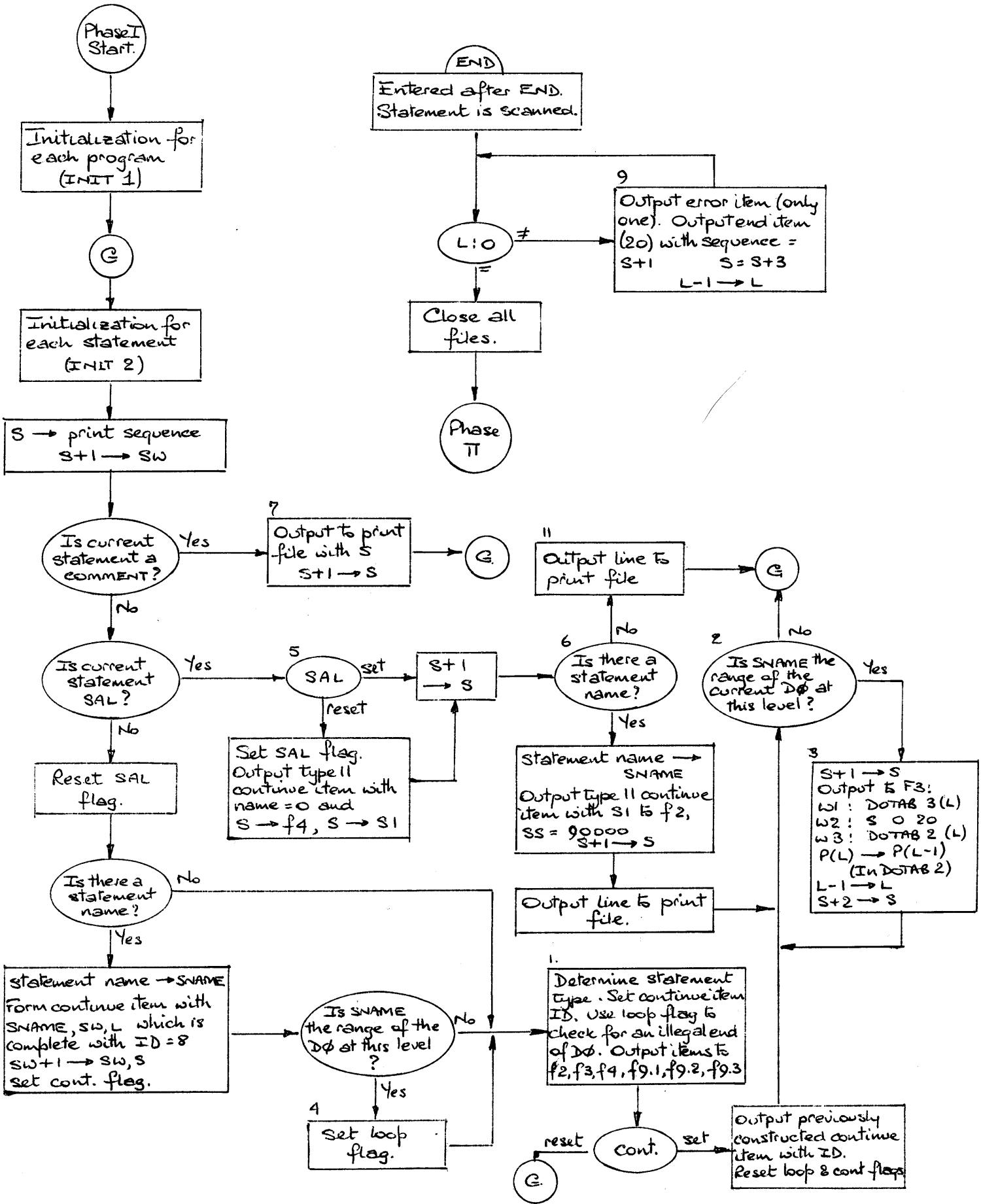
= 1: expresion is common and is not in the first major sequence of its occurrence.

= 2: expression is common and is in the first major sequence of its occurrence.

If u = 1, 2, then ffff = the integer which represents which subexpression.

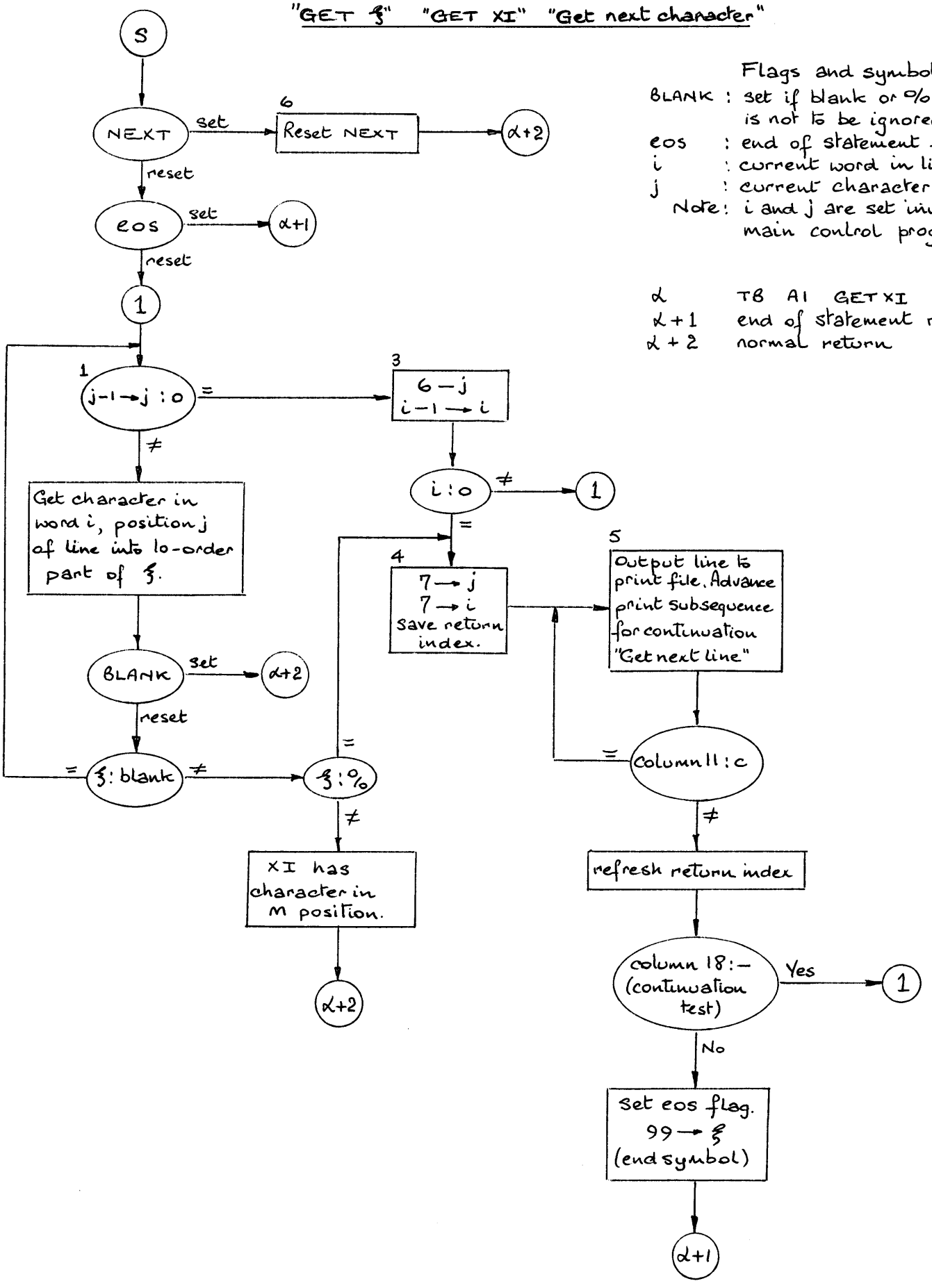
Note: After Phase VII, the mode word described above no longer exists. The code items to File 8 incorporate the information in the mode word of the File 7 items. The Phase VIII report contains this information.

General Phase I flow with details on continue items and sequencing.



# LARC SCIENTIFIC COMPILER Phase I

## "GET ξ" "GET XI" "Get next character"

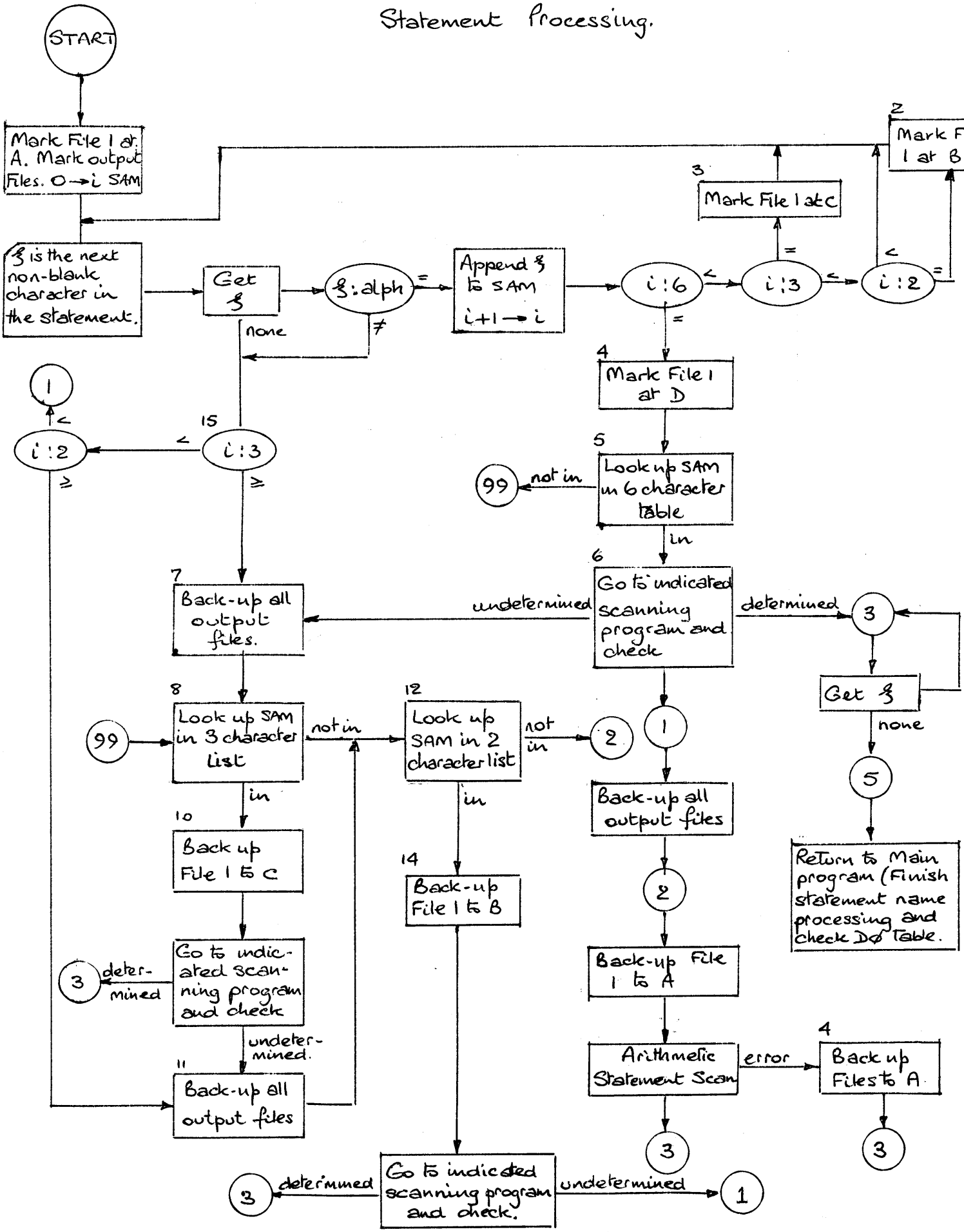



Flags and symbols.

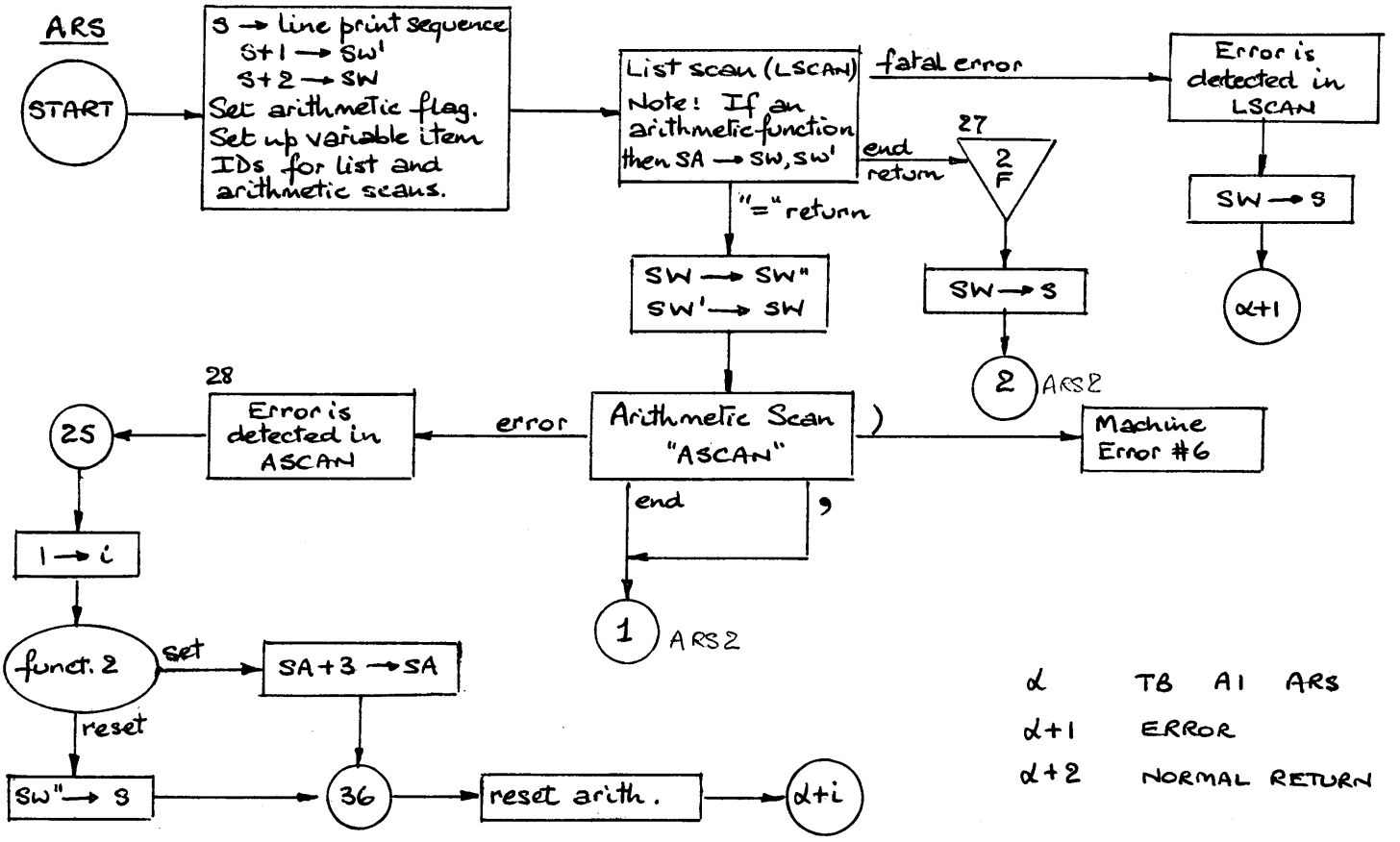
BLANK : set if blank or % character is not to be ignored.
eos : end of statement flag.
i : current word in line
j : current character position
Note: i and j are set initially by main control program.

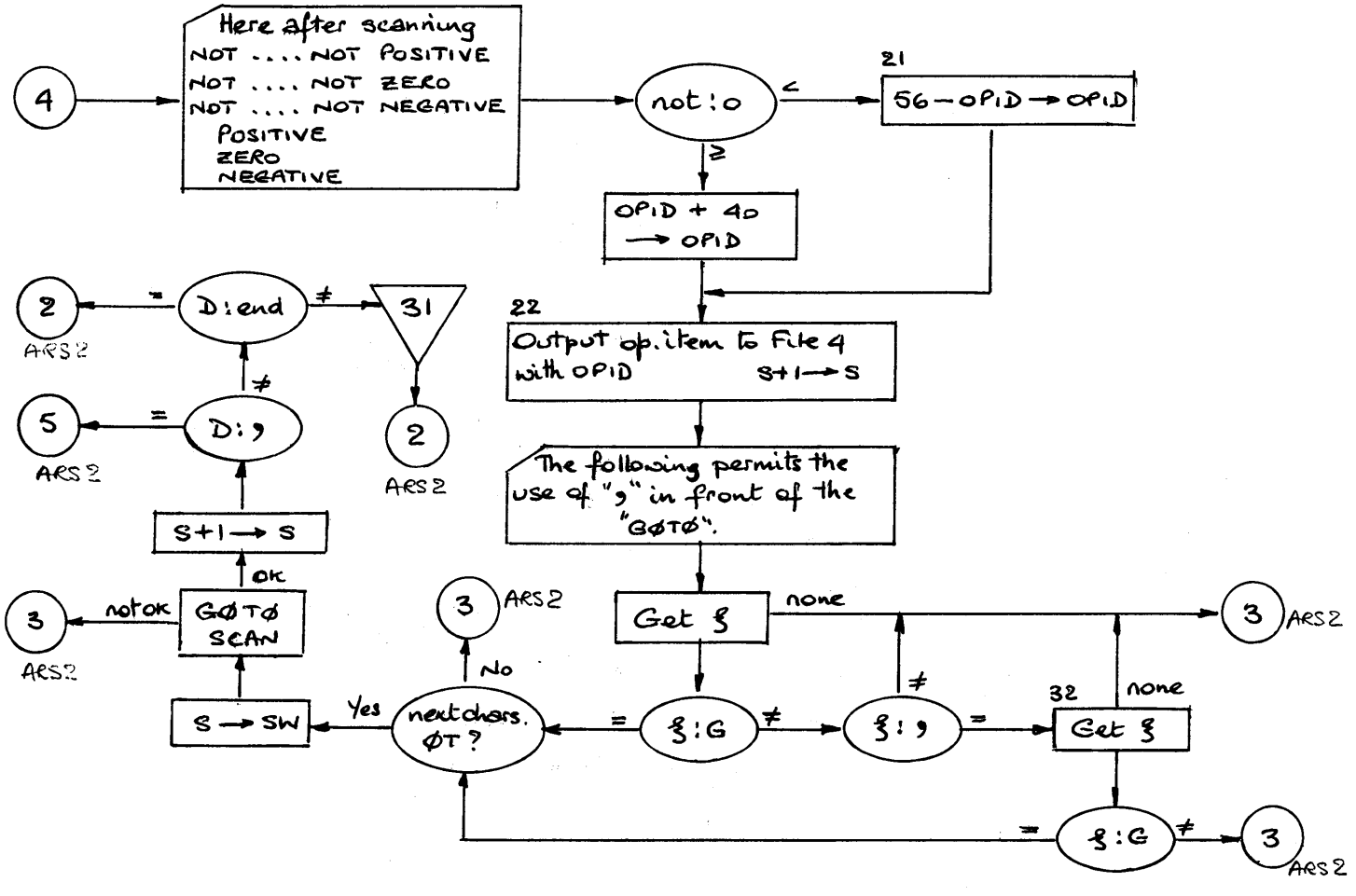
| | |
|---|---|
| α | TB A1 GETXI |
| α+1 | end of statement return (ξ=99) |
| α+2 | normal return |

# Statement Processing.



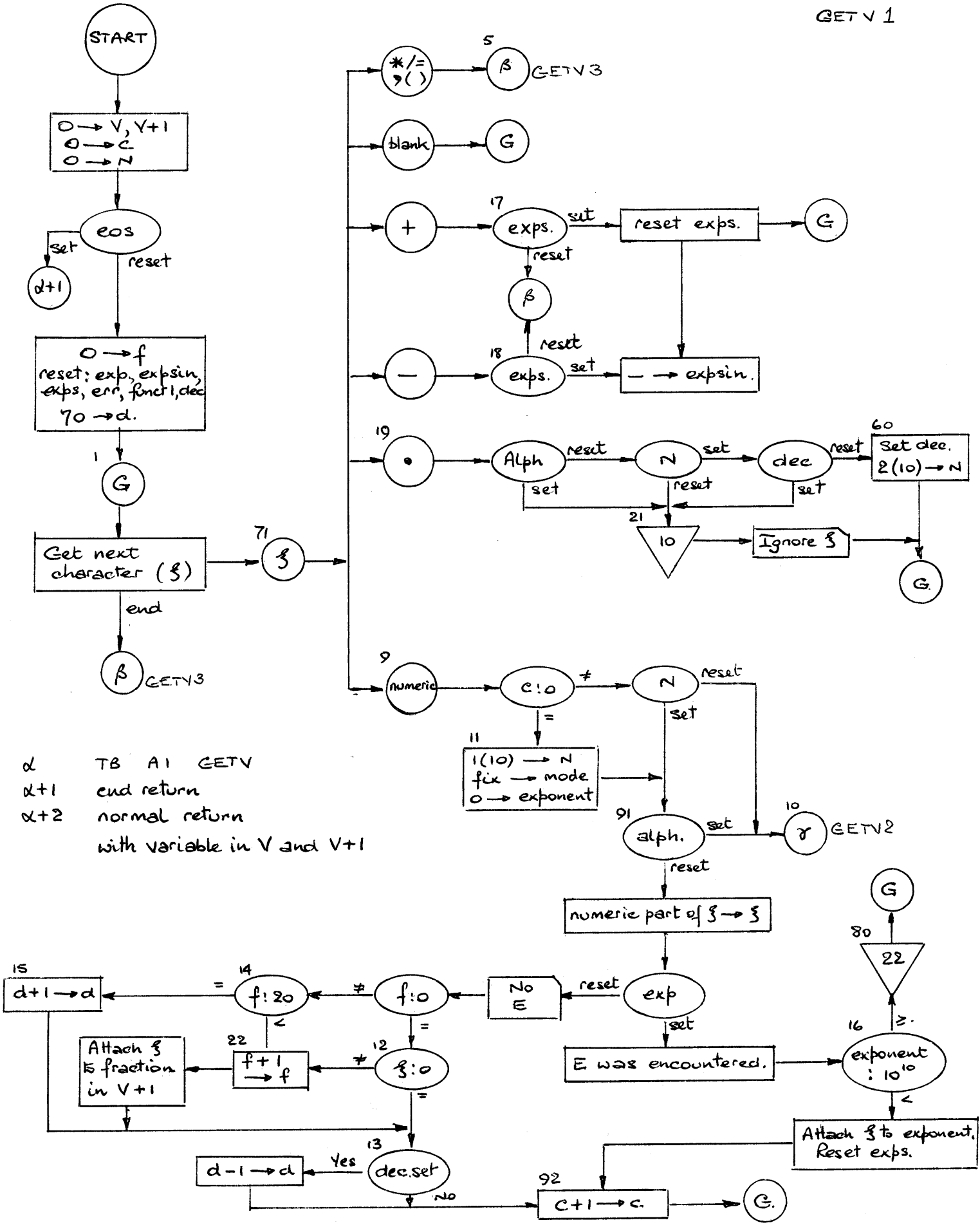
# LARC SCIENTIFIC COMPILER Phase I
## ARITHMETIC STATEMENT SCAN.



# "GET V"

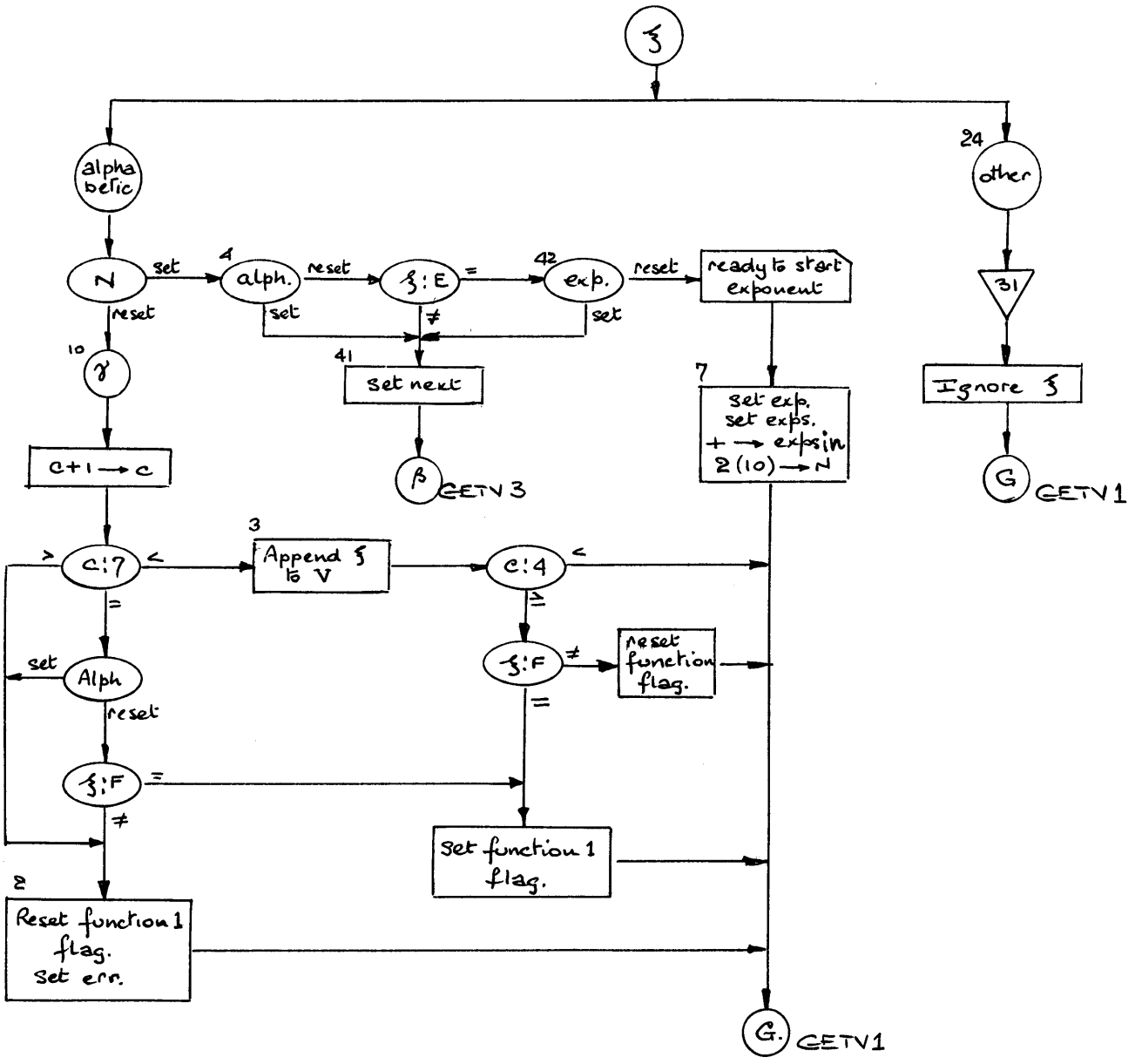
GETV 1




α TB A1 GETV
α+1 end return
α+2 normal return
with variable in V and V+1

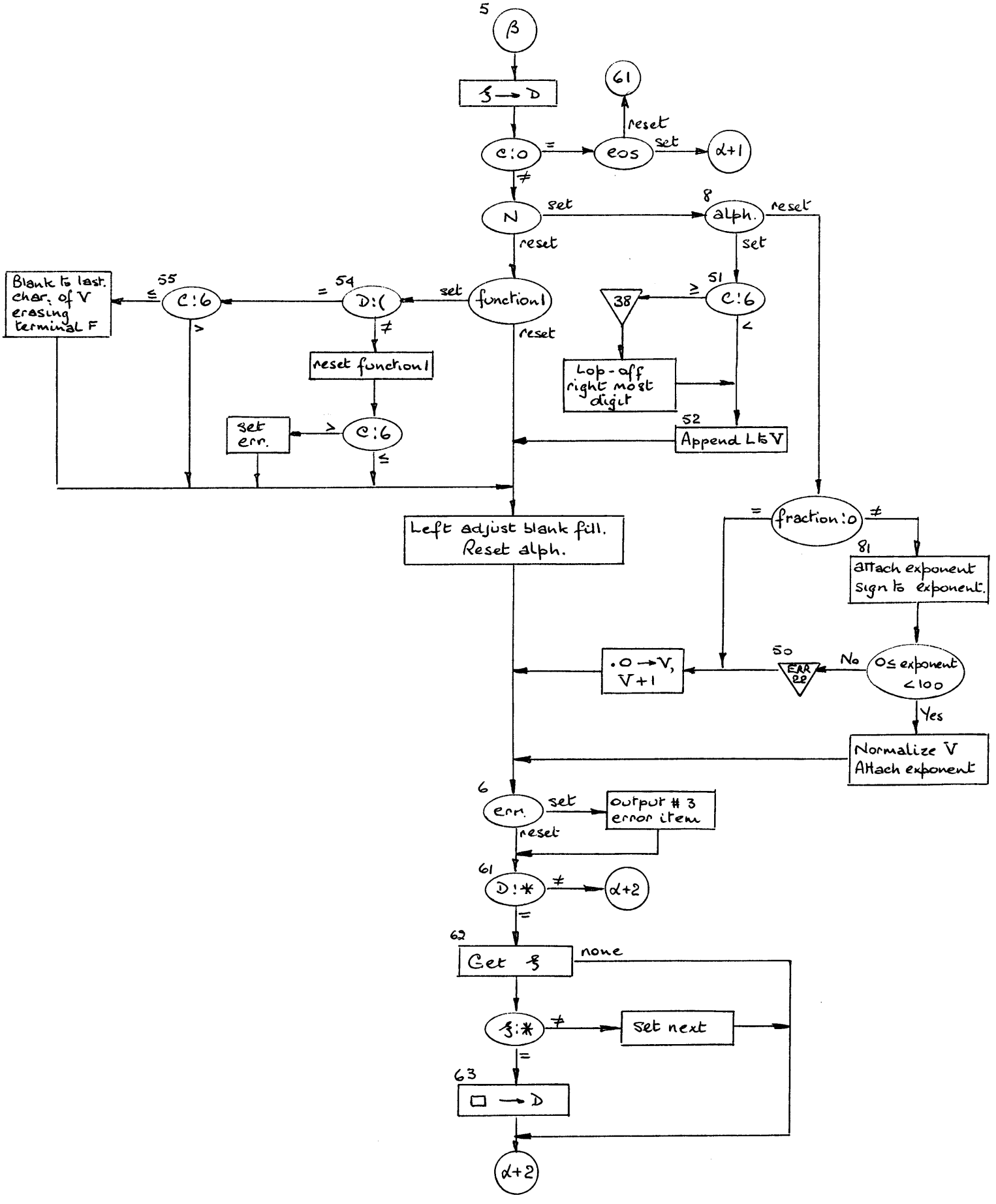
"GET V" (2)

GETV2

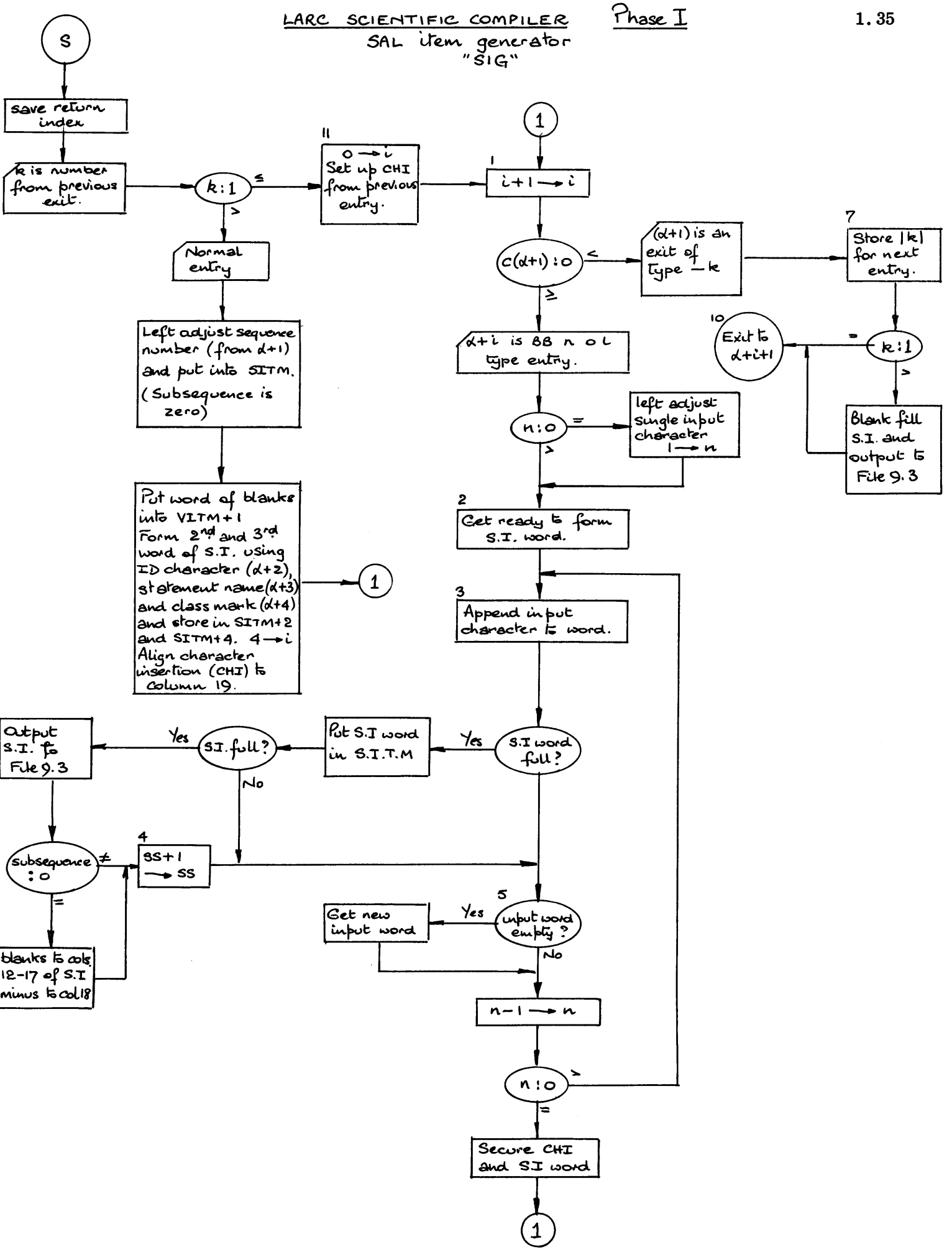

ξ
alpha betic
24
other
N
set
4
alph.
reset
ξ:E
=
42
exp.
reset
ready to start exponent
reset
set
≠
set
10
γ
41
set next
7
31
set exp.
set exps.
+ → expsin
2(10) → N
Ignore ξ
c+1 → c
β GETV3
G GETV1
3
c:7
Append ξ to V
c:4
>
<
=
≥
set
Alph
reset
ξ:F
≠
reset function flag.
=
=
set function 1 flag.
2
Reset function 1 flag.
Set err.
G. GETV1

"GET V" (3)

GETV3.

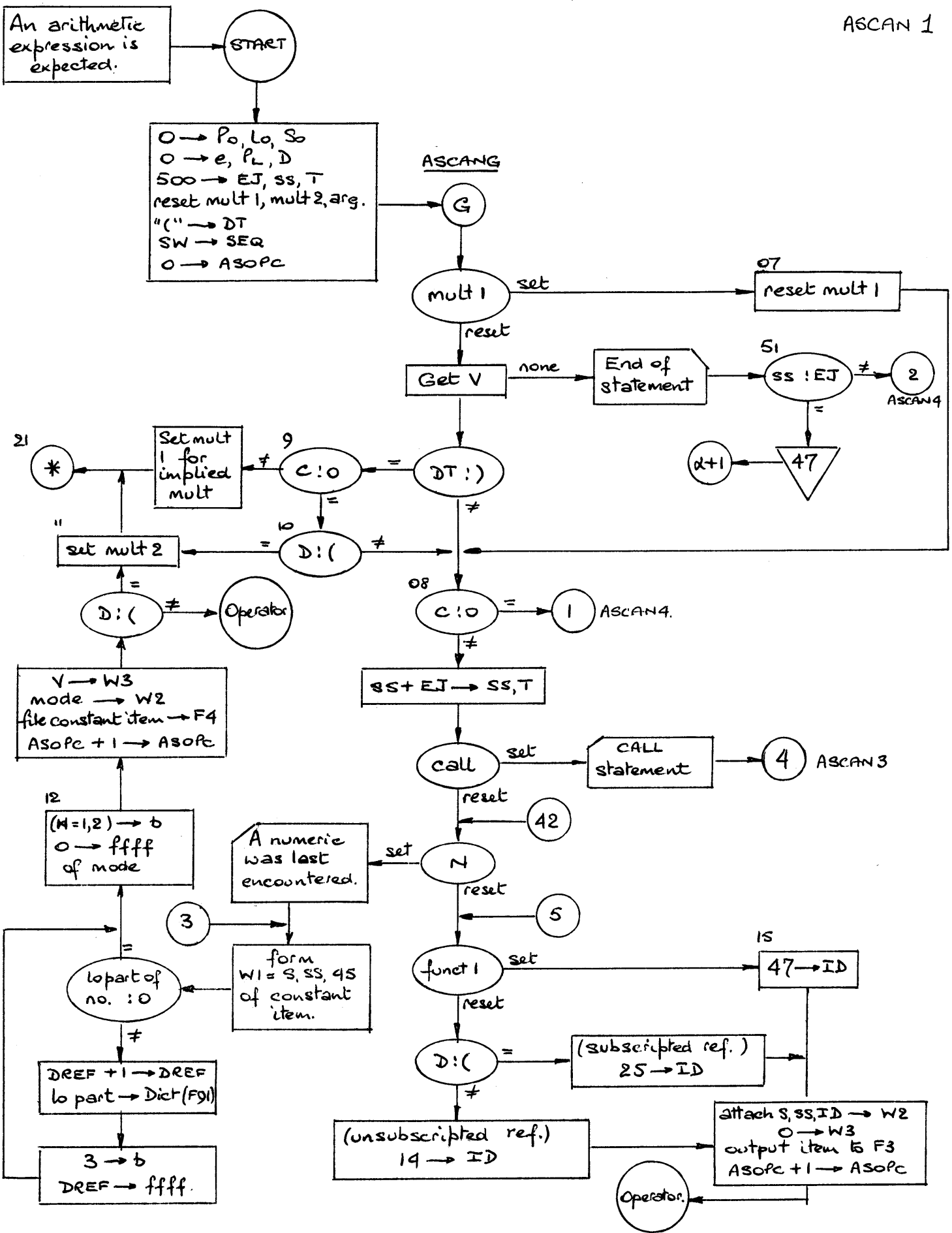

5
β
§ → D
61
reset
C:0
=
eos
set
α+1
≠
N
set
8
alph.
reset
reset
set
Blank to last char. of V erasing terminal F
55
≤
C:6
>
=
54
D:(
set
function
≠
reset function
set err.
≥
C:6
≤
reset
51
C:6
≥
38
Lop-off right most digit
<
52
Append L to V
Left adjust blank fill. Reset alph.
=
fraction:0
≠
81
attach exponent sign to exponent.
.0 → V, V+1
50
ERR 22
No
0 ≤ exponent < 100
Yes
Normalize V Attach exponent
6
err.
set
output # 3 error item
reset
61
D:*
≠
α+2
=
62
Get §
none
§:*
≠
set next
=
63
□ → D
α+2

SAL item generator
"SIG"



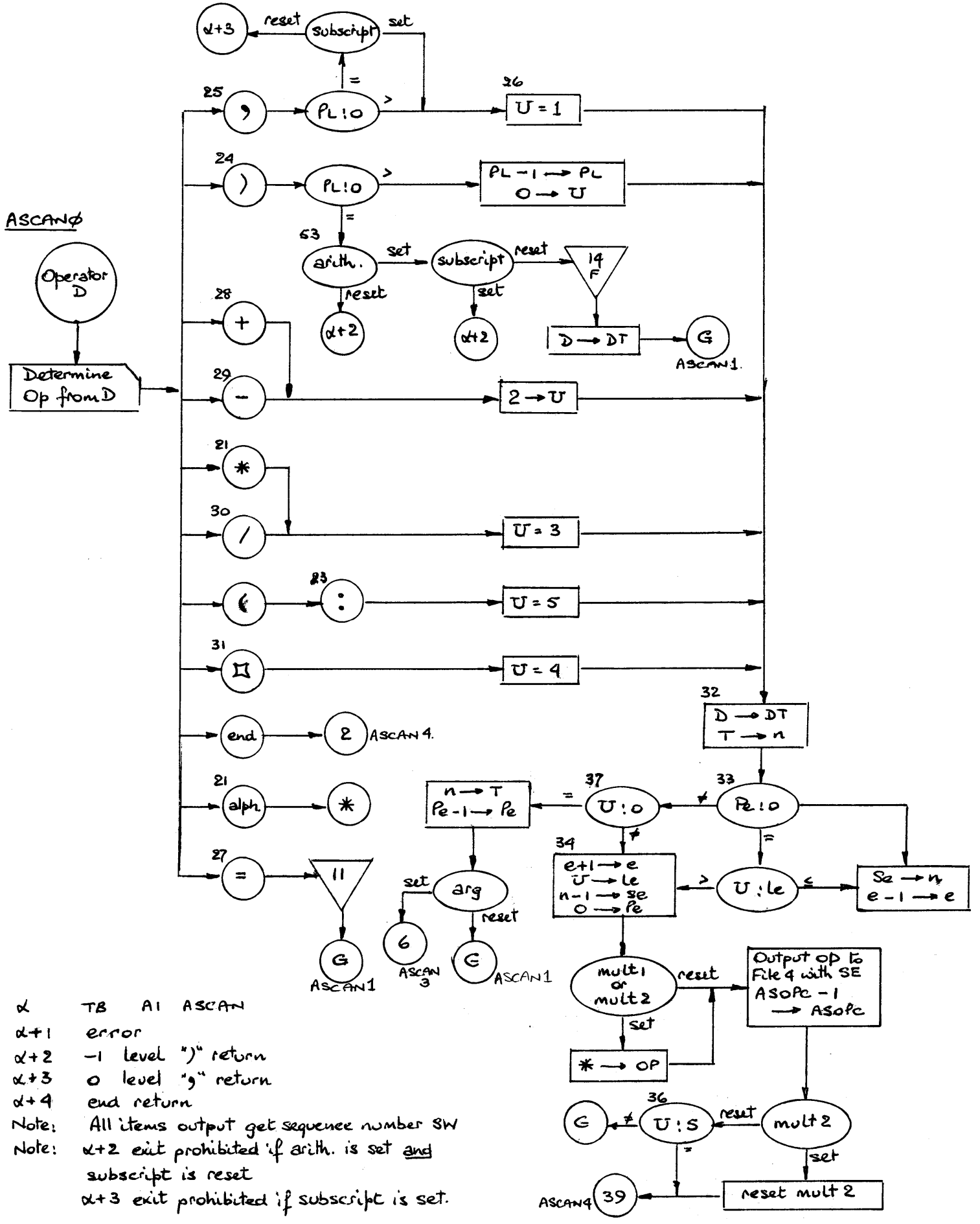
# ARITHMETIC SCAN

ASCAN 1



# ARITHMETIC SCAN.

ASCAN2.



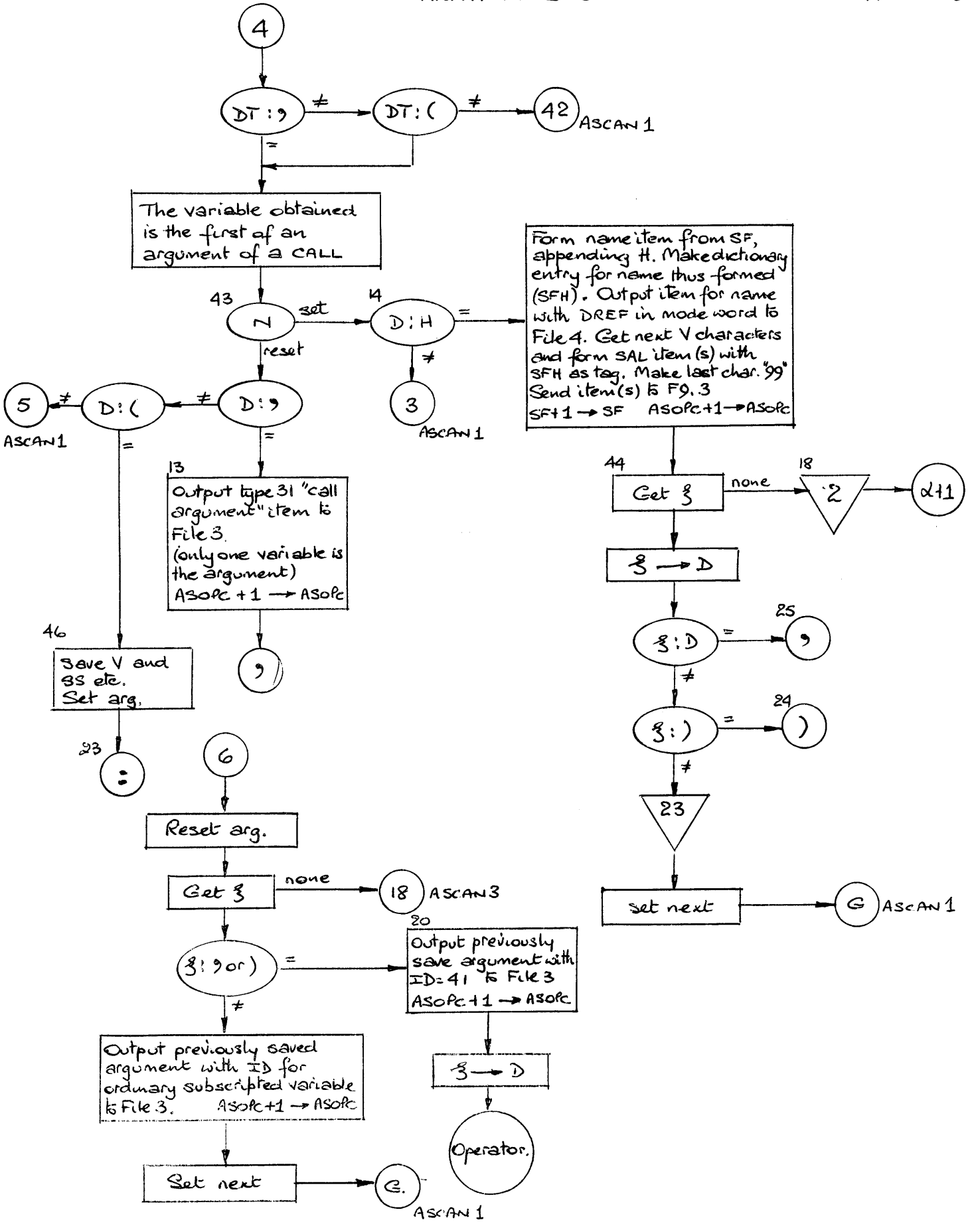

| | | | |
|---|---|---|---|
| α | TB | A1 | ASCAN |
| α+1 | error | | |
| α+2 | −1 | level | ")" return |
| α+3 | 0 | level | "," return |
| α+4 | end | return | |

Note: All items output get sequence number SW

Note: α+2 exit prohibited if arith. is set and subscript is reset

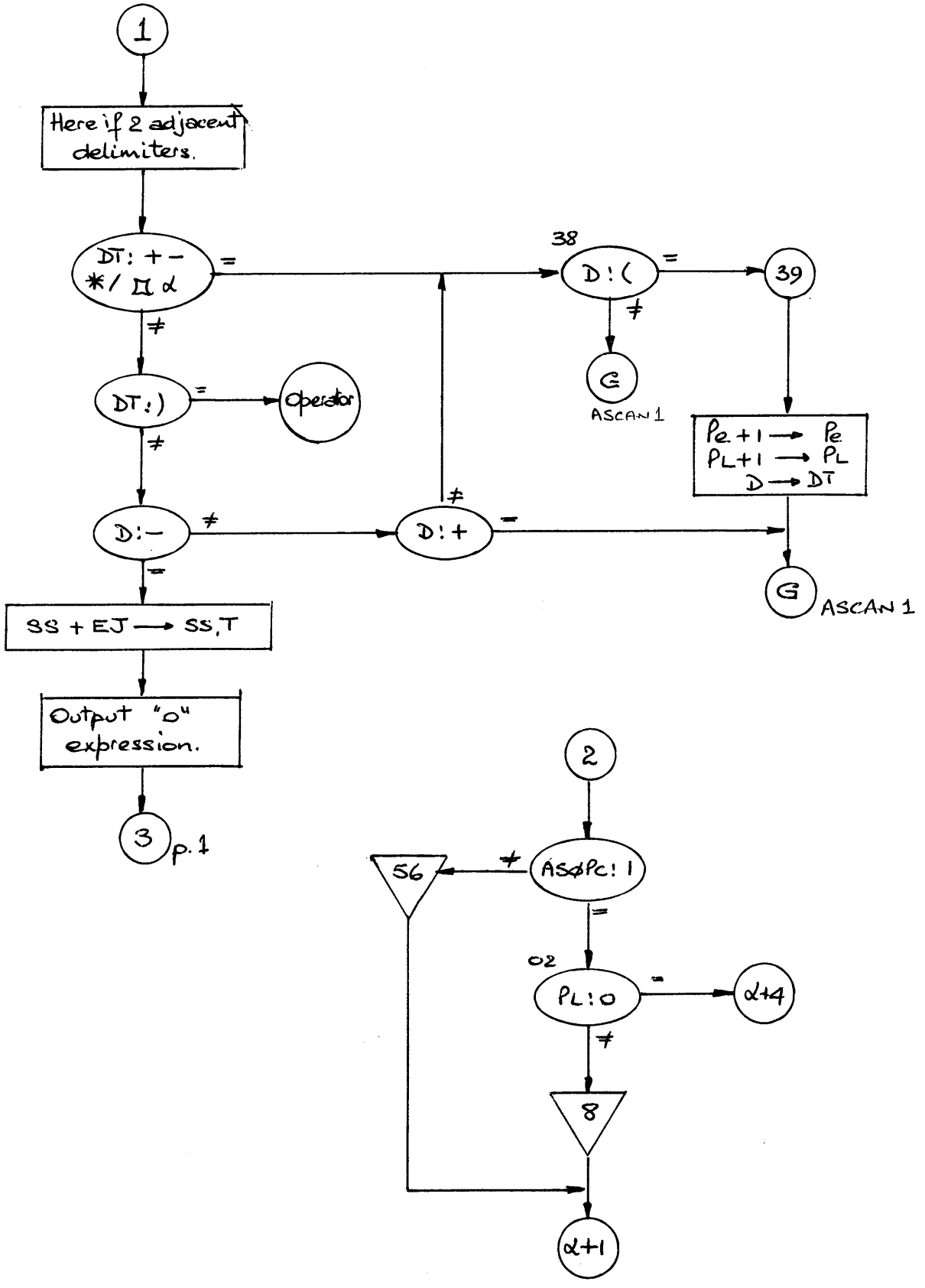
α+3 exit prohibited if subscript is set.

ARITHMETIC SCAN ASCAN 3



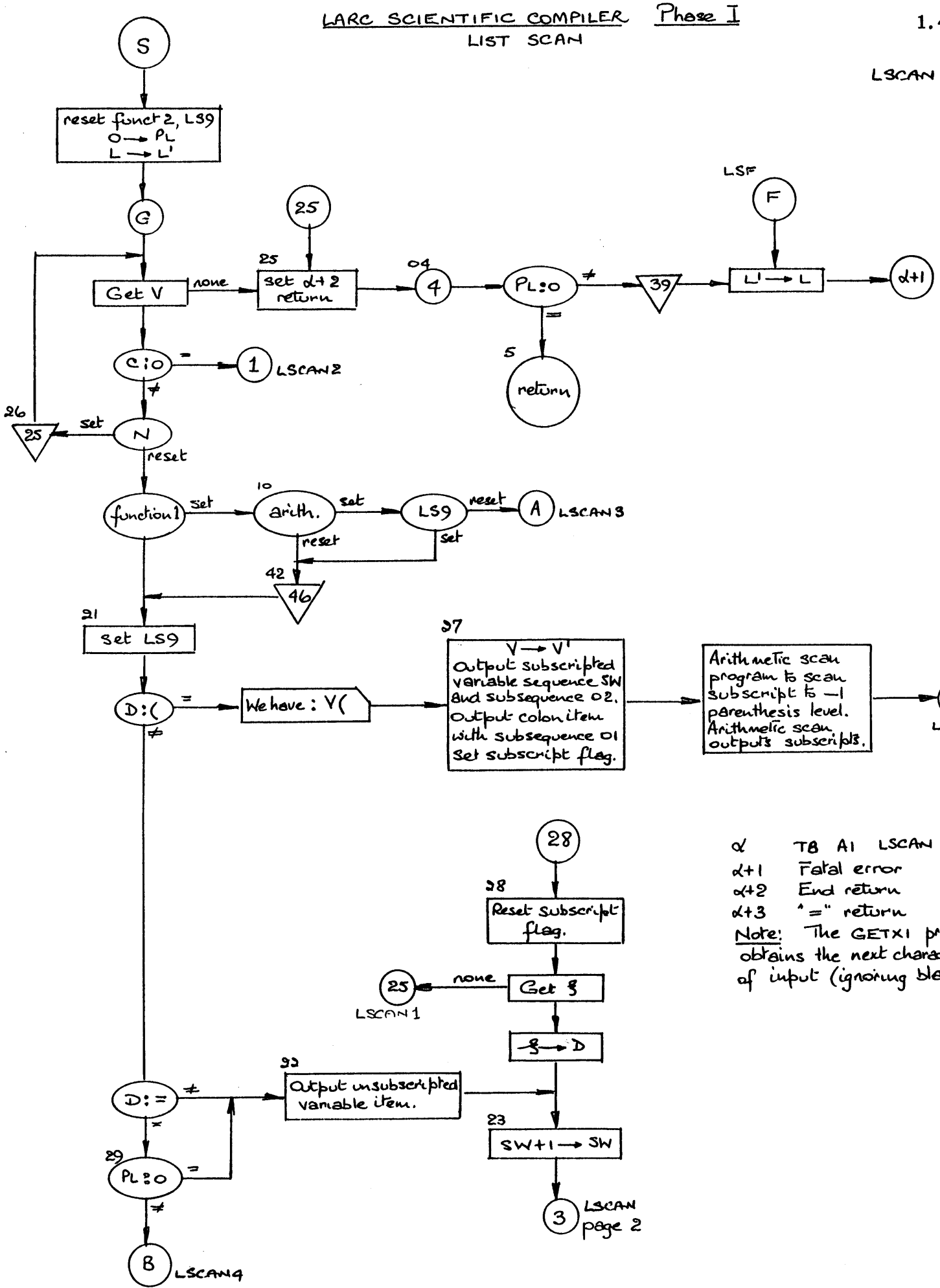
ARITHMETIC SCAN

ASCAN4.



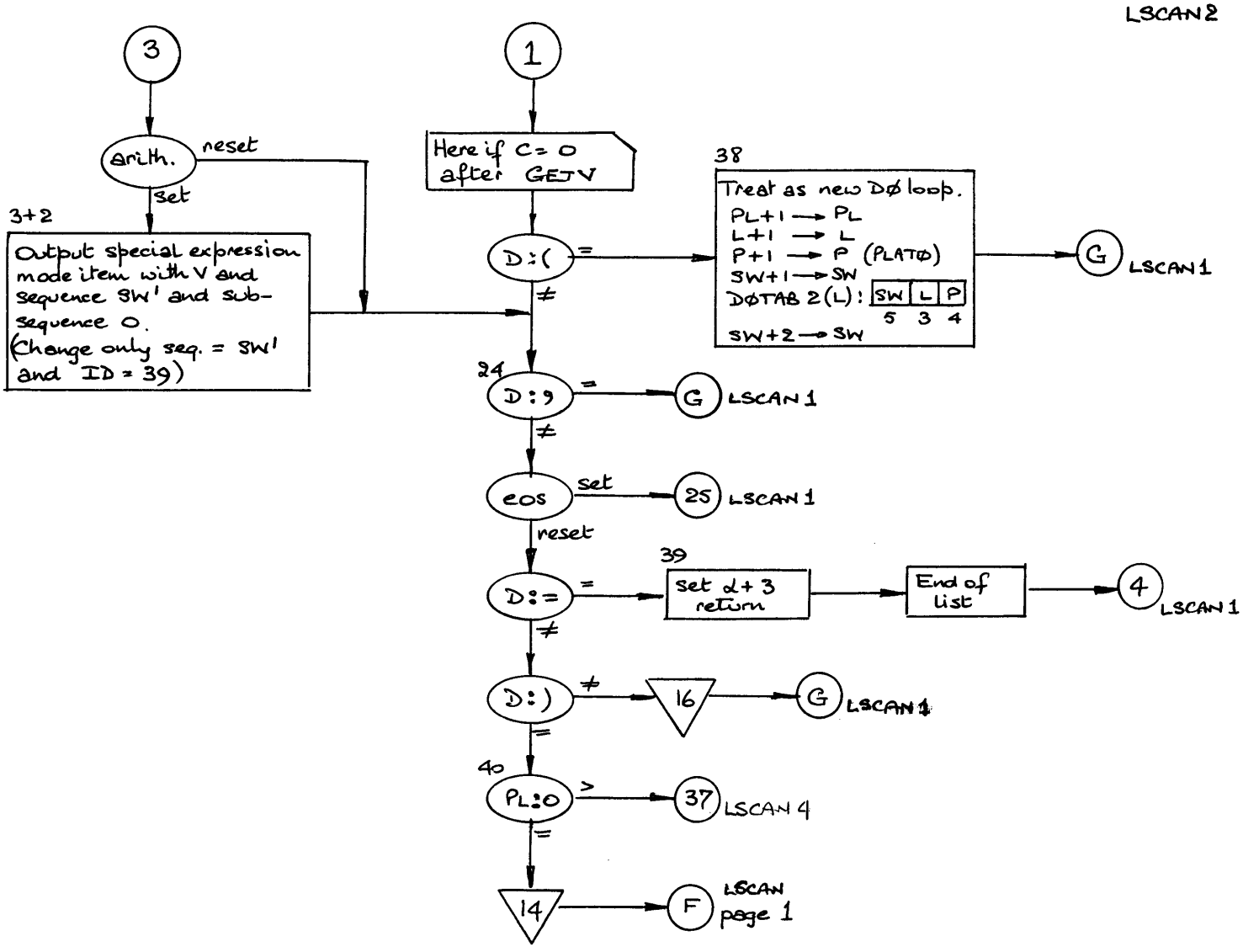
# LARC SCIENTIFIC COMPILER Phase I
## LIST SCAN

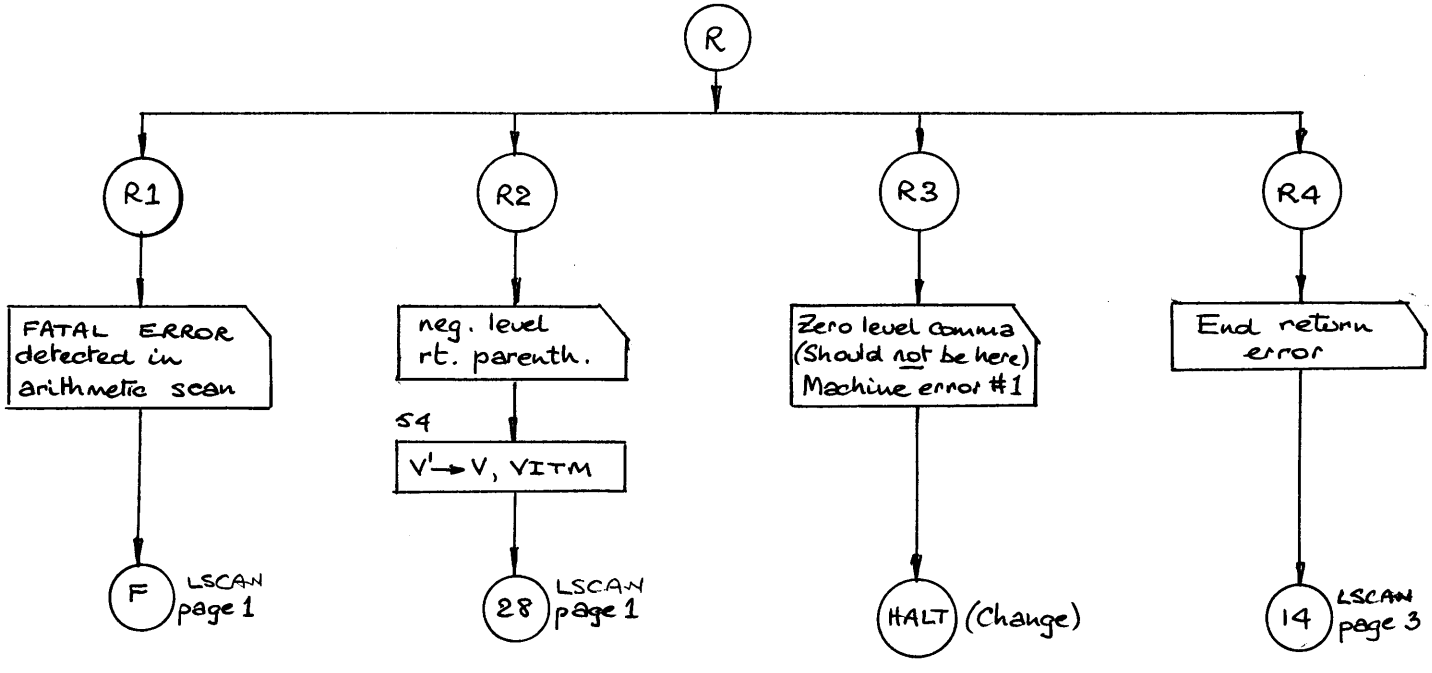
LSCAN 1

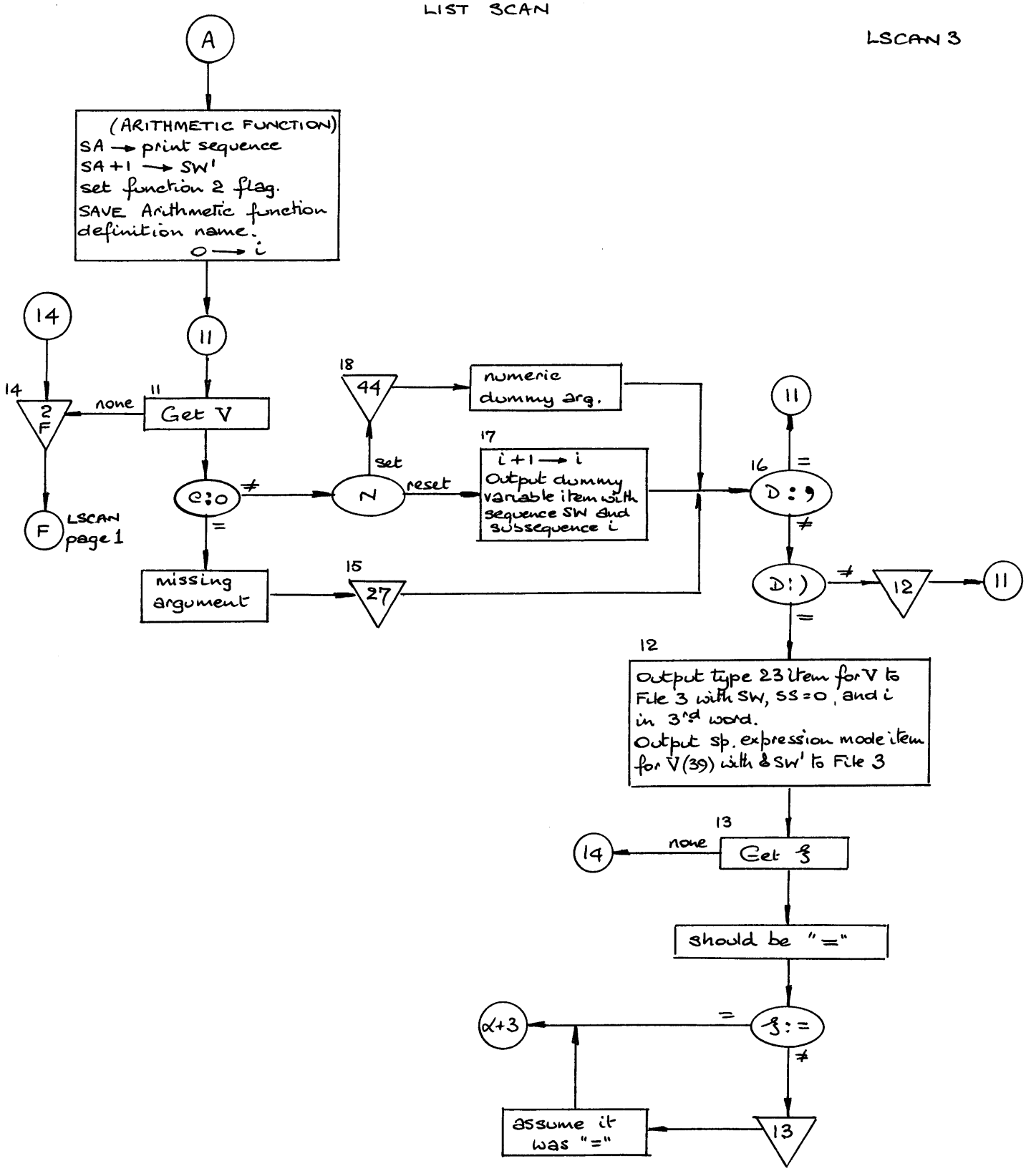



α TB A1 LSCAN
α+1 Fatal error
α+2 End return
α+3 "=" return

Note: The GETX1 program obtains the next character of input (ignoring blanks)

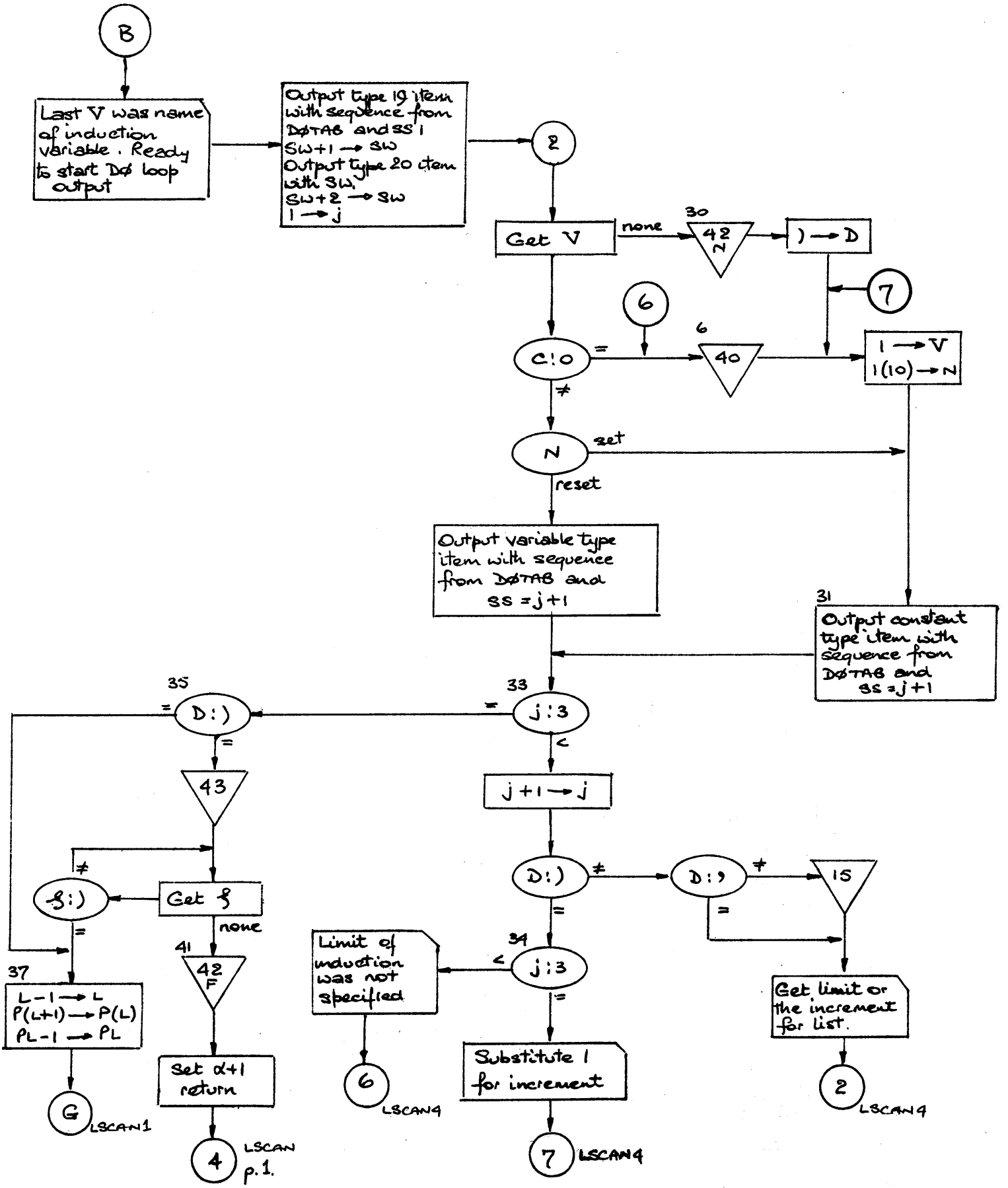

3
Arith.
reset
set
3+2
Output special expression mode item with V and sequence SW' and sub-sequence 0. (Change only seq. = SW' and ID = 39)
1
Here if C = 0 after GETV
D:(
=
≠
38
Treat as new DØ loop.
PL+1 → PL
L+1 → L
P+1 → P (PLATØ)
SW+1 → SW
DØTAB 2(L): SW | L | P
5 3 4
SW+2 → SW
G LSCAN 1
24
D:9
=
G LSCAN 1
≠
eos
set
25 LSCAN 1
reset
D:=
=
39
set 2+3 return
End of list
4 LSCAN 1
≠
D:)
≠
16
G LSCAN 1
=
40
PL:0
>
37 LSCAN 4
=
14
F LSCAN page 1


R
R1
FATAL ERROR detected in arithmetic scan
F LSCAN page 1
R2
neg. level rt. parenth.
54
V'→V, VITM
28 LSCAN page 1
R3
Zero level comma (Should not be here) Machine error #1
HALT (Change)
R4
End return error
14 LSCAN page 3


A
(ARITHMETIC FUNCTION)
SA → print sequence
SA+1 → SW'
set function 2 flag.
SAVE Arithmetic function definition name.
0 → i
14
11
14
2 F
none
11
Get V
18
44
numeric dummy arg.
17
i+1 → i
Output dummy variable item with sequence SW and subsequence i
set
reset
N
@:0
≠
=
F
LSCAN page 1
missing argument
15
27
11
=
16
D:9
≠
D!)
≠
12
11
=
12
Output type 23 item for V to File 3 with SW, SS=0, and i in 3rd word.
Output sp. expression mode item for V(39) with & SW' to File 3
13
14
none
Get $
should be "="
=
α+3
$:=
≠
assume it was "="
13

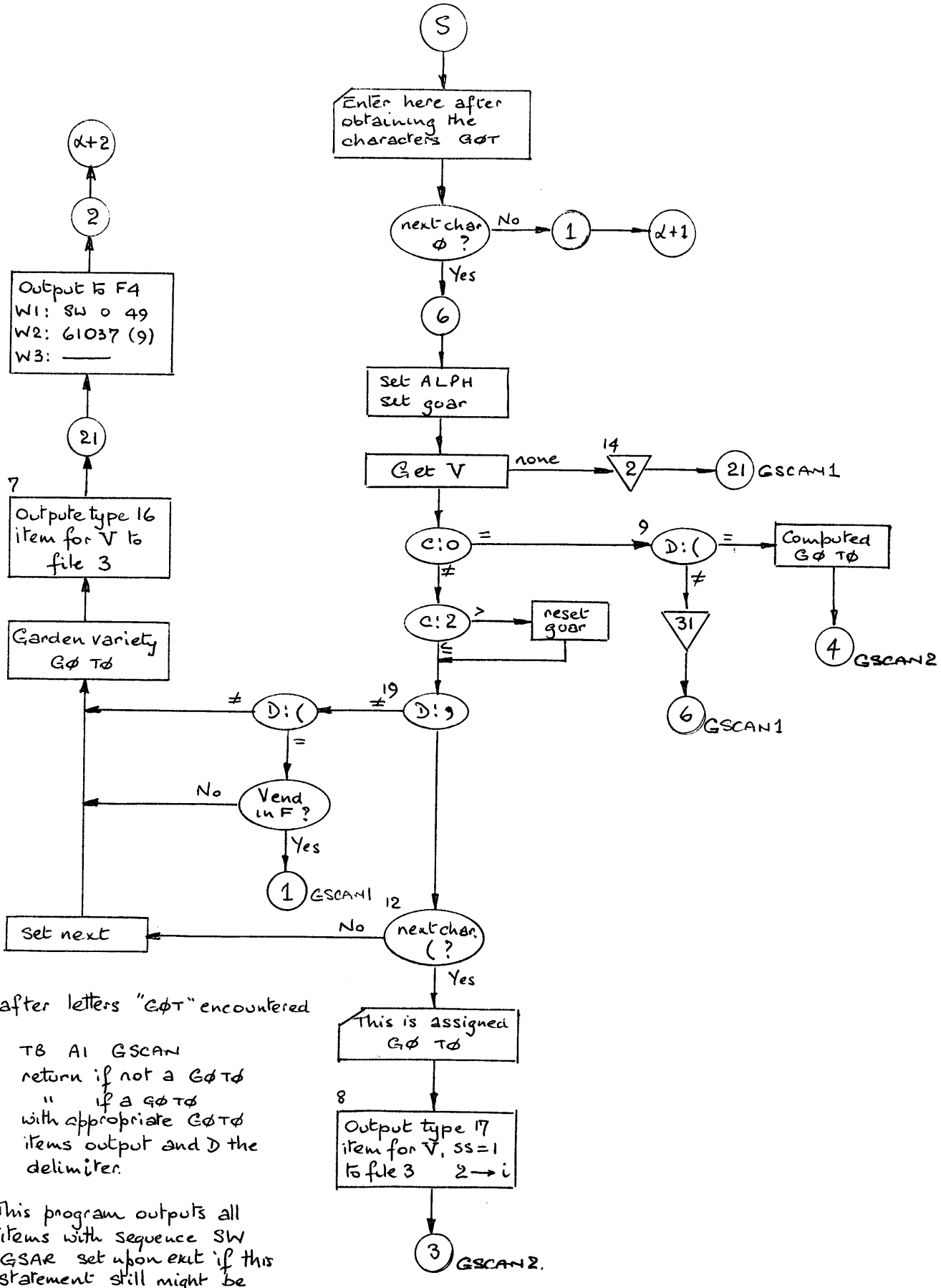
# LIST SCAN

LSCAN 4.



GØ TØ SCAN (GSCAN) (1)

GSCAN1




Enter after letters "GØT" encountered

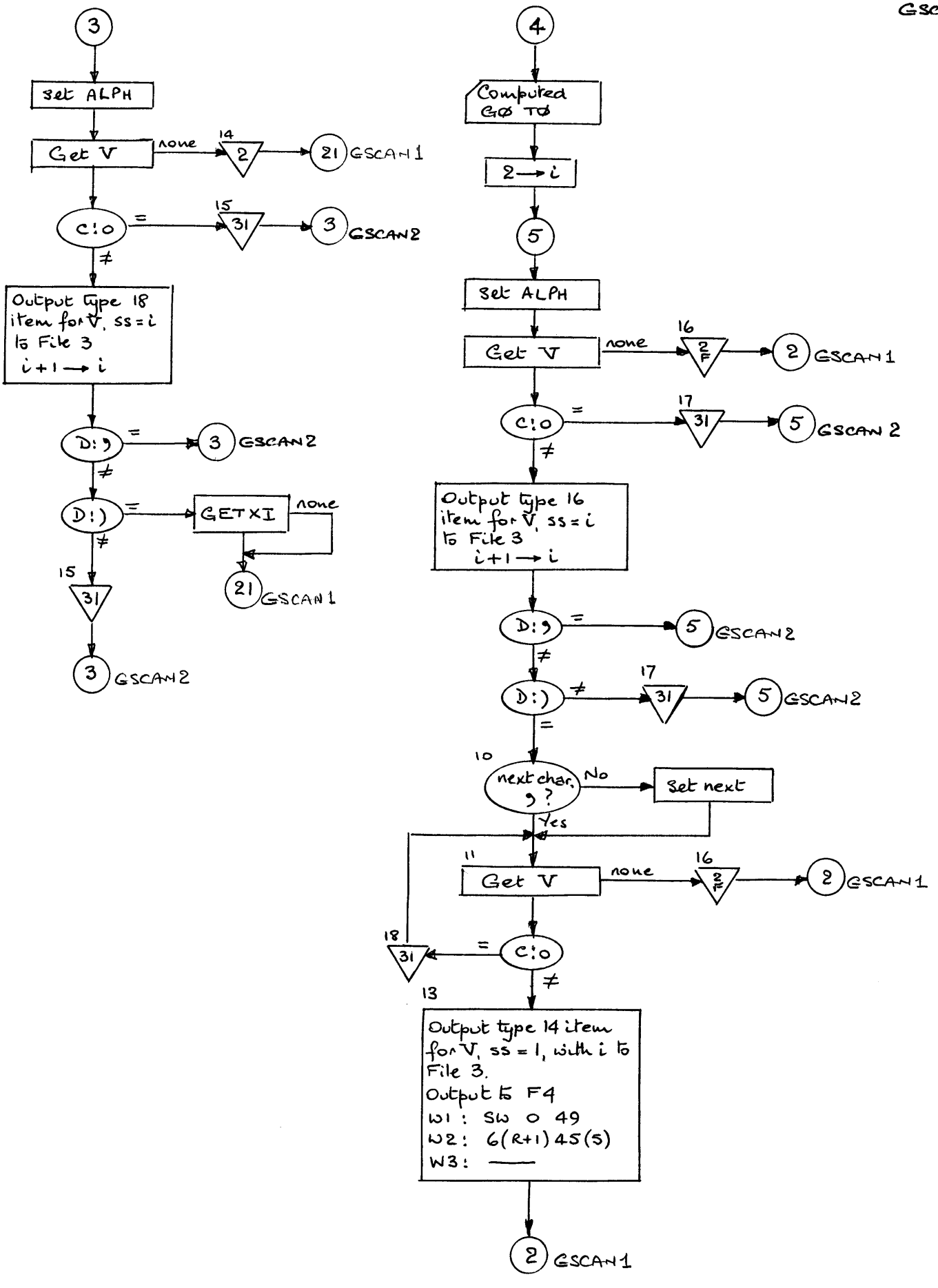
| | |
|---|---|
| α | TB A1 GSCAN |
| α+1 | return if not a GØ TØ |
| α+2 | " if a GØ TØ with appropriate GØ TØ items output and D the delimiter. |

Note: This program outputs all items with sequence SW

Note: GSAR set upon exit if this statement still might be arithmetic.

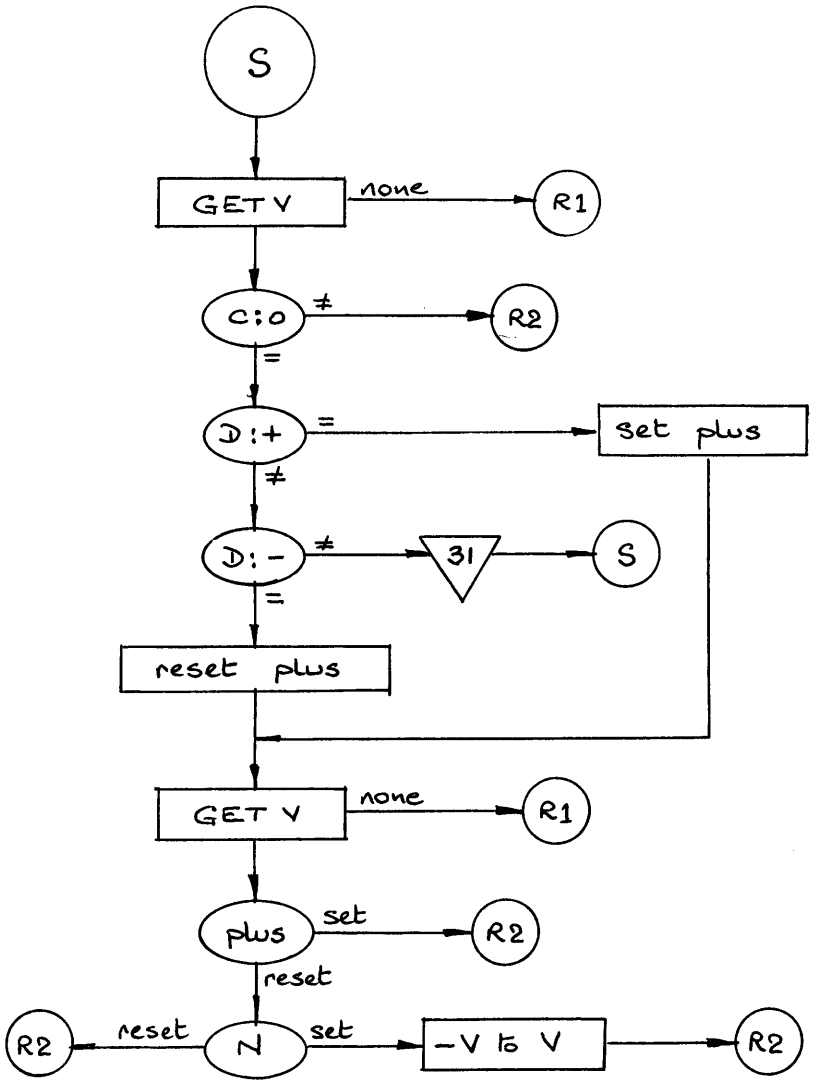
# GØ TØ SCAN (GSCAN) (2)





Subroutine GETSV

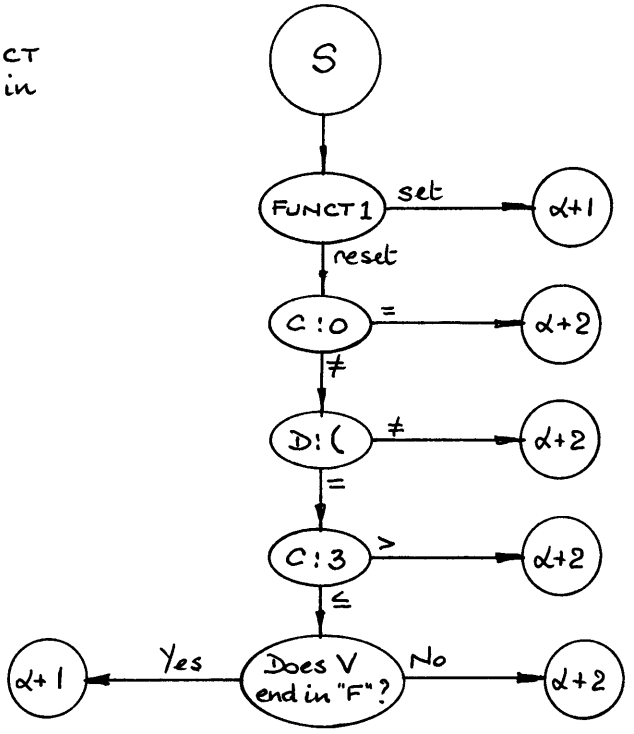
| | |
|---|---|
| α | TB A1 GETSV |
| α+1 | EØS |
| α+2 | return with signed number in V, V+1 |




Subroutine VFUNCT

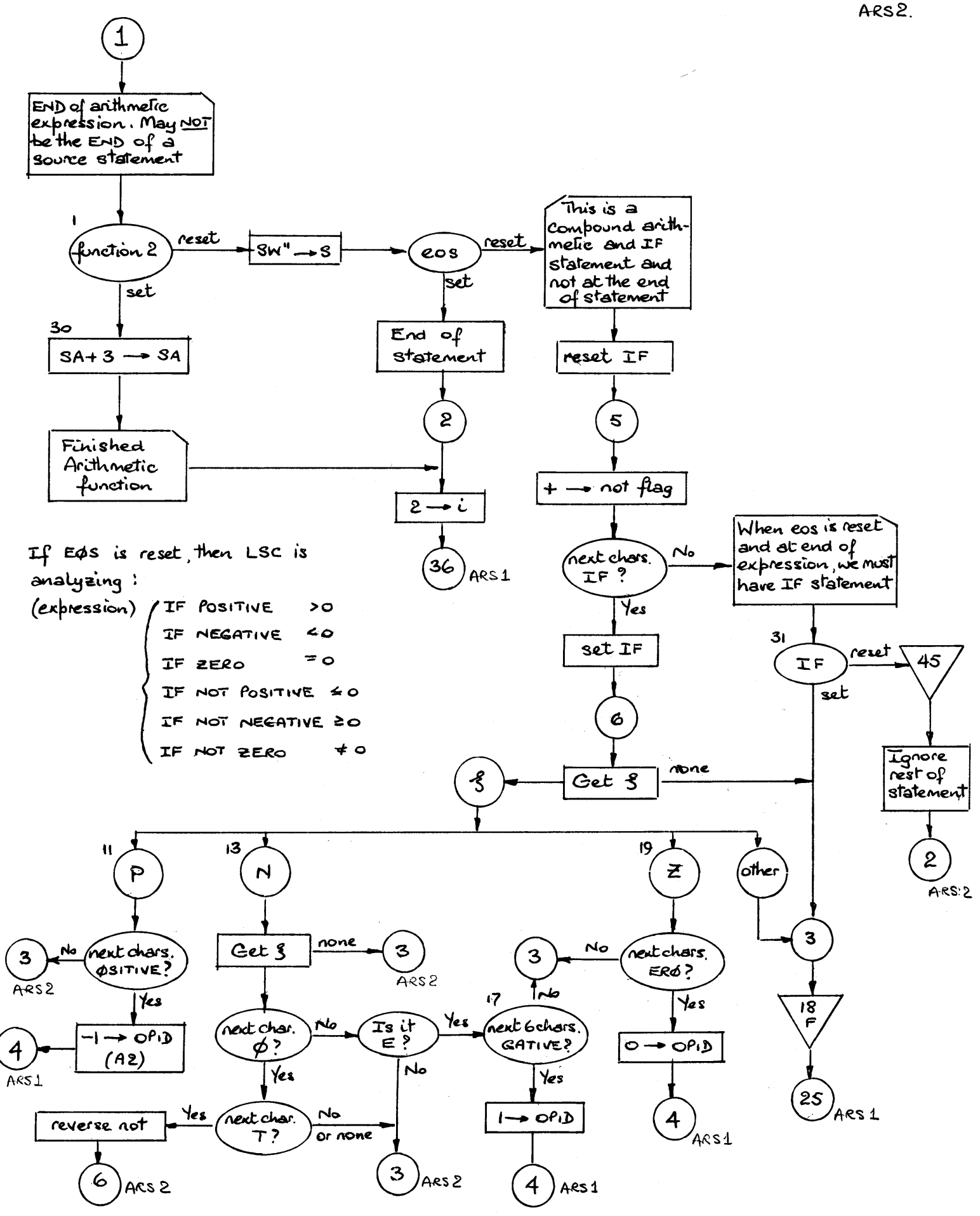
Use: α TB A1 VFUNCT

| | |
|---|---|
| α+1 | here if V ends in F and D=( |
| α+2 | here if not |



# ARITHMETIC STATEMENT SCAN

ARS2.

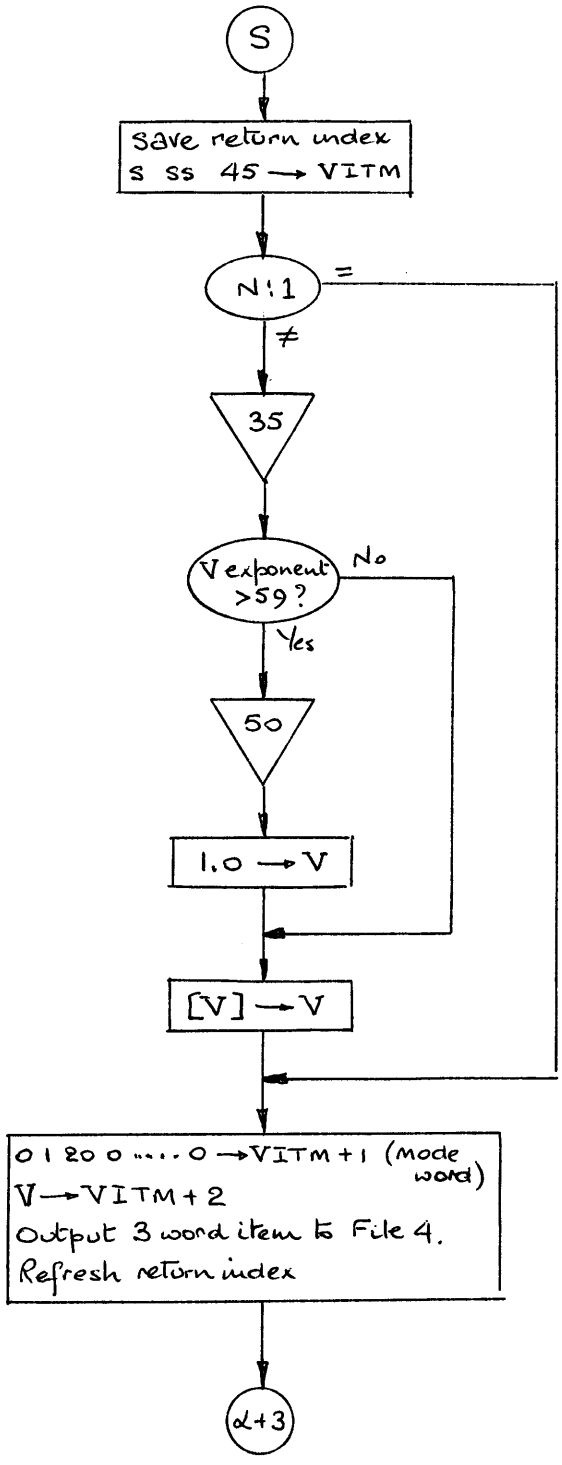



If EØS is reset, then LSC is analyzing :

(expression)

- IF POSITIVE >0
- IF NEGATIVE <0
- IF ZERO =0
- IF NOT POSITIVE ≤0
- IF NOT NEGATIVE ≥0
- IF NOT ZERO ≠0

CONITM.

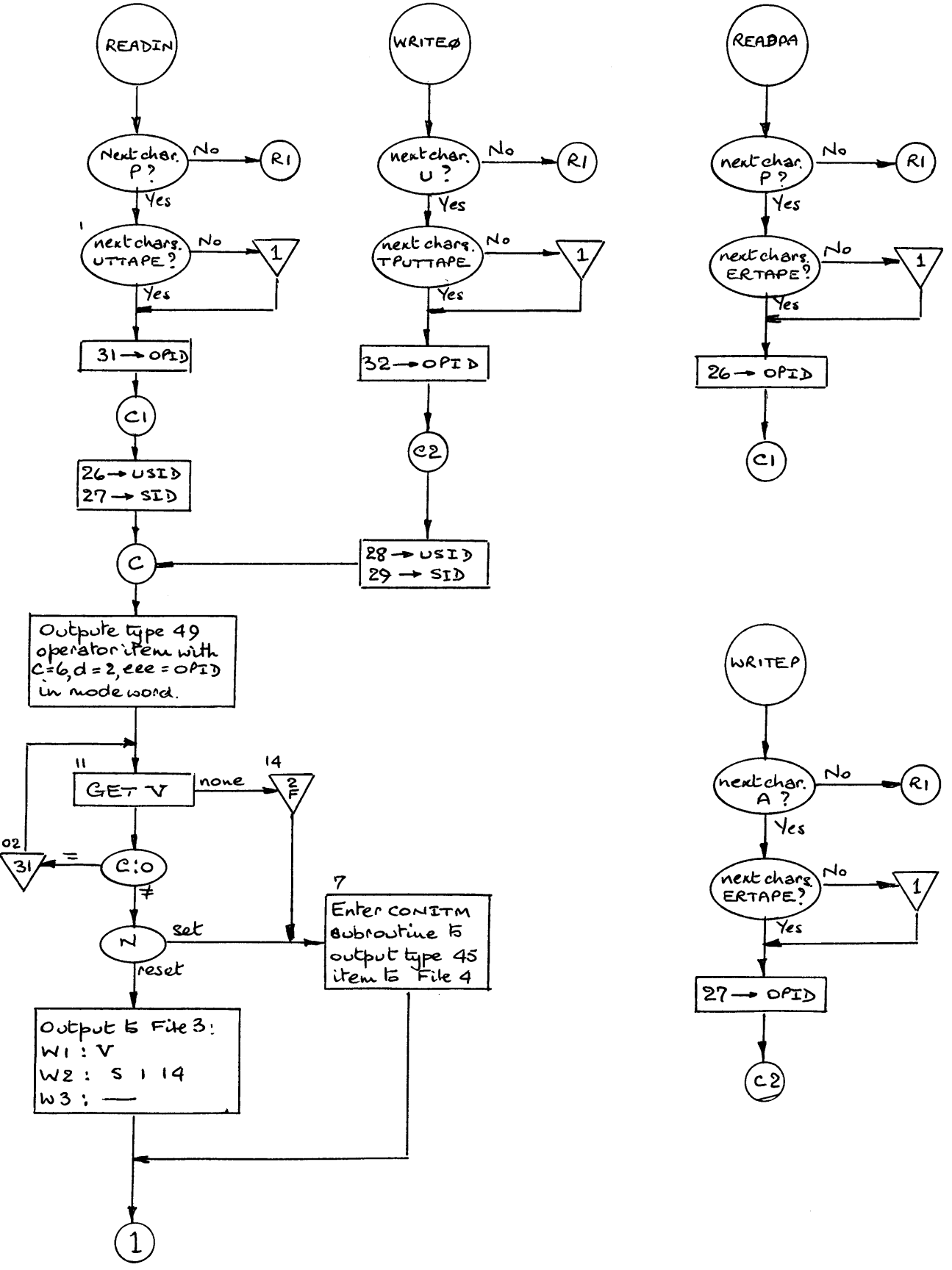



| | |
|---|---|
| α | TB AI CONITM |
| α+1 | Sequence (hi 5 dits) |
| α+2 | Subsequence (next 5 dits) |
| α+3 | return |

READINPUTTAPE i,n,list
READOUTPUTTAPE i,n,list

(1)

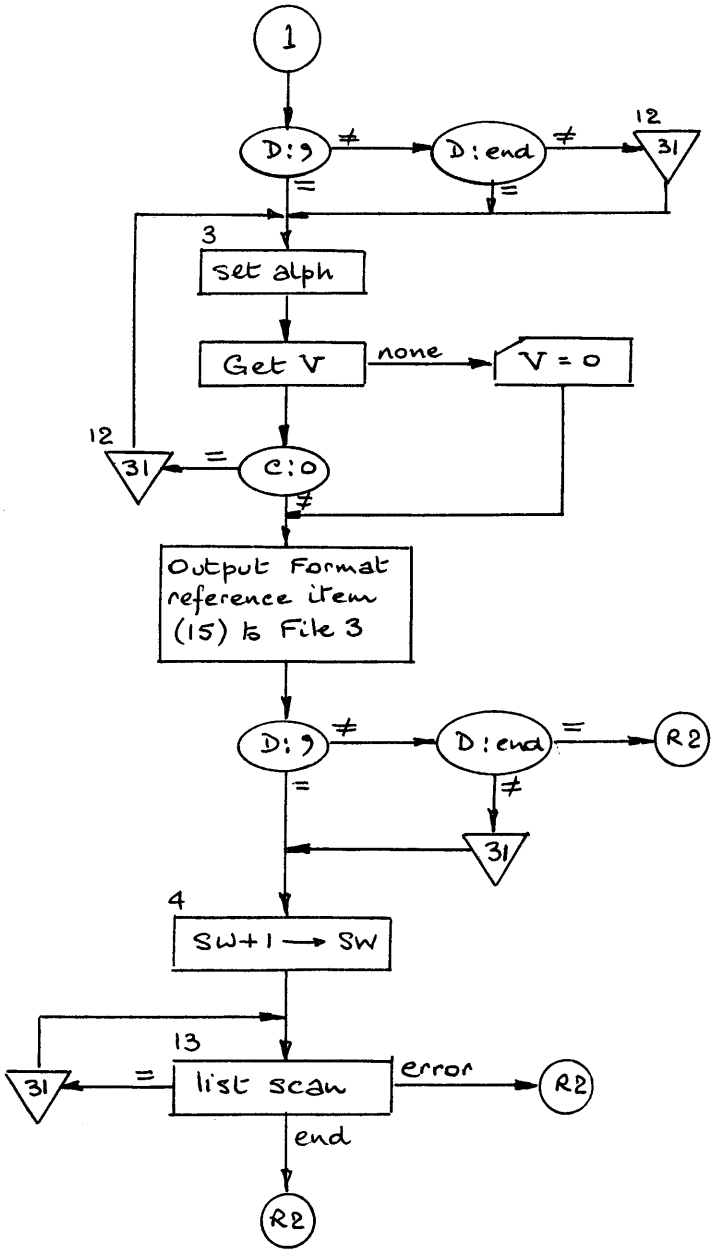
READPAPERTAPE i,n,list
WRITEPAPERTAPE i,n,list
TYPEWRITE i,n,list

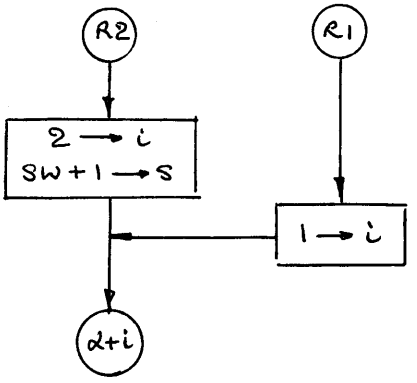


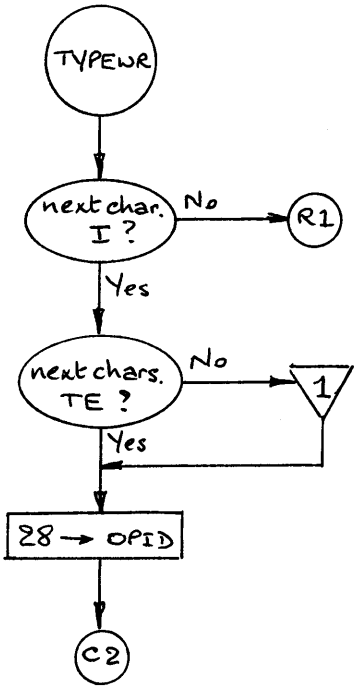
READINPUTTAPE i,n, list
WRITEOUTPUTTAPE i,n, list

(2)

READPAPERTAPE i,n, list
WRITEPAPERTAPE i,n, list
TYPEWRITE i,n, list

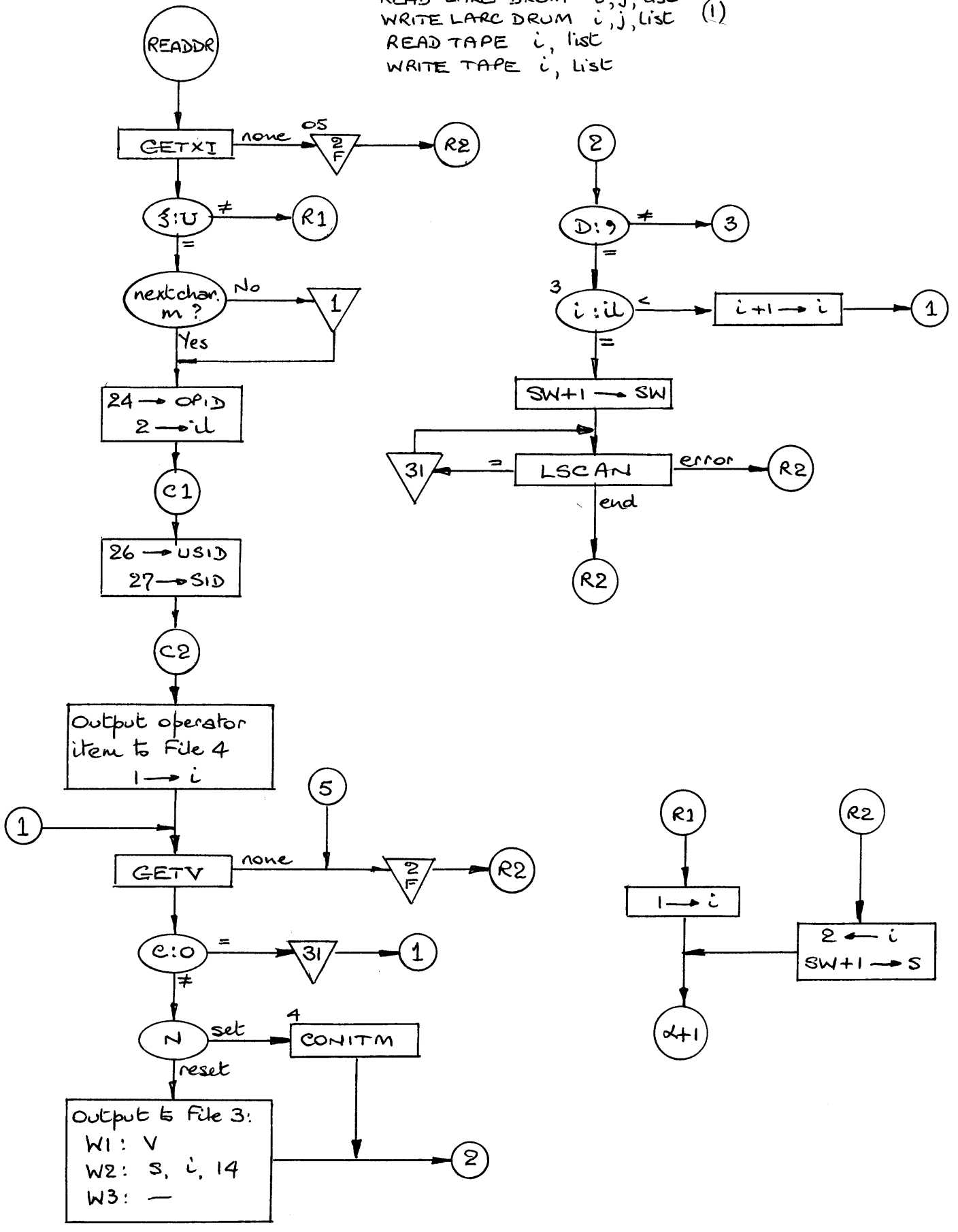


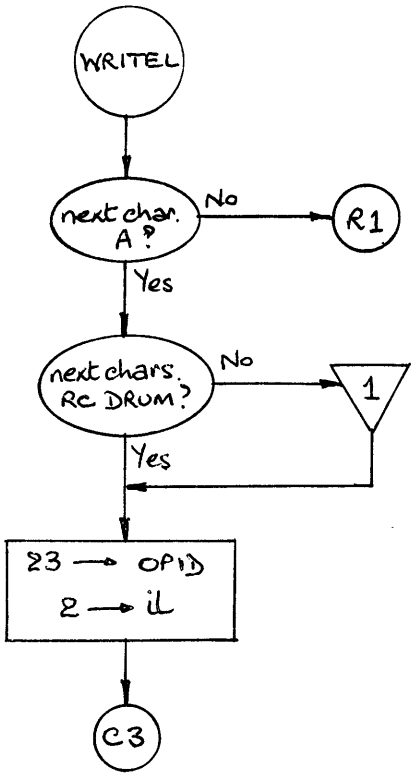
# LARC SCIENTIFIC COMPILER Phase I

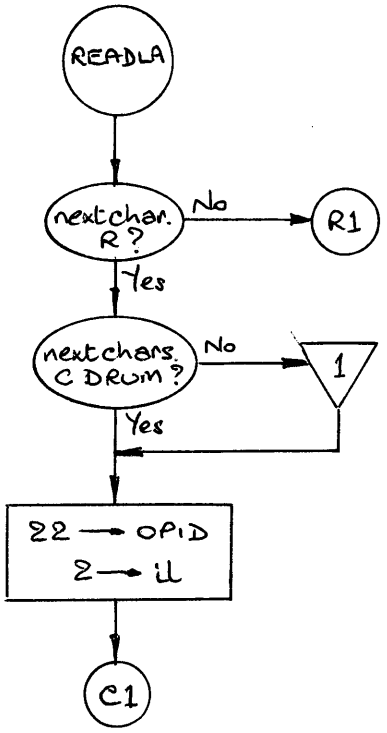
READ DRUM i, j, list
WRITE DRUM i, j, list
READ LARC DRUM i, j, list
WRITE LARC DRUM i, j, list (1)
READ TAPE i, list
WRITE TAPE i, list

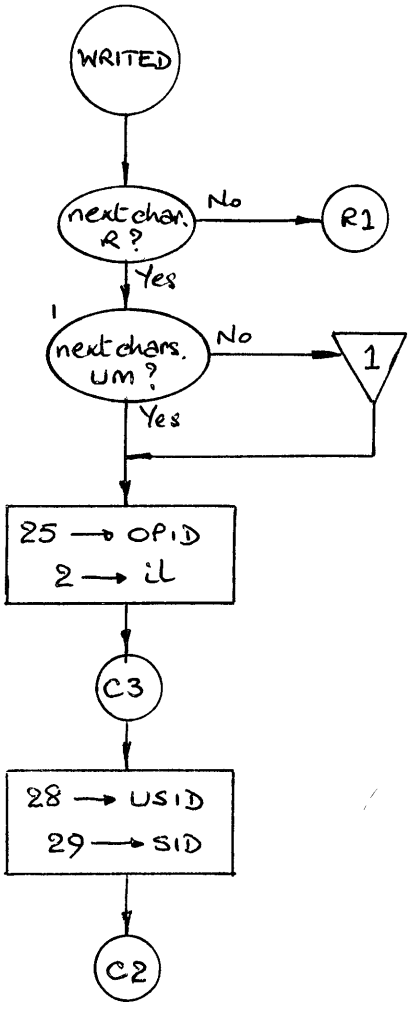


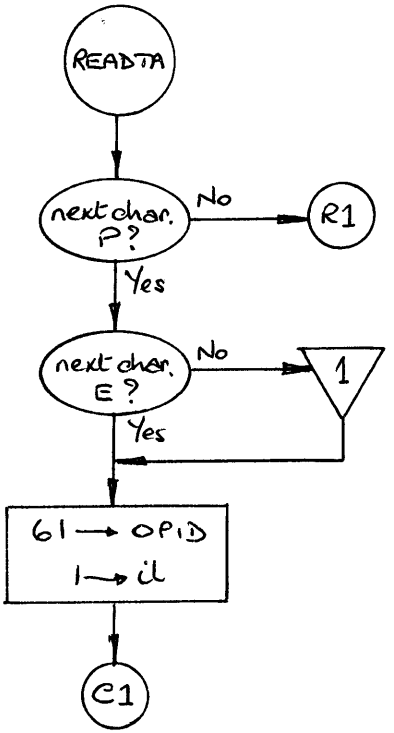
READ DRUM i, j, list
WRITE DRUM i, j, list
READ LARC DRUM i, j, list
WRITE LARC DRUM i, j, list
READ TAPE i, list
WRITE TAPE i, list

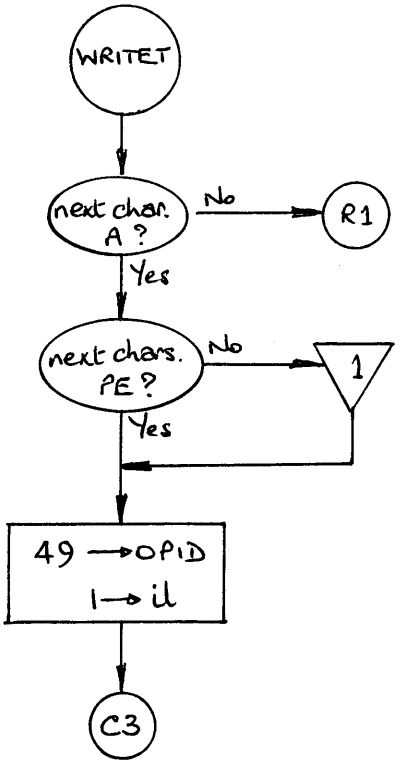
(2)

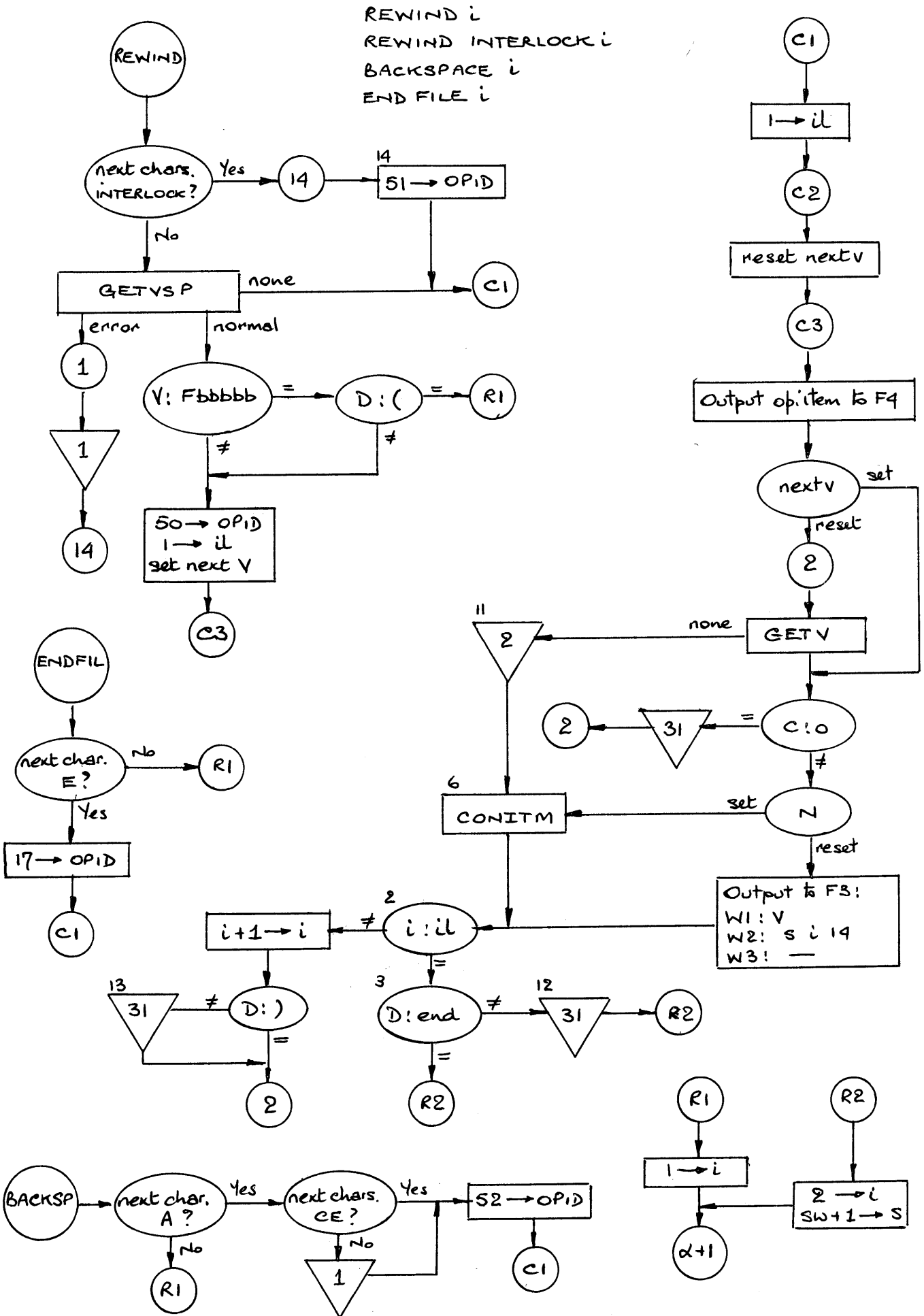


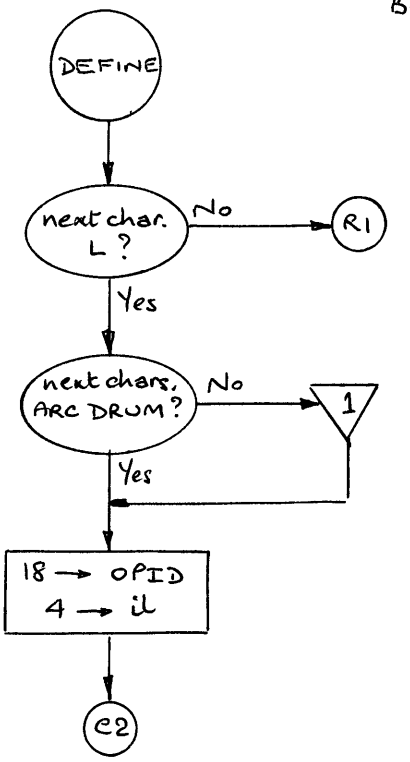



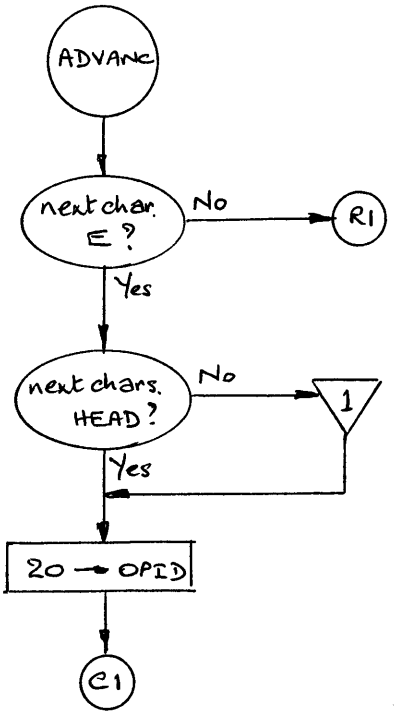



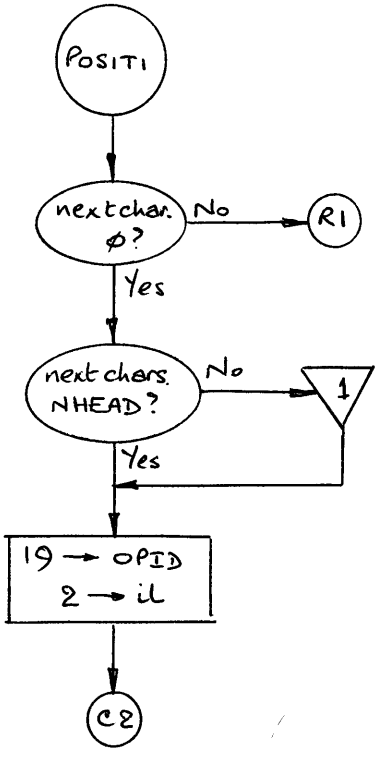



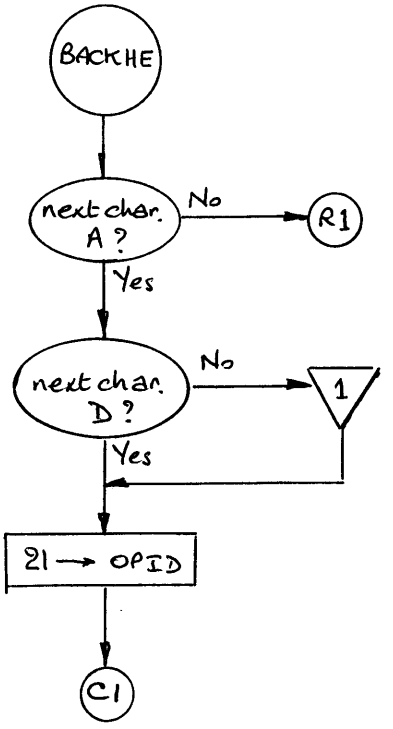

REWIND i
REWIND INTERLOCK i
BACKSPACE i
END FILE i
REWIND
next chars. INTERLOCK?
Yes
14
14
51 → OPID
No
GETVSP
none
C1
error
normal
1
1
14
V: Fbbbbb
=
D: (
=
R1
≠
≠
50 → OPID
1 → il
set next V
C3
C1
1 → il
C2
reset nextV
C3
Output op.item to F4
nextv
set
reset
2
11
2
none
GETV
ENDFIL
next char. E?
No
R1
Yes
17 → OPID
C1
2
31
=
C:0
≠
6
CONITM
set
N
reset
Output to FS:
W1: V
W2: S i 14
W3: —
2
i+1 → i
≠
i : il
=
13
31
≠
D: )
=
2
3
D: end
≠
12
31
R2
=
R2
BACKSP
next char. A ?
Yes
No
R1
next chars. CE?
Yes
No
1
52 → OPID
C1
R1
1 → i
α+1
R2
2 → i
SW+1 → S

DEFINE LARC DRUM i, j, k, l.
POSITION HEAD i, j
ADVANCE HEAD i
BACK HEAD i



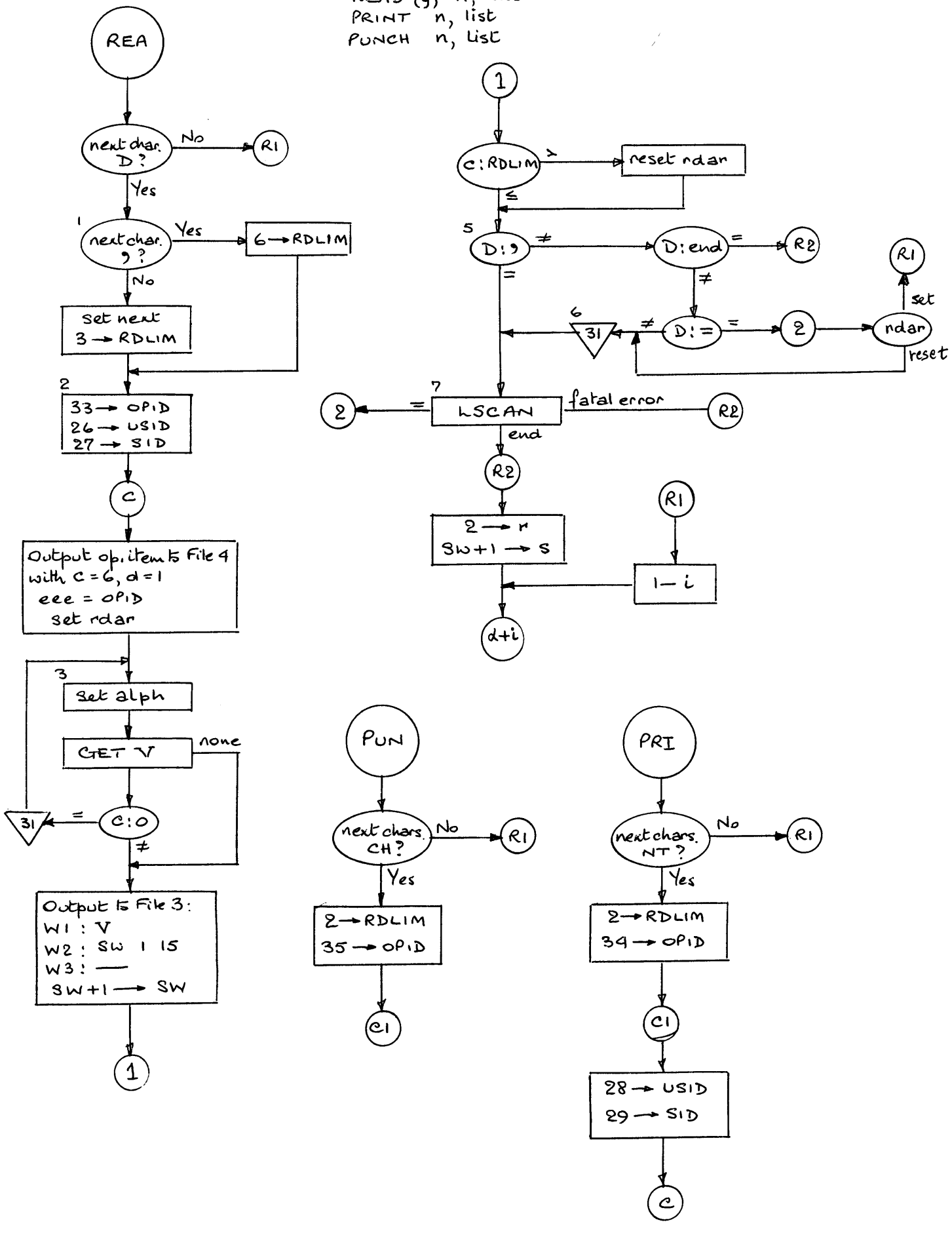



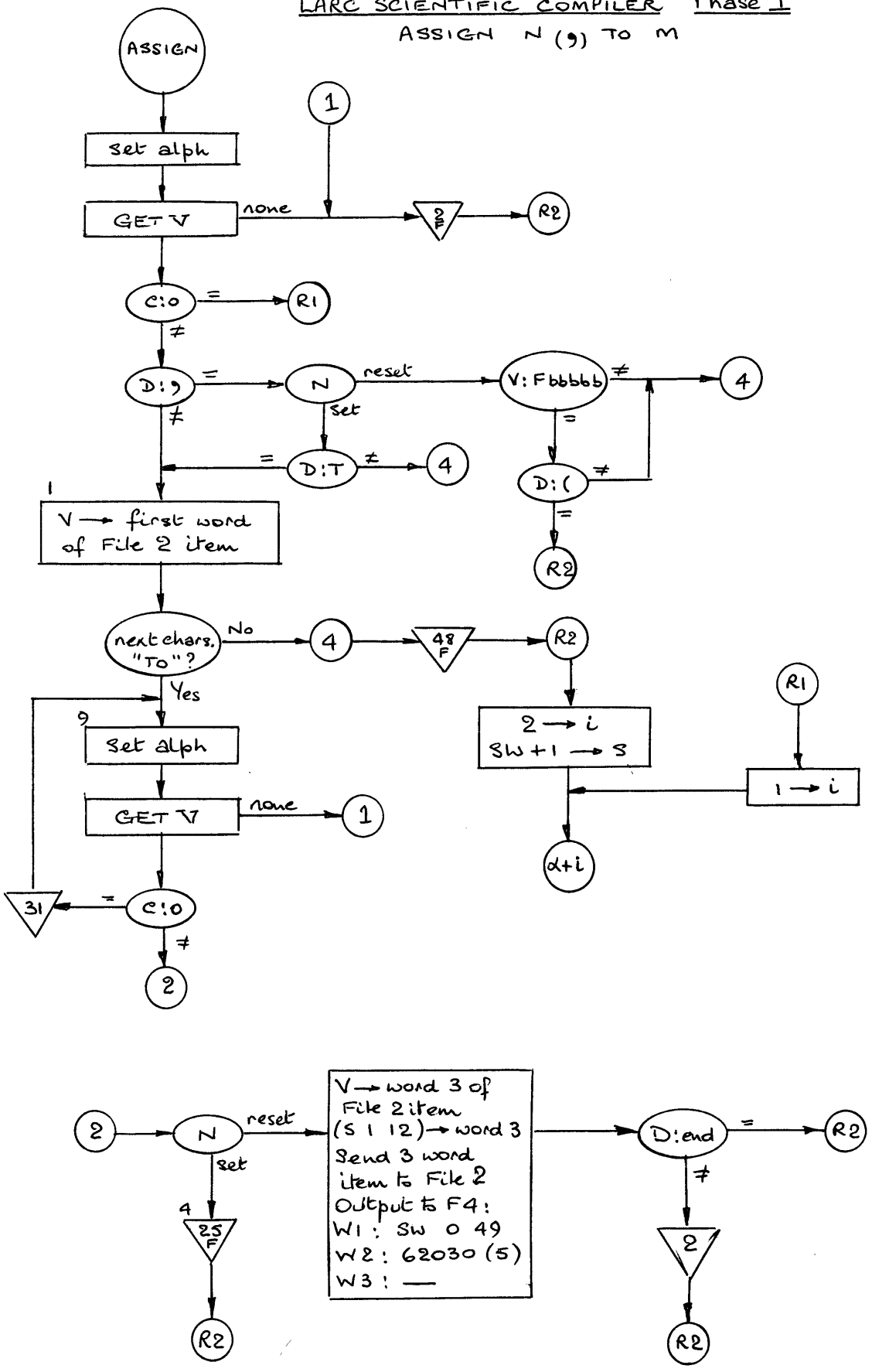



# LARC SCIENTIFIC COMPILER Phase I

READ (9) n, list
PRINT n, list
PUNCH n, list

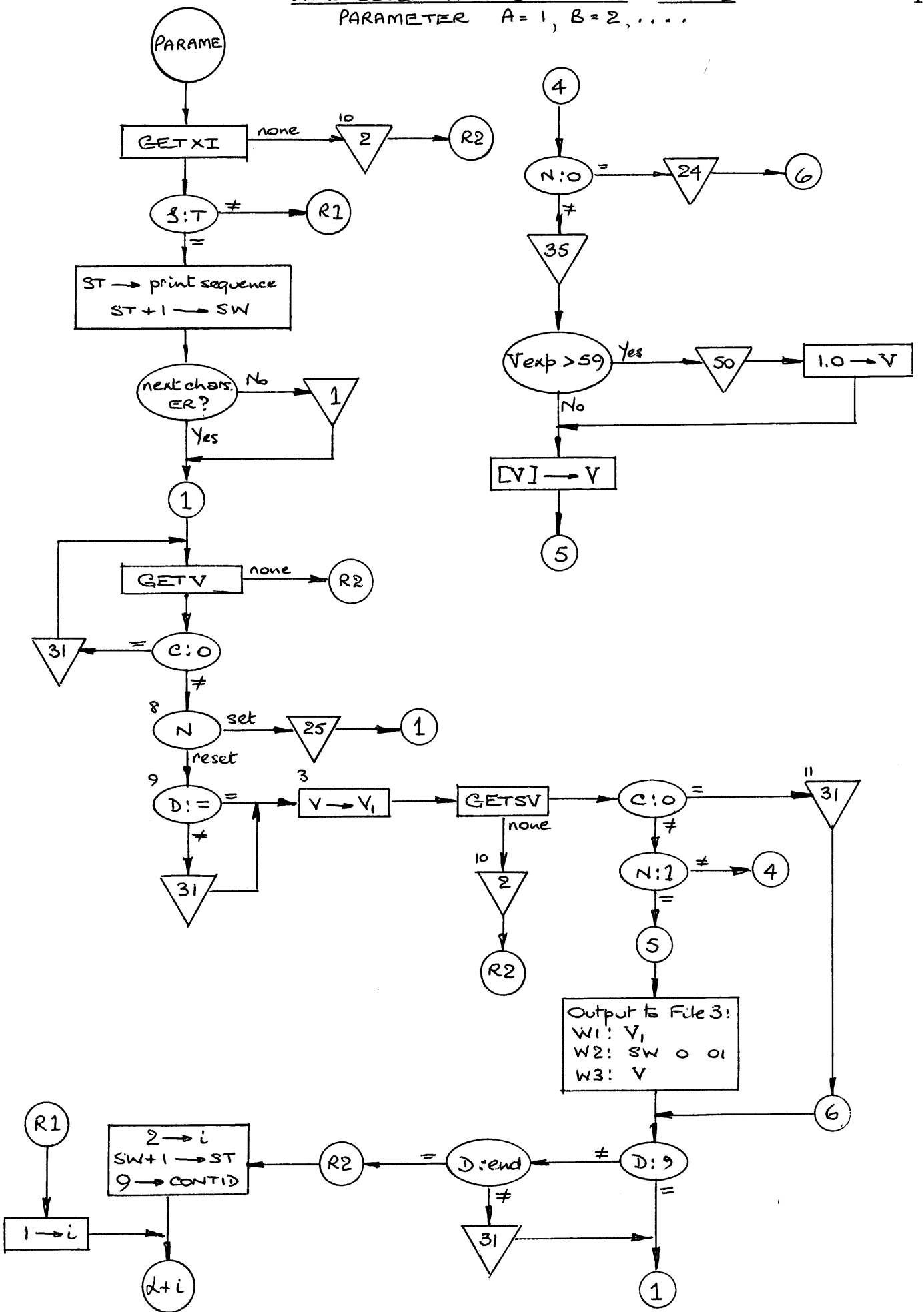


ASSIGN N (9) TO M



# LARC SCIENTIFIC COMPILER Phase I

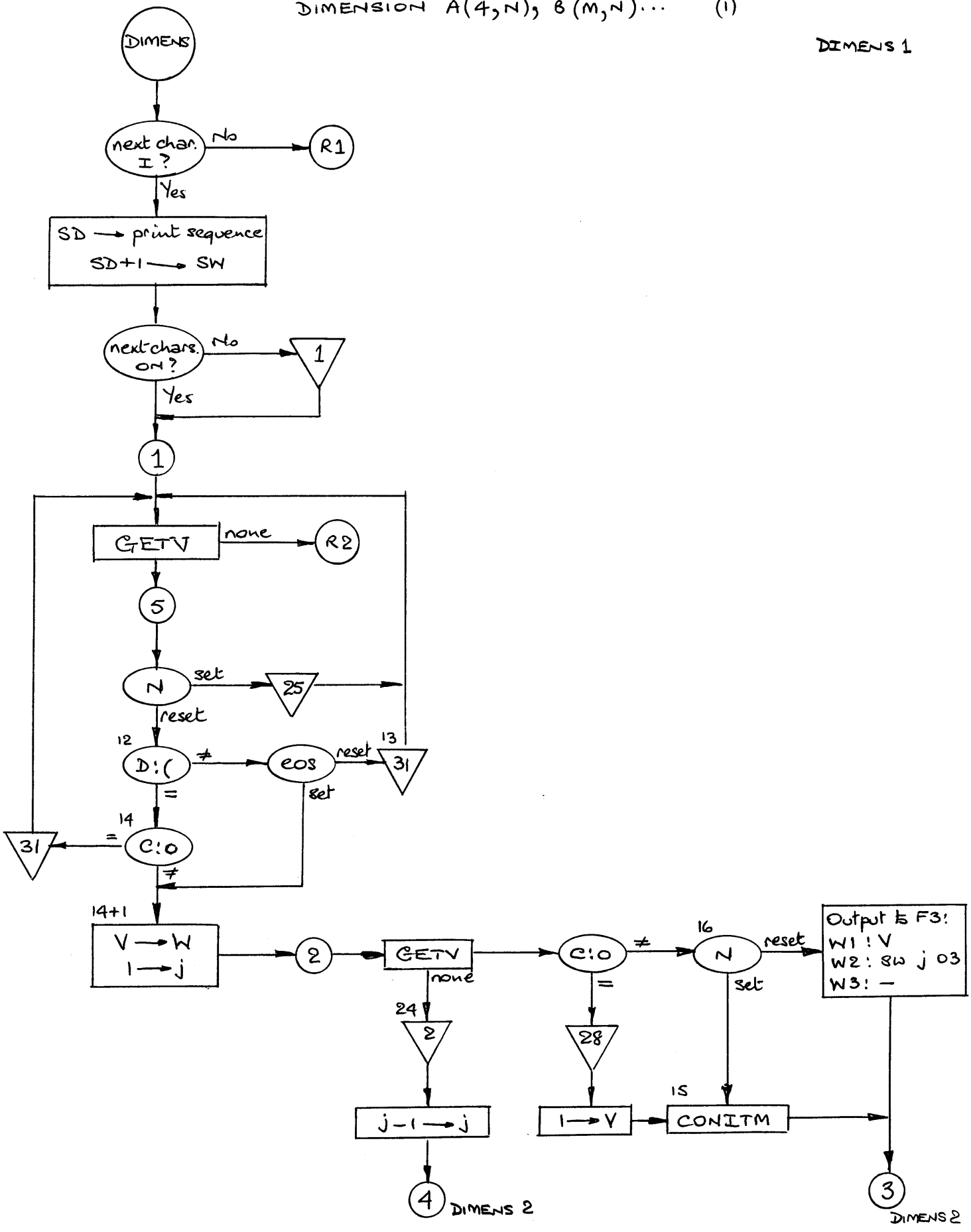
PARAMETER A = 1, B = 2, . . . .



# LARC SCIENTIFIC COMPILER Phase I

DIMENSION A(4,N), B(M,N)... (1)

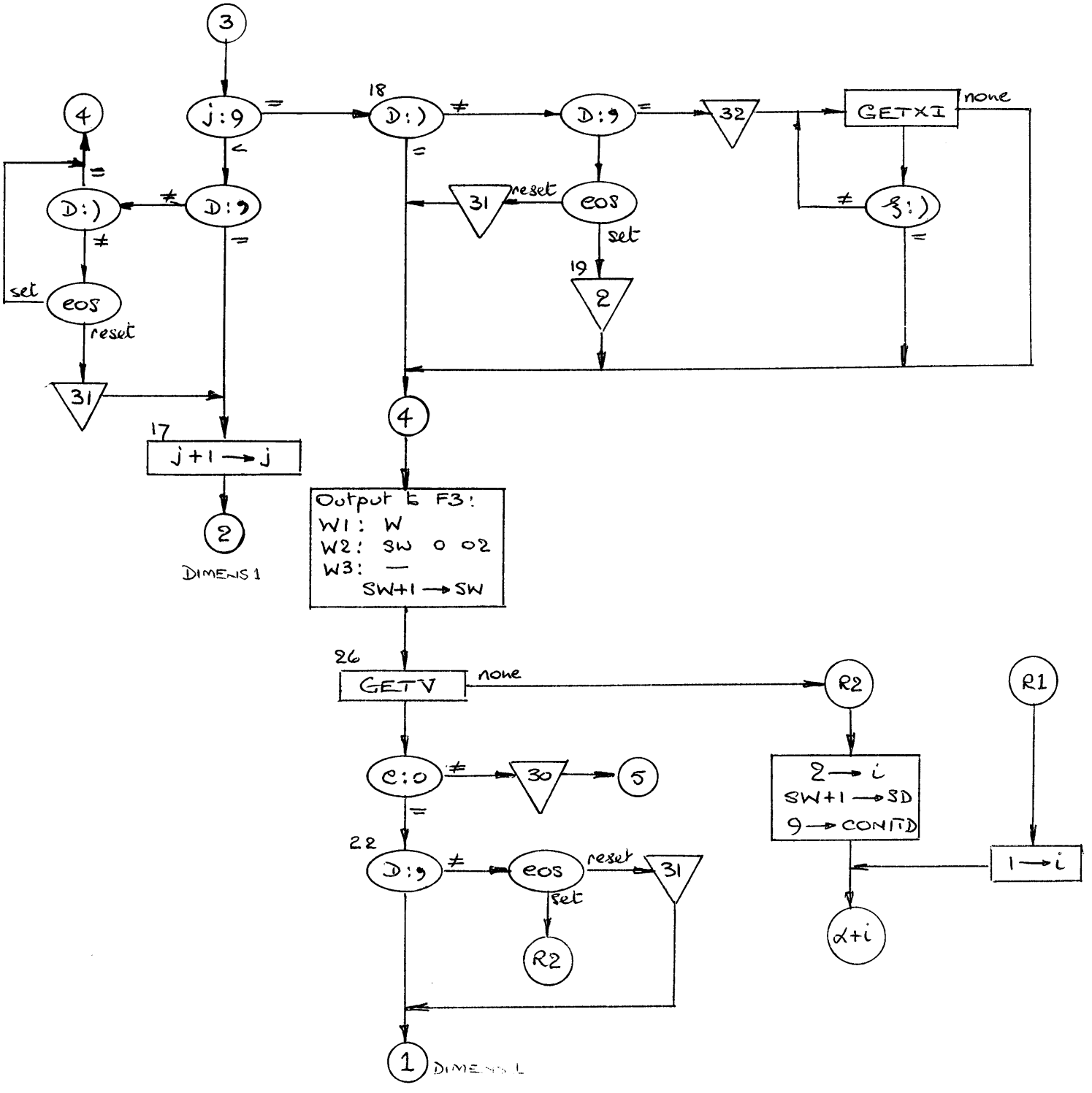
DIMENS 1



# LARC SCIENTIFIC COMPILER Phase I

DIMENSION A(4,N), B(M,N)... (2)

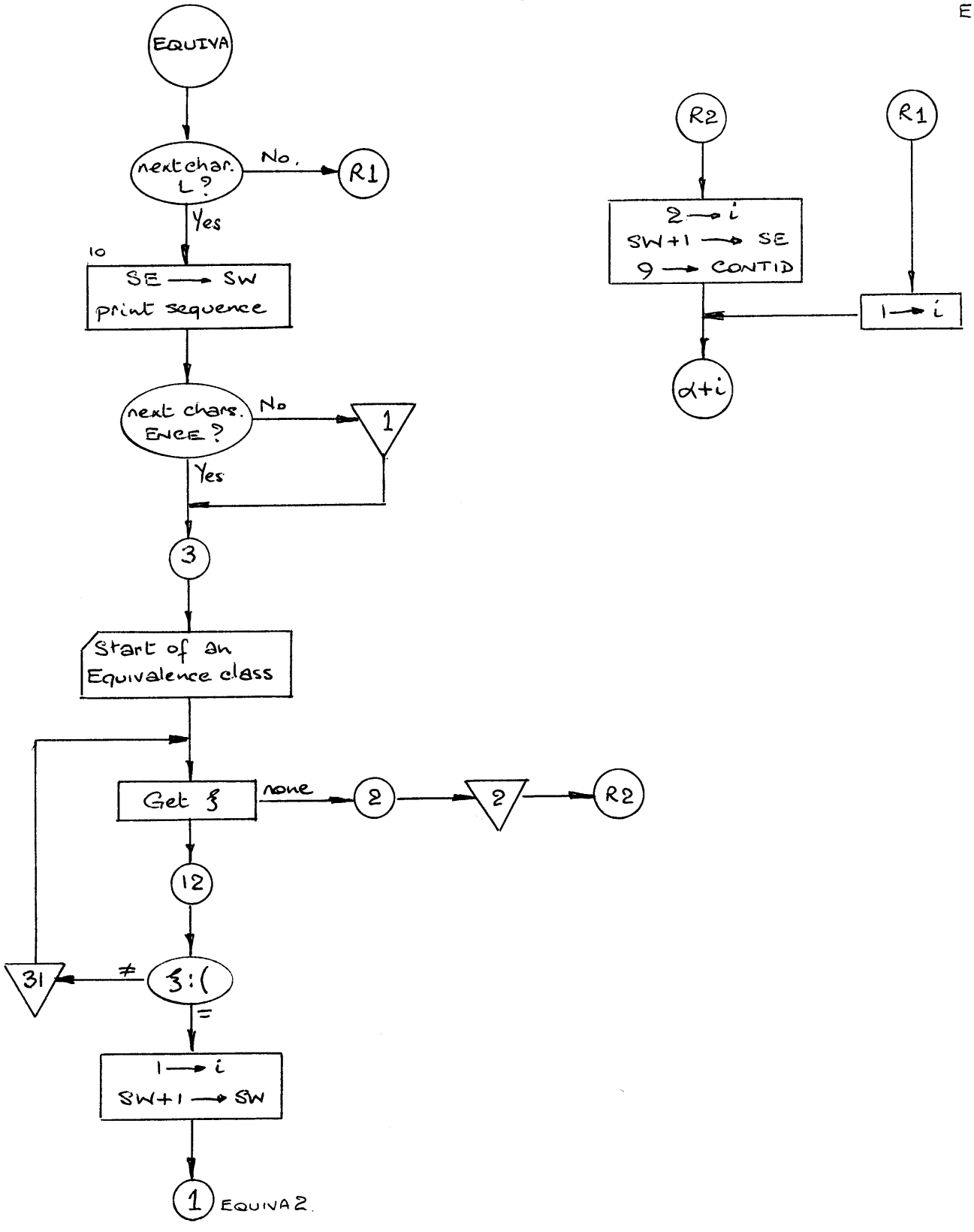
DIMENS 2



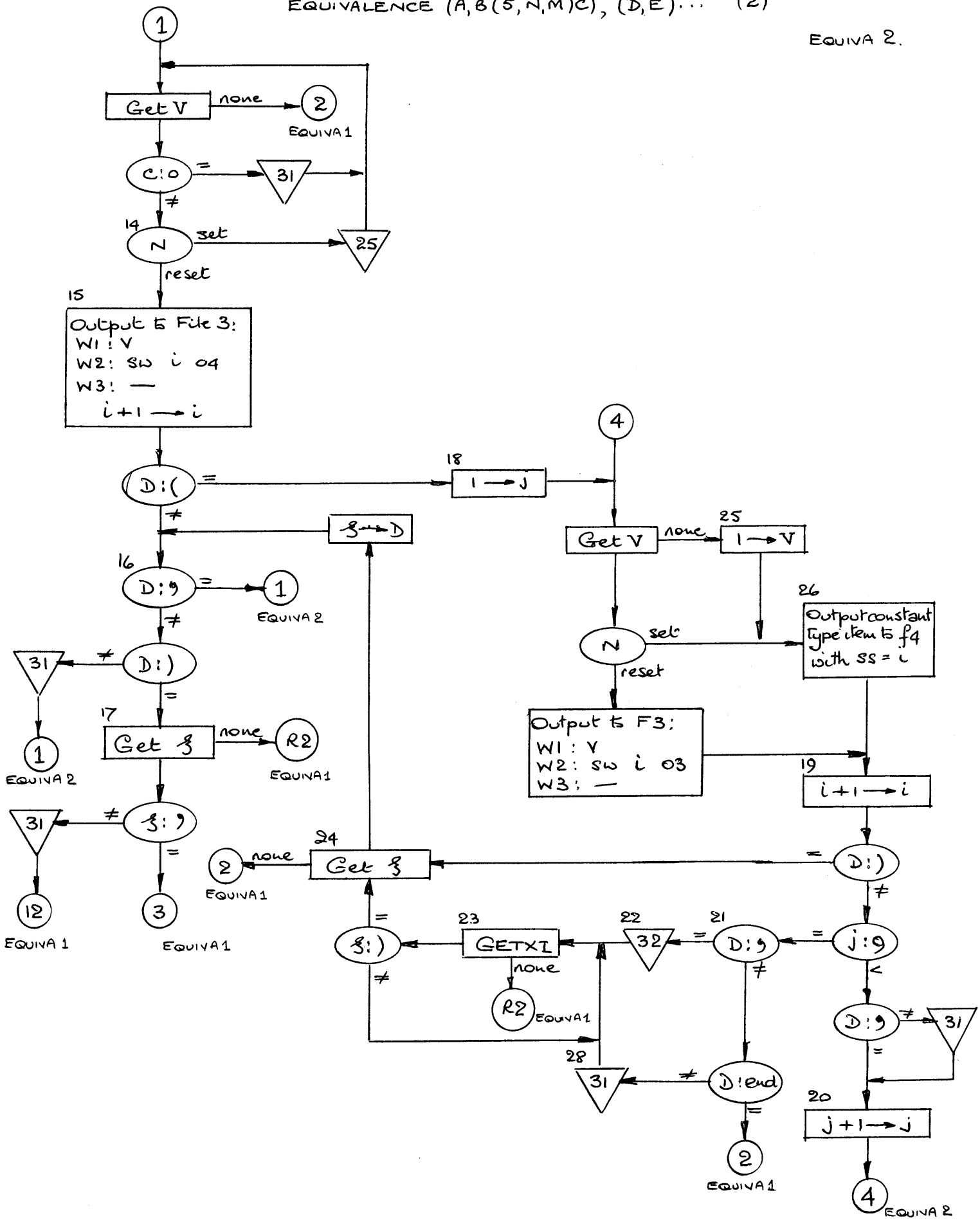
LARC SCIENTIFIC COMPILER Phase I

EQUIVALENCE (A,B (5,N,M)C), (D,E)... (1)

EQUIVA 1

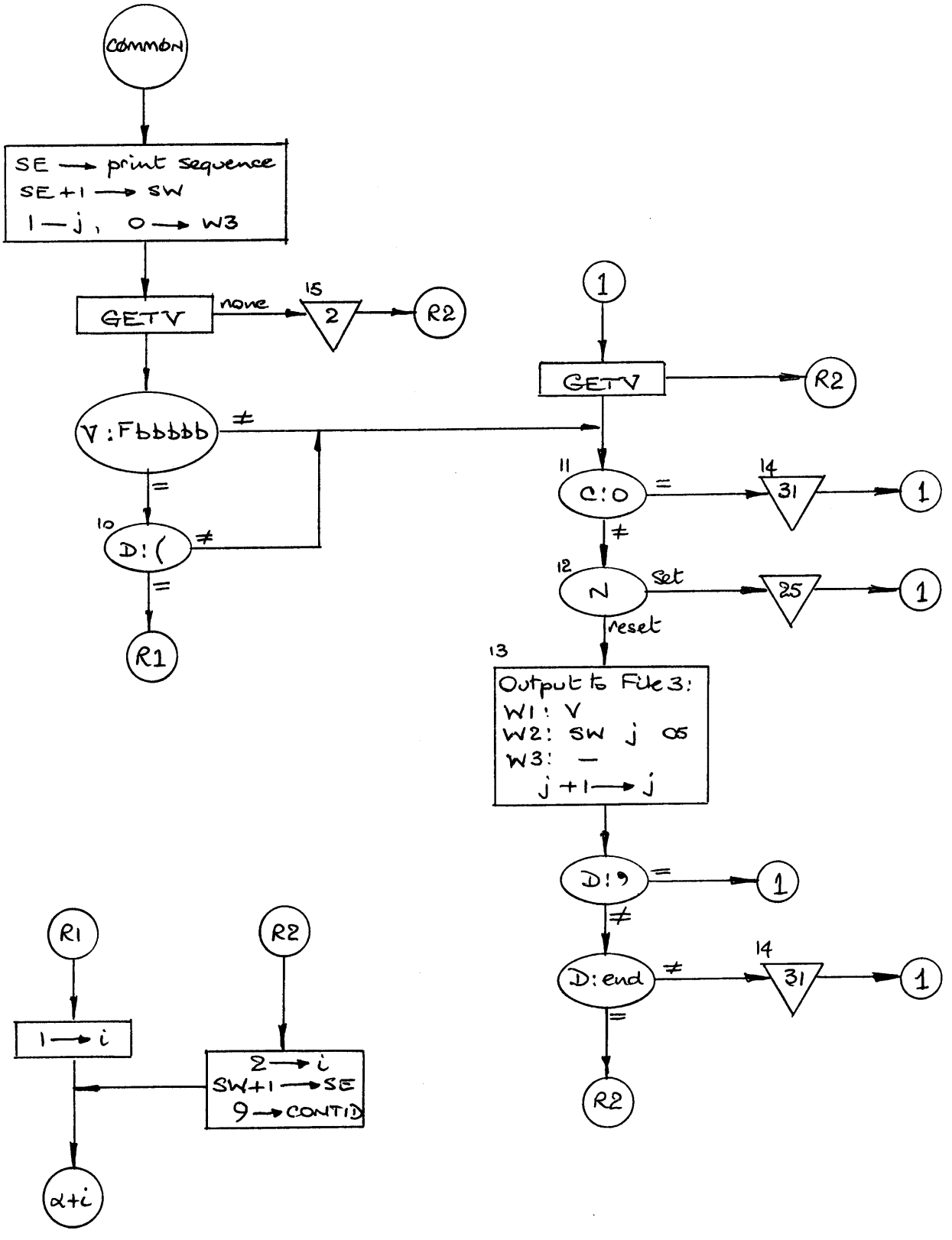


EQUIVA 2.


1
Get V
none
2
EQUIVA 1
C:0
=
31
≠
14
N
set
25
reset
15
Output to File 3:
W1 : V
W2: SW i 04
W3: —
i+1 → i
D:(
=
18
1 → J
4
≠
g → D
Get V
none
25
1 → V
16
D:9
=
1
EQUIVA 2
≠
N
set
reset
26
Output constant
type item to f4
with SS = i
31
≠
D:)
=
Output to F3:
W1 : V
W2: SW i 03
W3: —
17
Get g
none
R2
EQUIVA 1
19
i+1 → i
1
EQUIVA 2
31
≠
g:)
=
24
2
none
Get g
=
D:)
≠
EQUIVA 1
12
EQUIVA 1
3
EQUIVA 1
=
g:)
23
GETXI
22
32
=
21
D:9
=
j:9
≠
none
R2
EQUIVA 1
<
D:9
≠
31
=
28
31
≠
D!end
=
20
j+1 → j
2
EQUIVA 1
4
EQUIVA 2

# LARC SCIENTIFIC COMPILER Phase I

COMMON A, B, C



INTEGER A, B, C

FLOATING

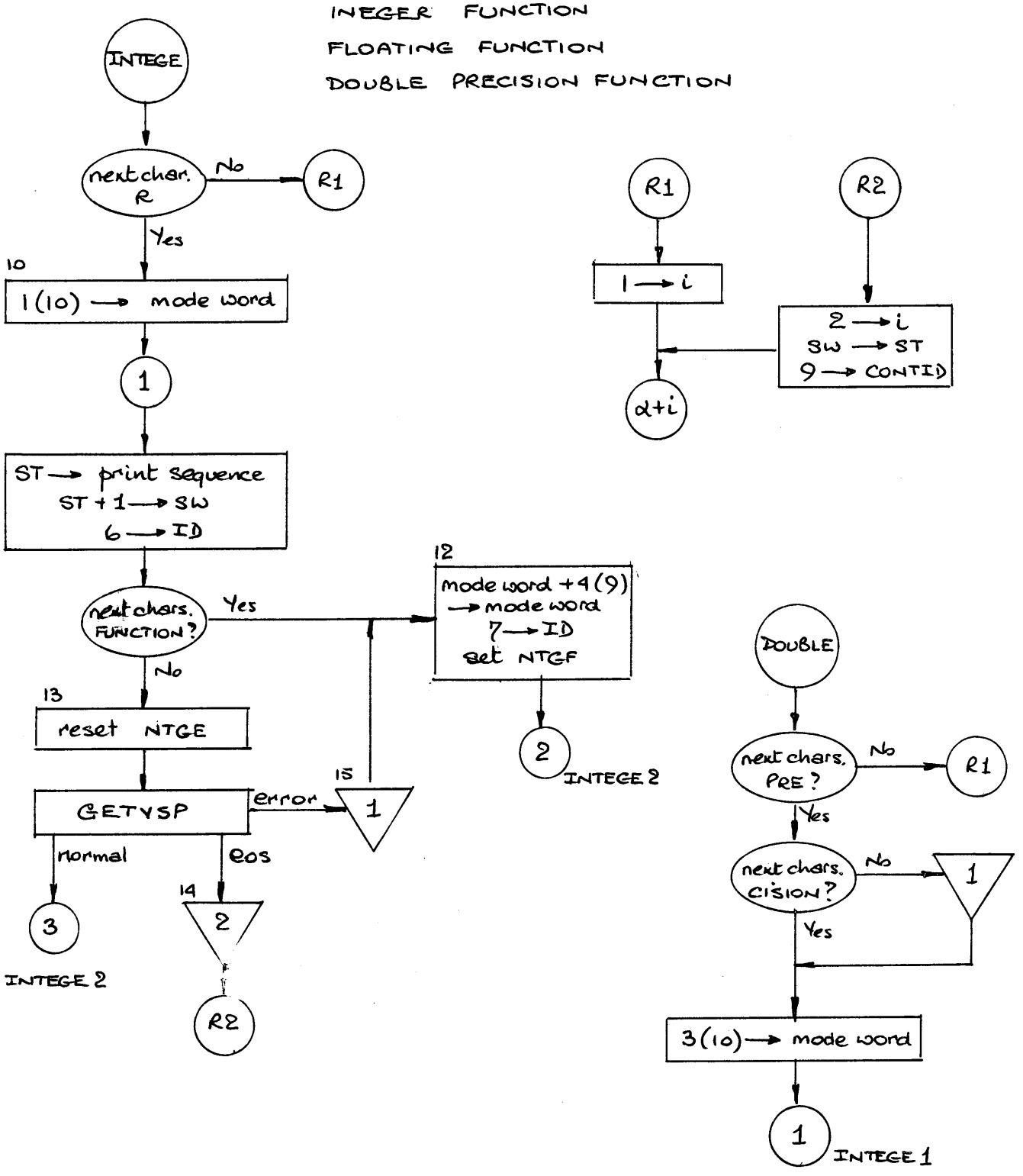
DOUBLE PRECISION

INEGER FUNCTION

FLOATING FUNCTION

DOUBLE PRECISION FUNCTION

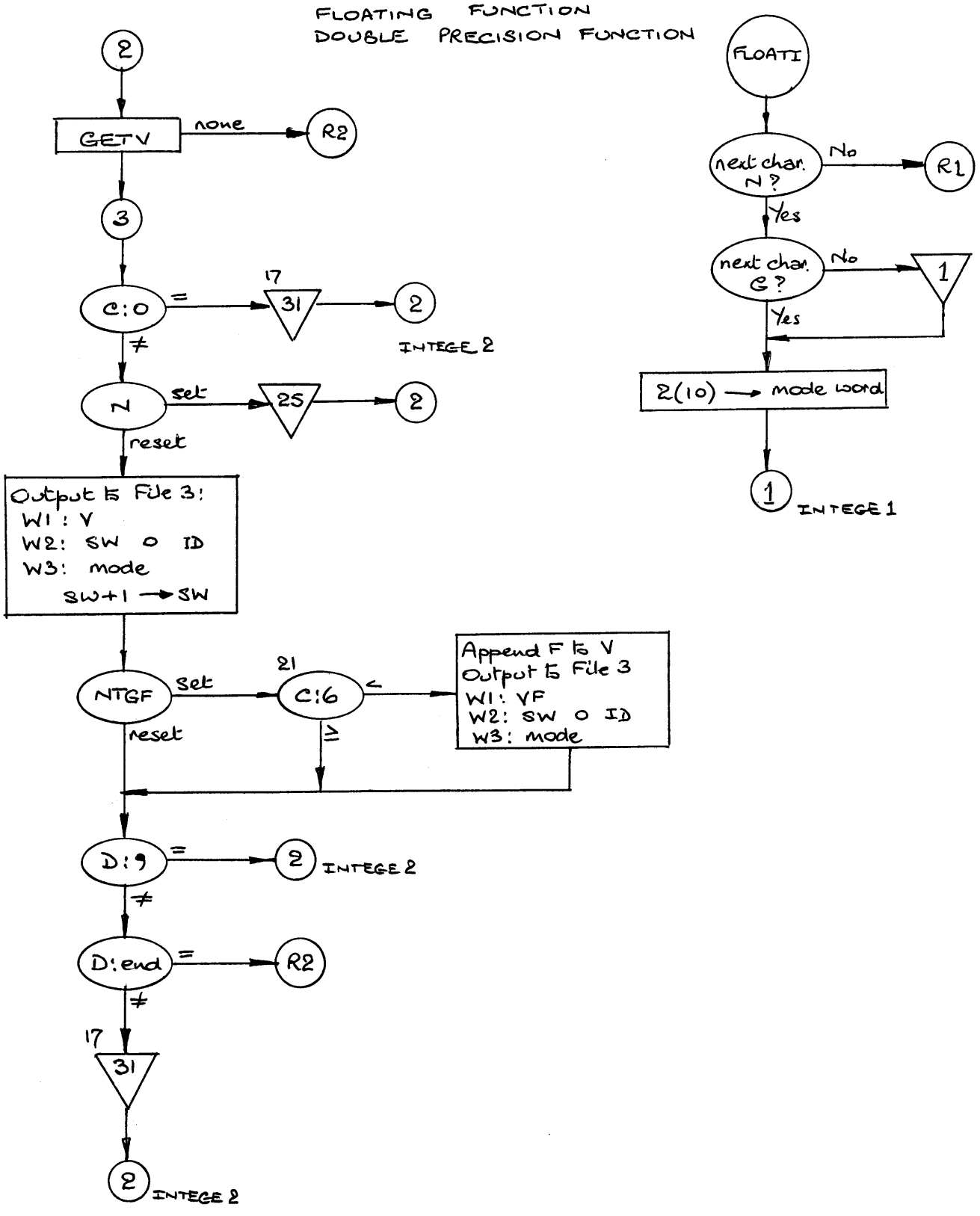
INTEGE 1



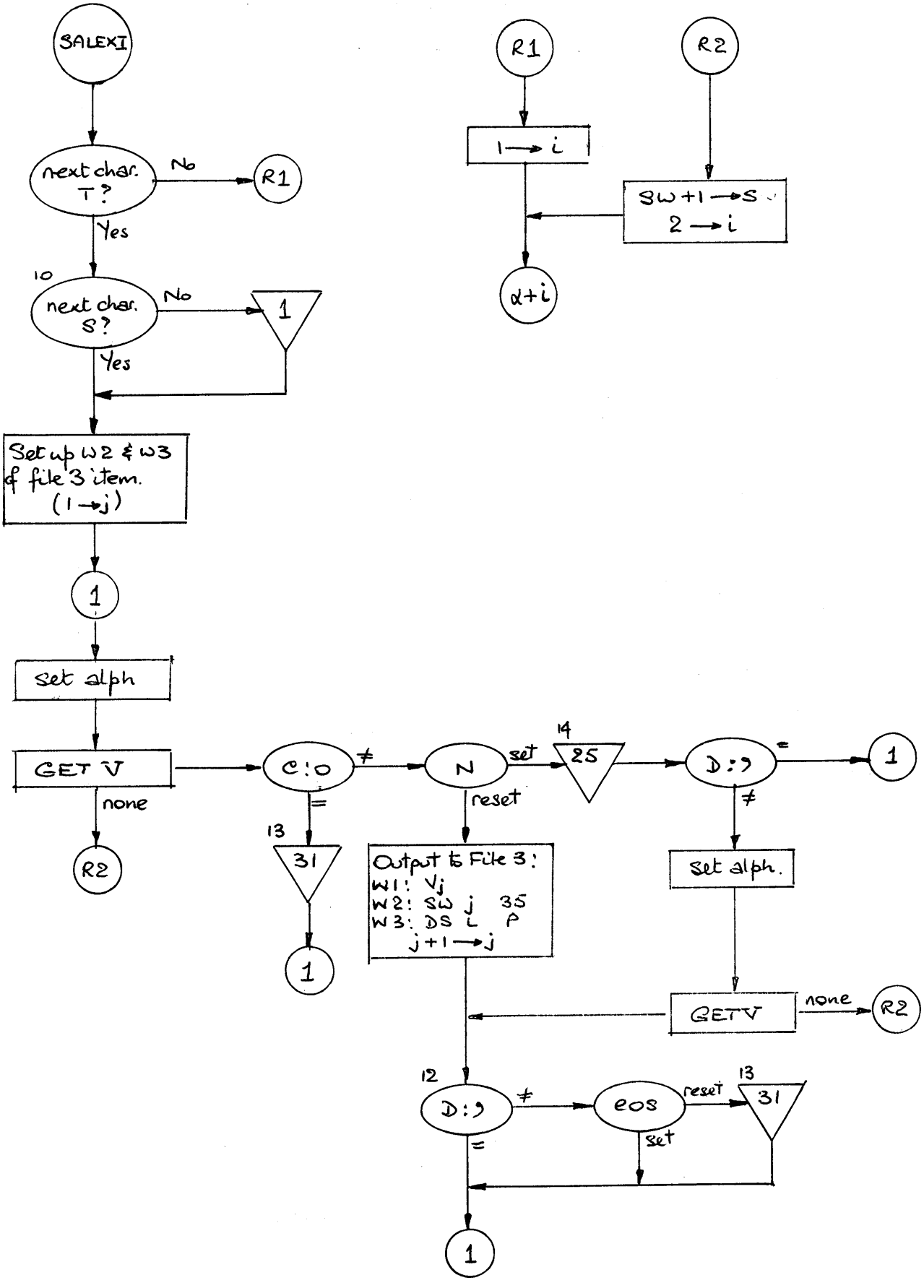
# LARC SCIENTIFIC COMPILER Phase I

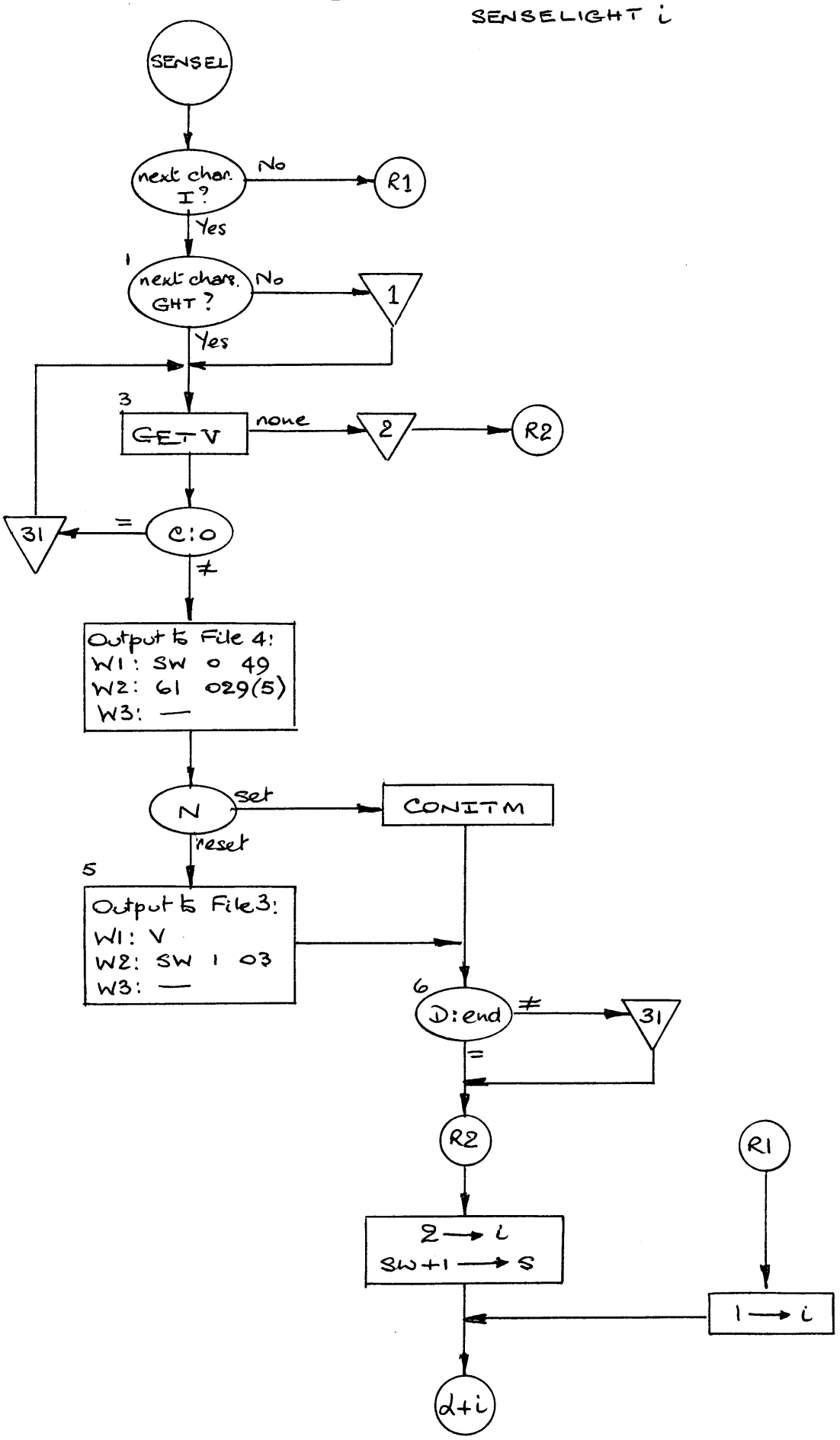
INTEGER A, B, C
FLOATING
DOUBLE PRECISION
INTEGER FUNCTION
FLOATING FUNCTION
DOUBLE PRECISION FUNCTION

(2)

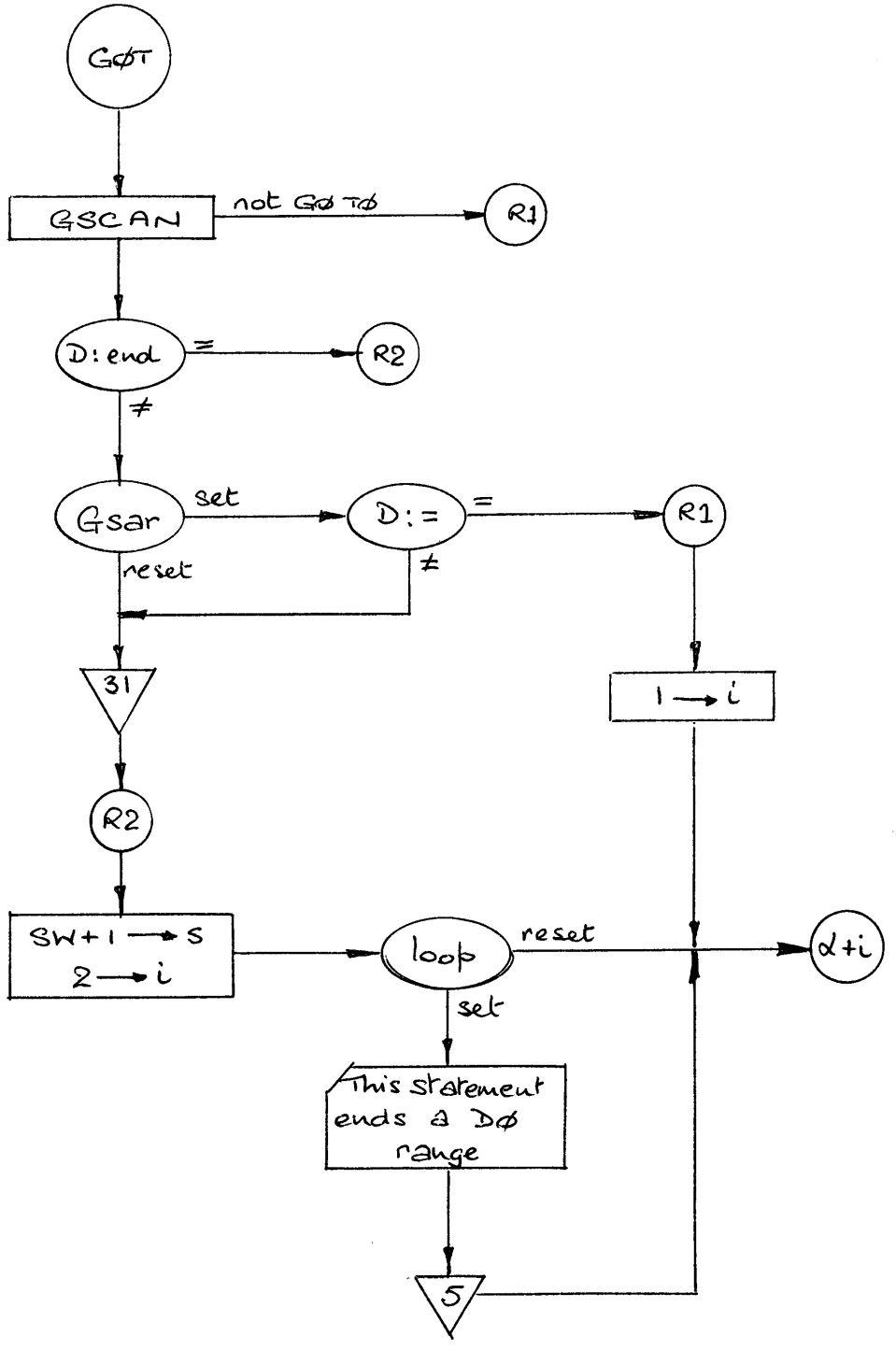


SALEXITS A, B, C




SENSEL
next char. I?
No
R1
Yes
1
next char. GHT ?
No
1
Yes
3
GET V
none
2
R2
31
=
C:0
≠
Output to File 4:
W1: SW 0 49
W2: 61 029(5)
W3: —
N
set
CONITM
reset
5
Output to File3:
W1: V
W2: SW 1 03
W3: —
6
D:end
≠
31
=
R2
R1
2 → i
SW+1 → S
1 → i
d+i

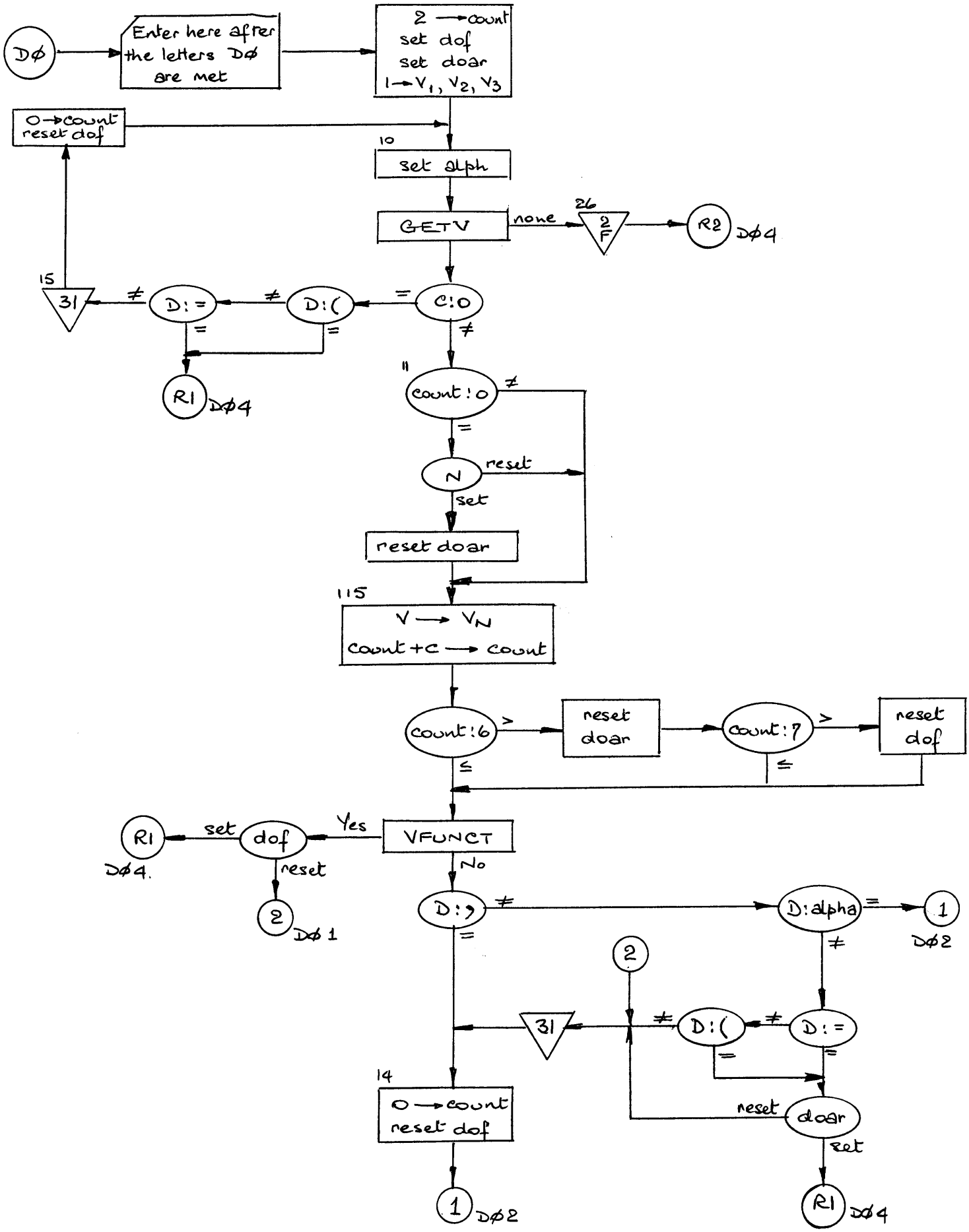
GØTØ n
GØTØ n, ($n_1$, $n_2$, . . .)
GØTØ n, ($n_1$, $n_2$, . . .), n

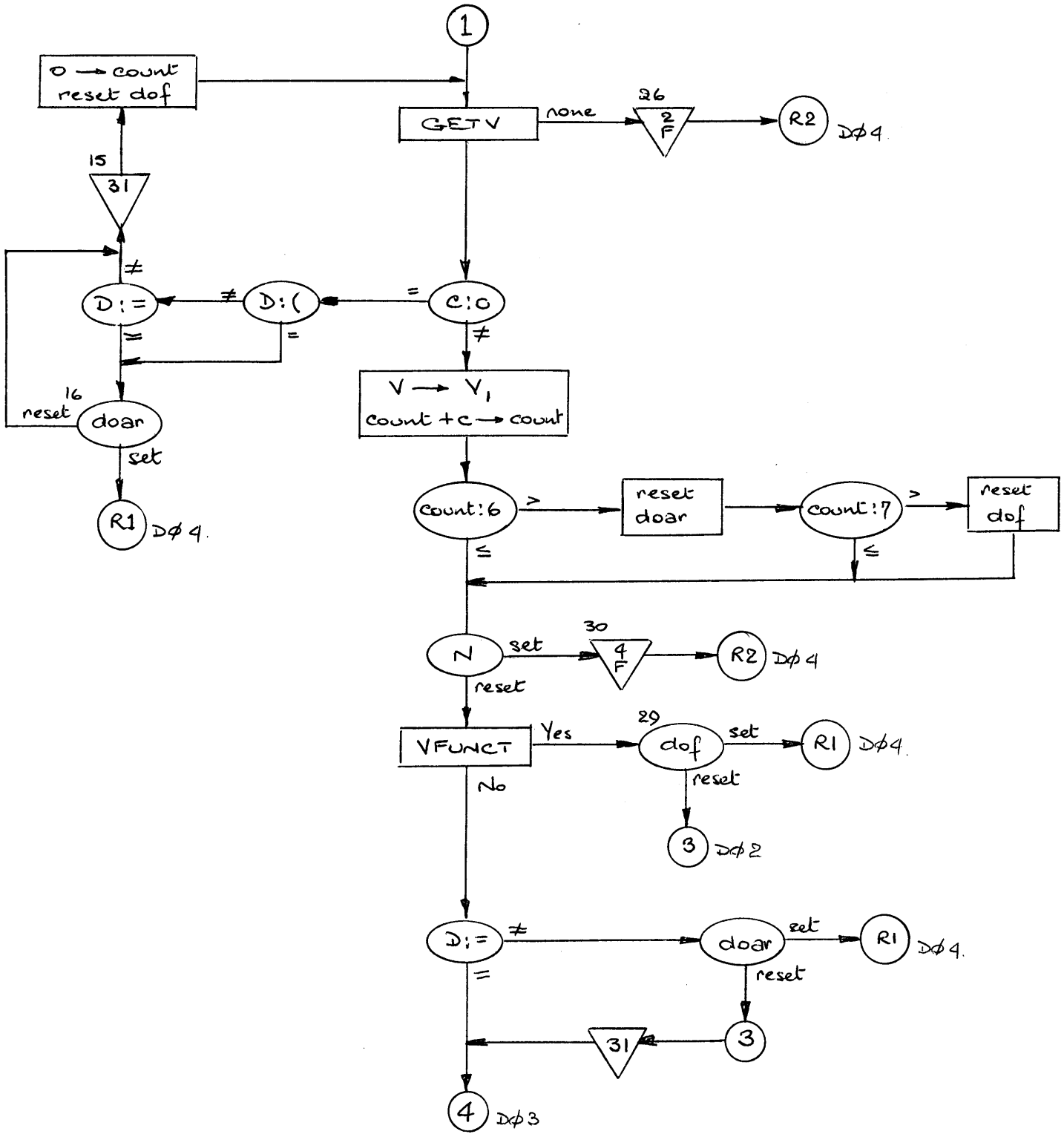


# LARC SCIENTIFIC COMPILER Phase I

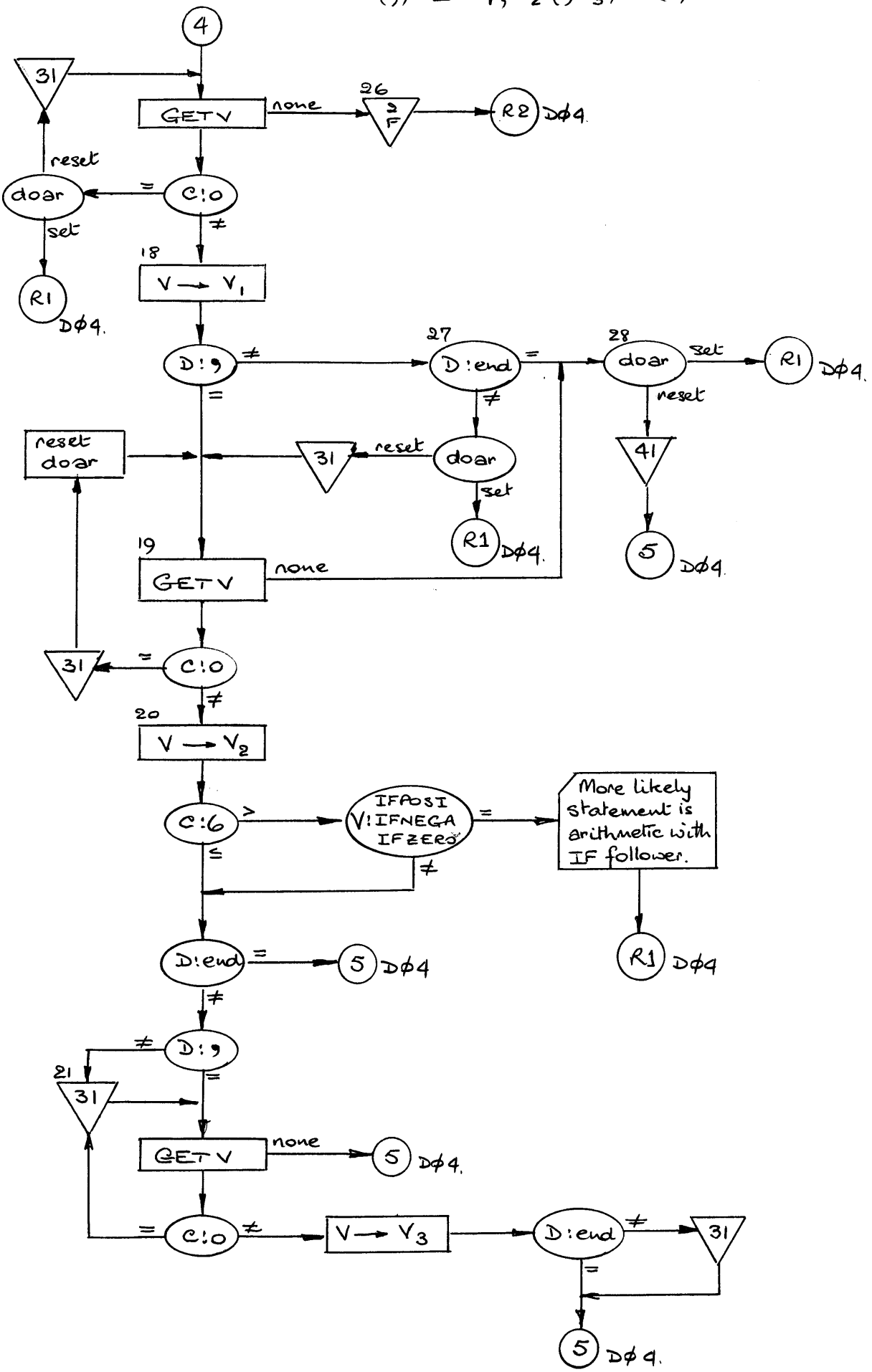
Dφ α (,) I = $i_1$ , $i_2$ (, $i_3$) (1)

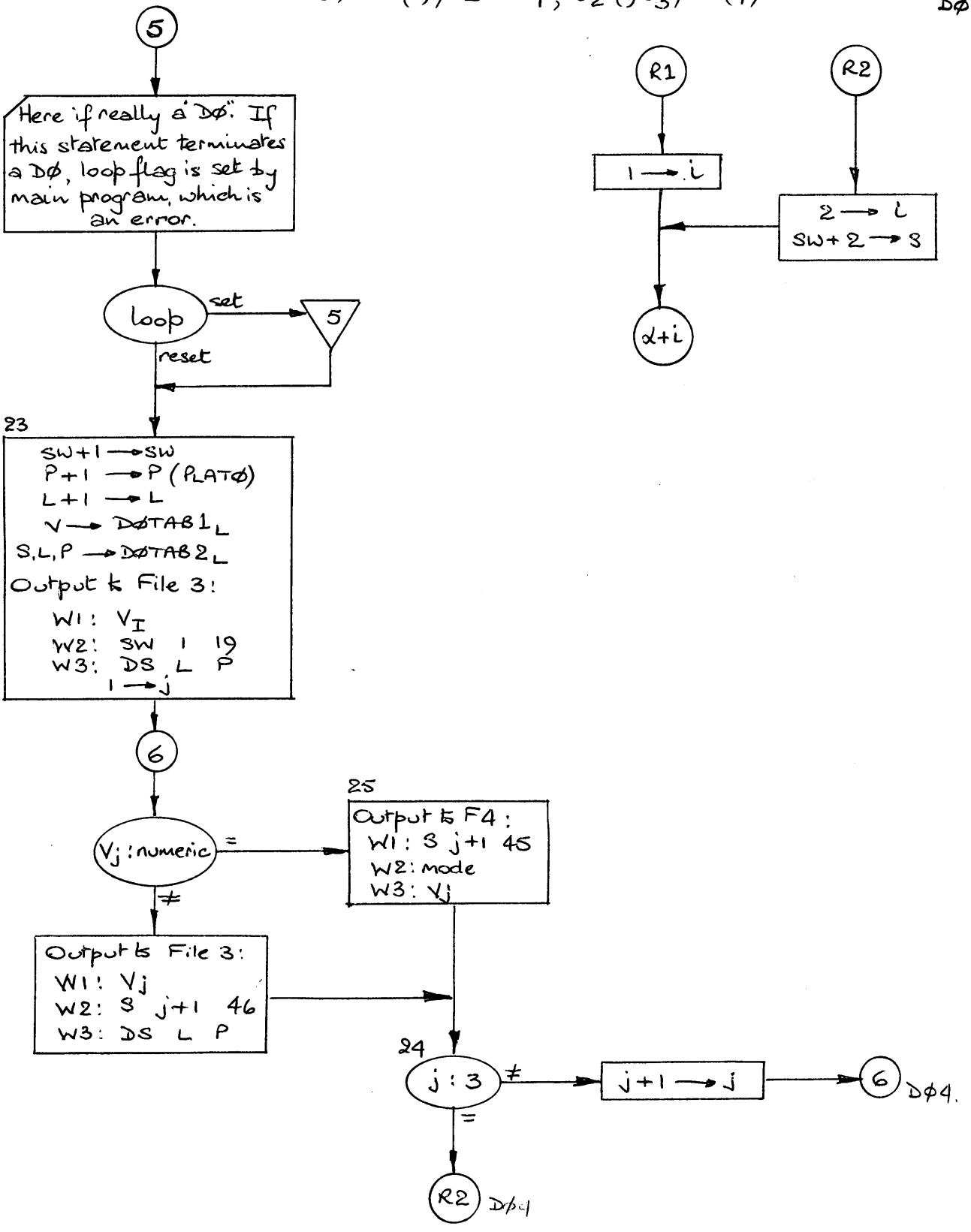
Dφ 1




1
0 → count
reset dof
GETV
none
26
2
F
R2
DØ 4
15
31
≠
D:=
≠
D:(
=
C:0
=
=
=
≠
16
reset
doar
set
R1
DØ 4
V → V1
count + c → count
count:6
>
reset
doar
count:7
>
reset
dof
≤
≤
N
set
30
4
F
R2
DØ 4
reset
VFUNCT
Yes
29
dof
set
R1
DØ 4
No
reset
3
DØ 2
D:=
≠
doar
set
R1
DØ 4
=
reset
31
3
4
DØ 3

DØ α (9) I = $i_1$, $i_2$ (9 $i_3$) (3)

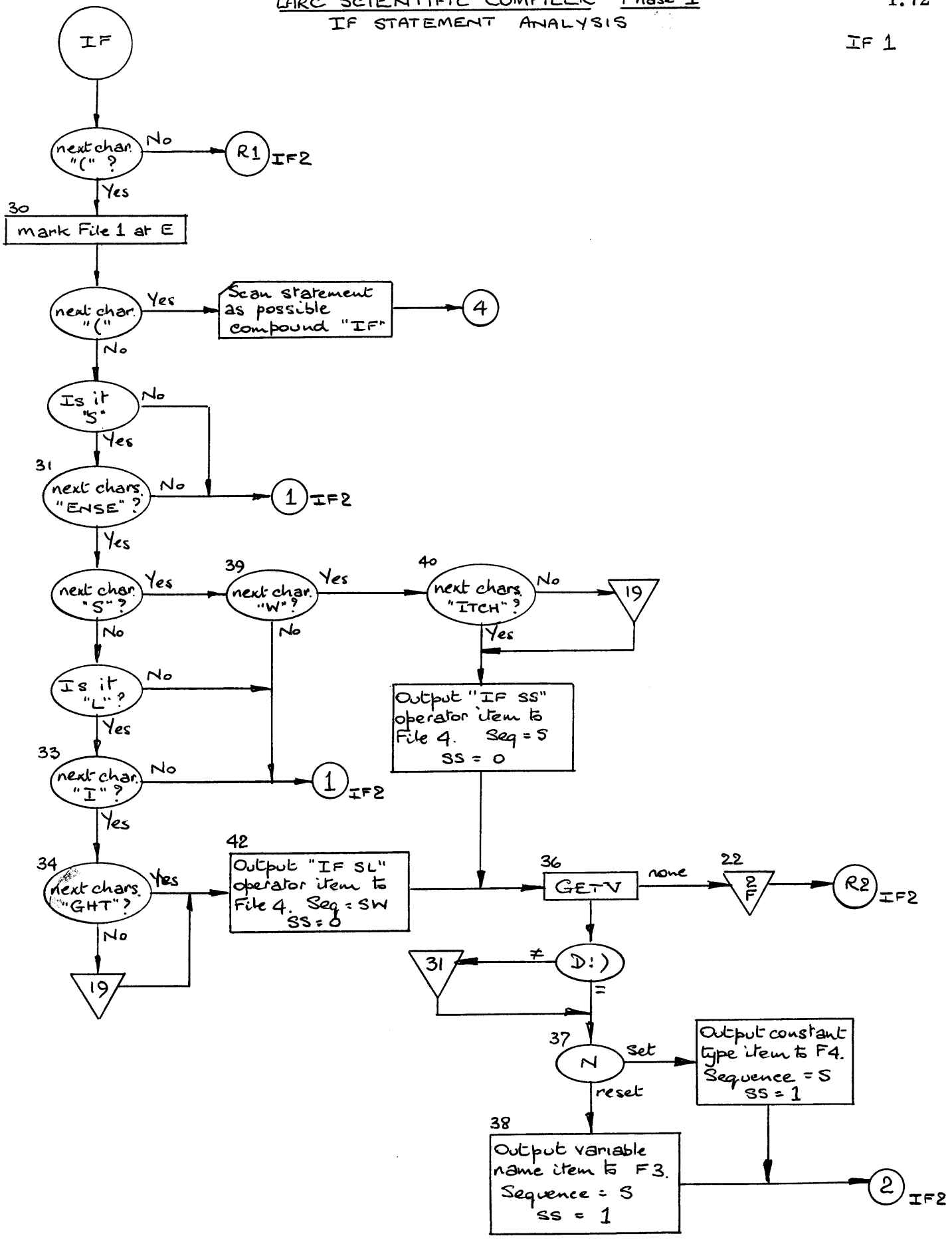


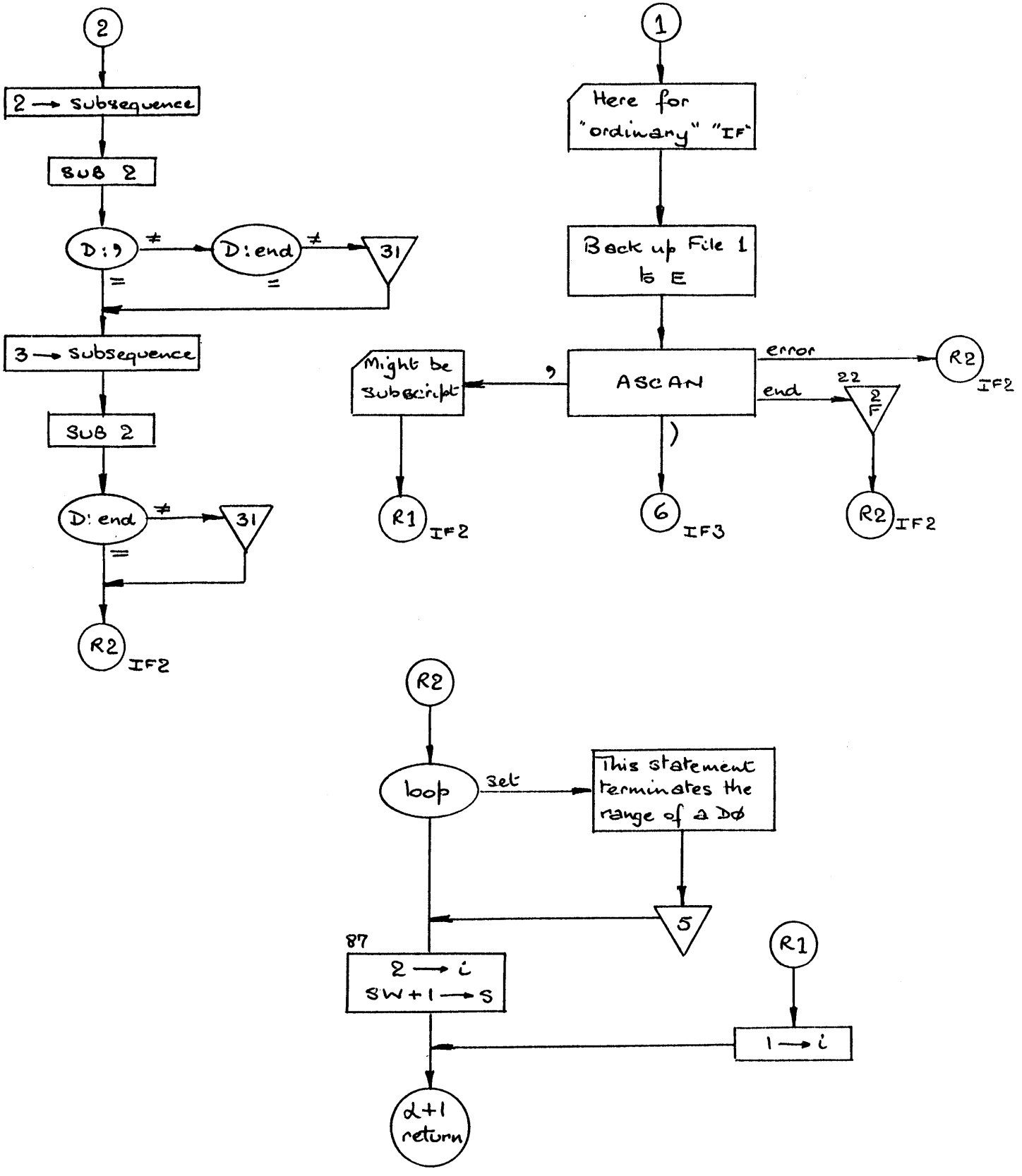

5
Here if really a 'DØ'. If this statement terminates a DØ, loop flag is set by main program, which is an error.
loop
set
5
reset
23
SW+1 → SW
P+1 → P (PLATØ)
L+1 → L
V → DØTAB1 L
S,L,P → DØTAB2 L
Output to File 3:
W1: VI
W2: SW 1 19
W3: DS L P
1 → j
6
Vj : numeric
=
≠
25
Output to F4:
W1: S j+1 45
W2: mode
W3: Vj
Output to File 3:
W1: Vj
W2: S j+1 46
W3: DS L P
24
j : 3
≠
=
j+1 → j
6
DØ4.
R2
DØ4
R1
1 → i
R2
2 → i
SW+2 → S
α+i

# LARC SCIENTIFIC COMPILER Phase I

## IF STATEMENT ANALYSIS

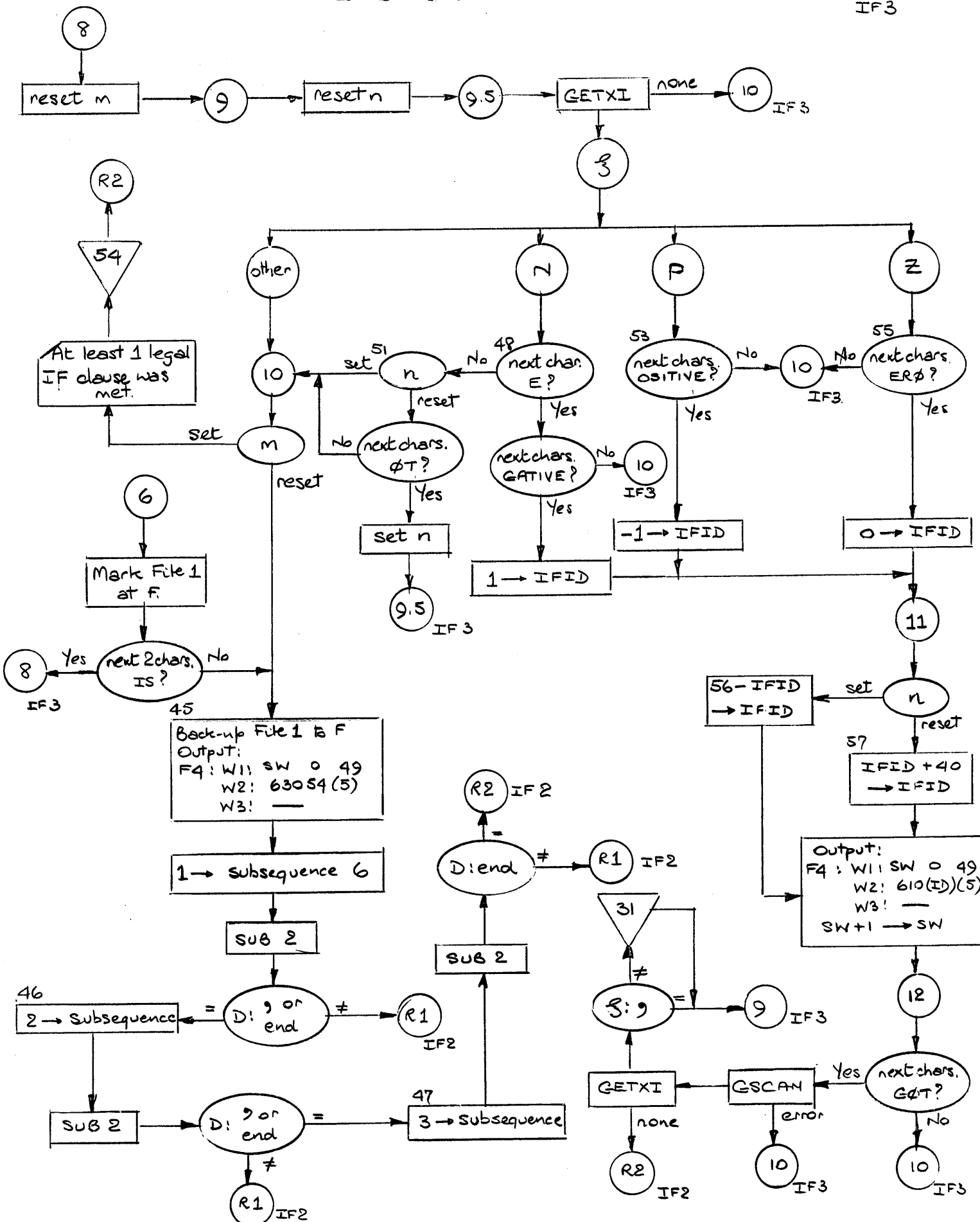
IF 1




2
2 → Subsequence
SUB 2
D:9
≠
D:end
≠
31
=
=
3 → Subsequence
SUB 2
D:end
≠
31
=
R2
IF2
1
Here for "ordinary" "IF"
Back up File 1 to E
ASCAN
error
R2
IF2
end
22
2
F
Might be Subscript
?
)
R1
IF2
6
IF3
R2
IF2
R2
loop
set
This statement terminates the range of a DØ
5
87
2 → i
SW+1 → S
R1
1 → i
d+1
return

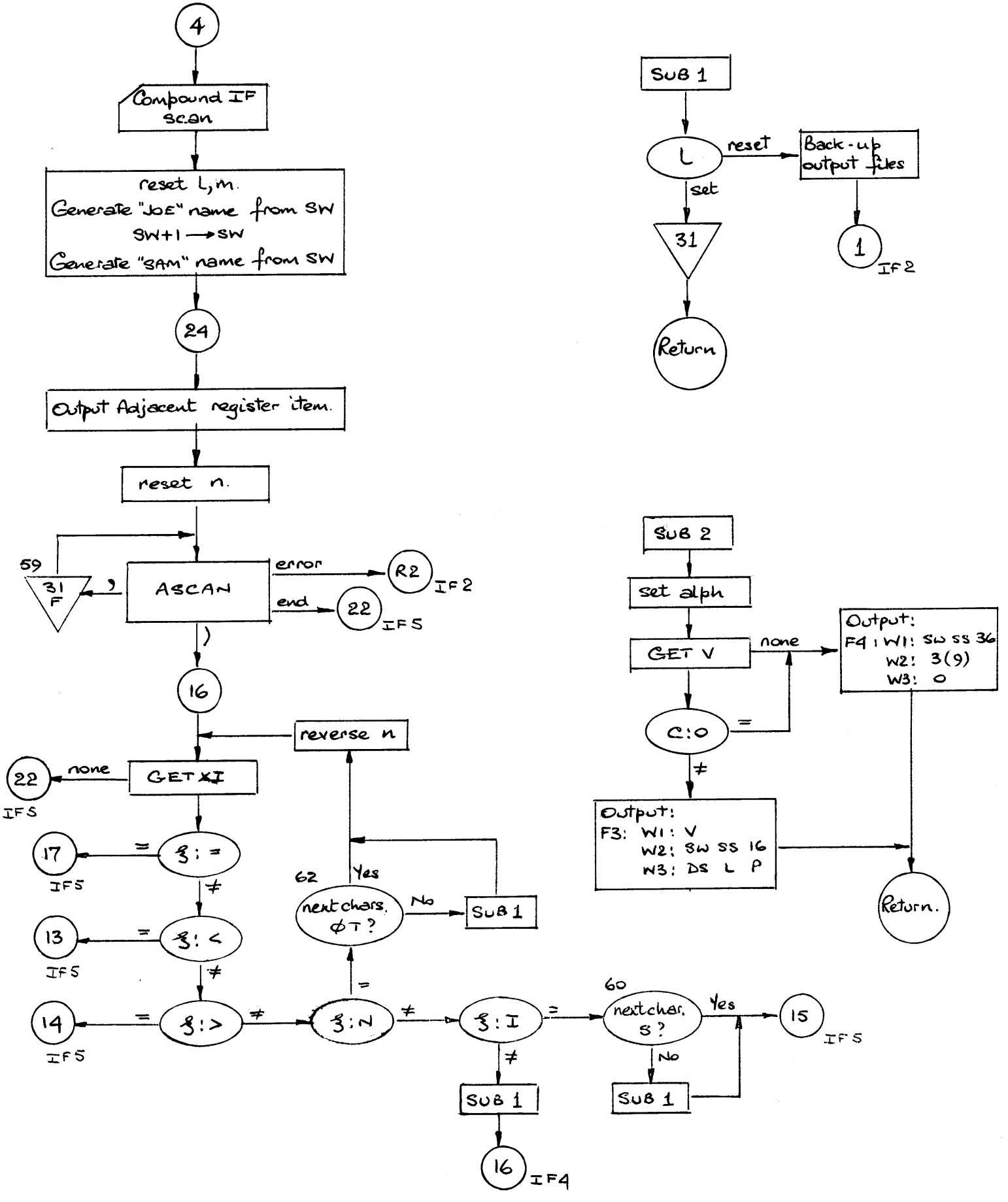
# LARC SCIENTIFIC COMPILER Phase I

## IF STATEMENT ANALYSIS.



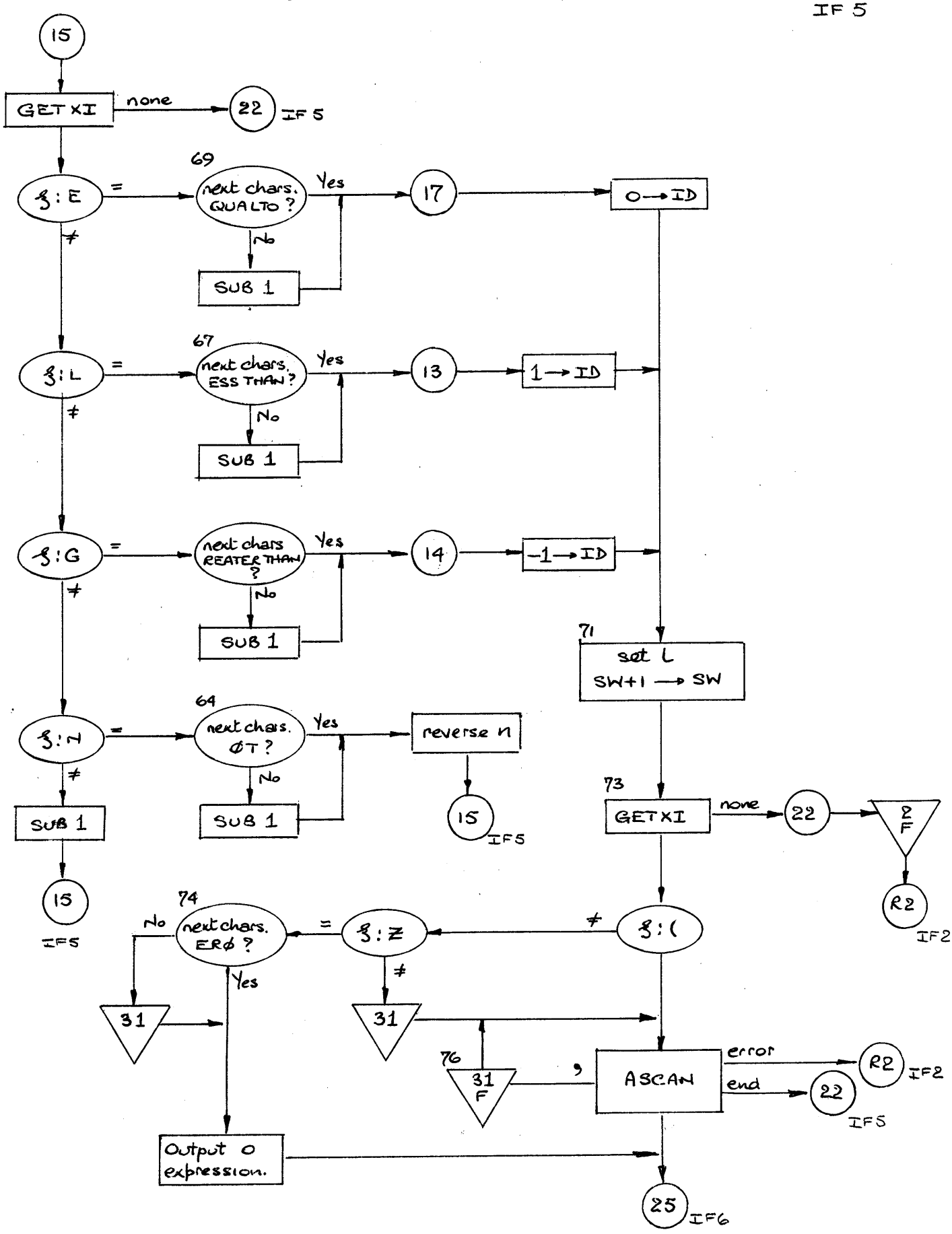
# IF STATEMENT ANALYSIS

IF4



# IF STATEMENT ANALYSIS

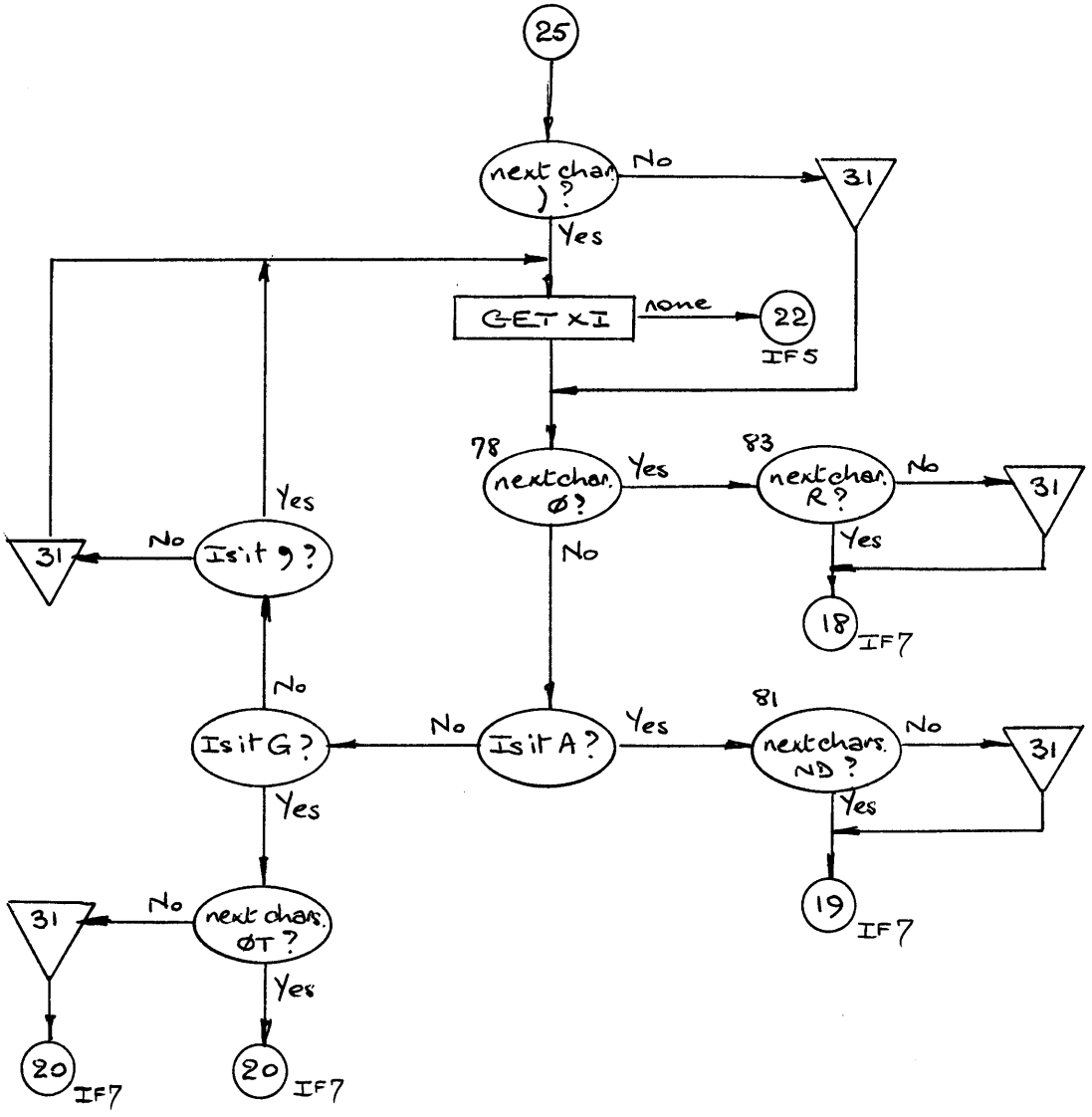




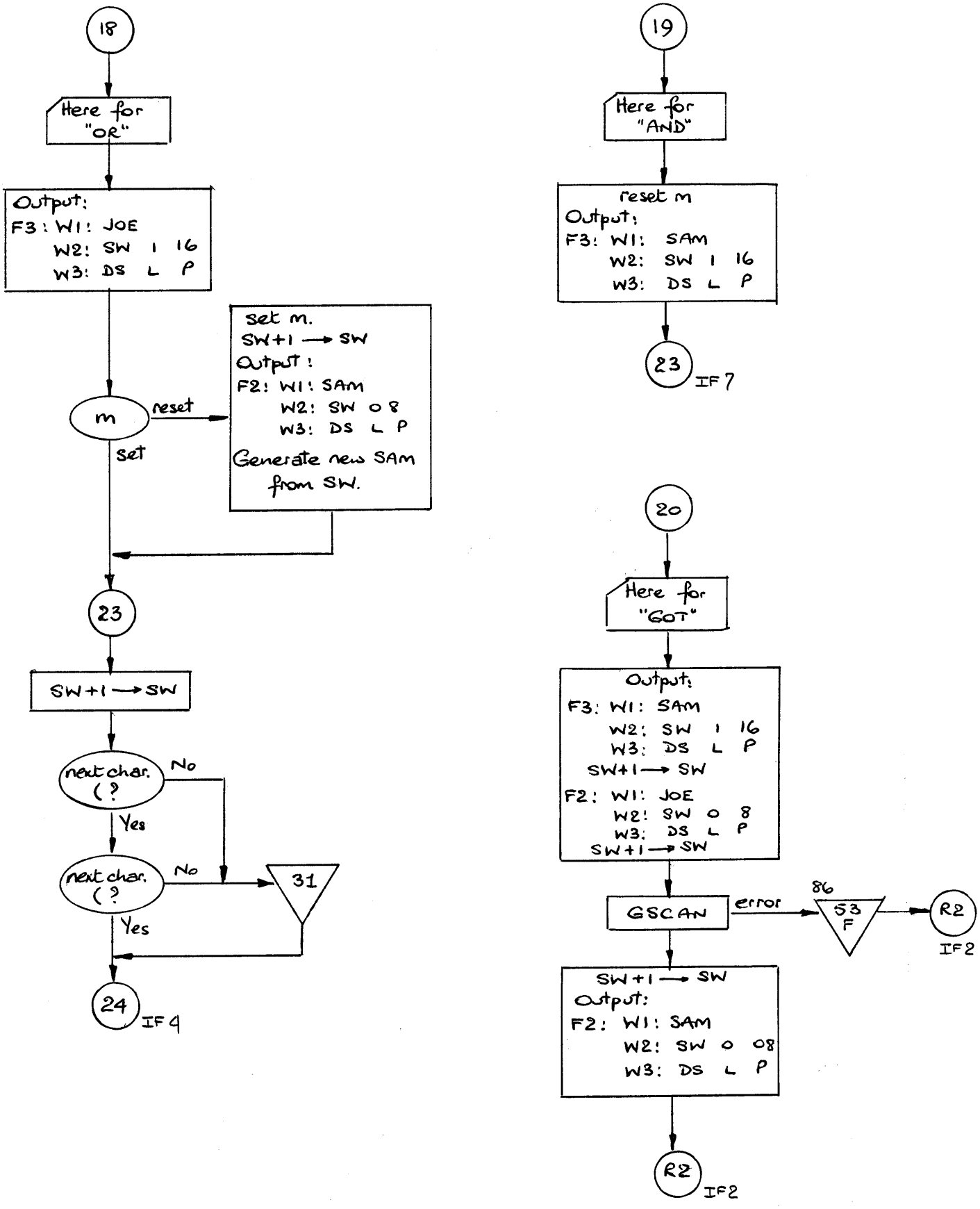
# LARC SCIENTIFIC COMPILER Phase I

## IF STATEMENT ANALYSIS

IF6

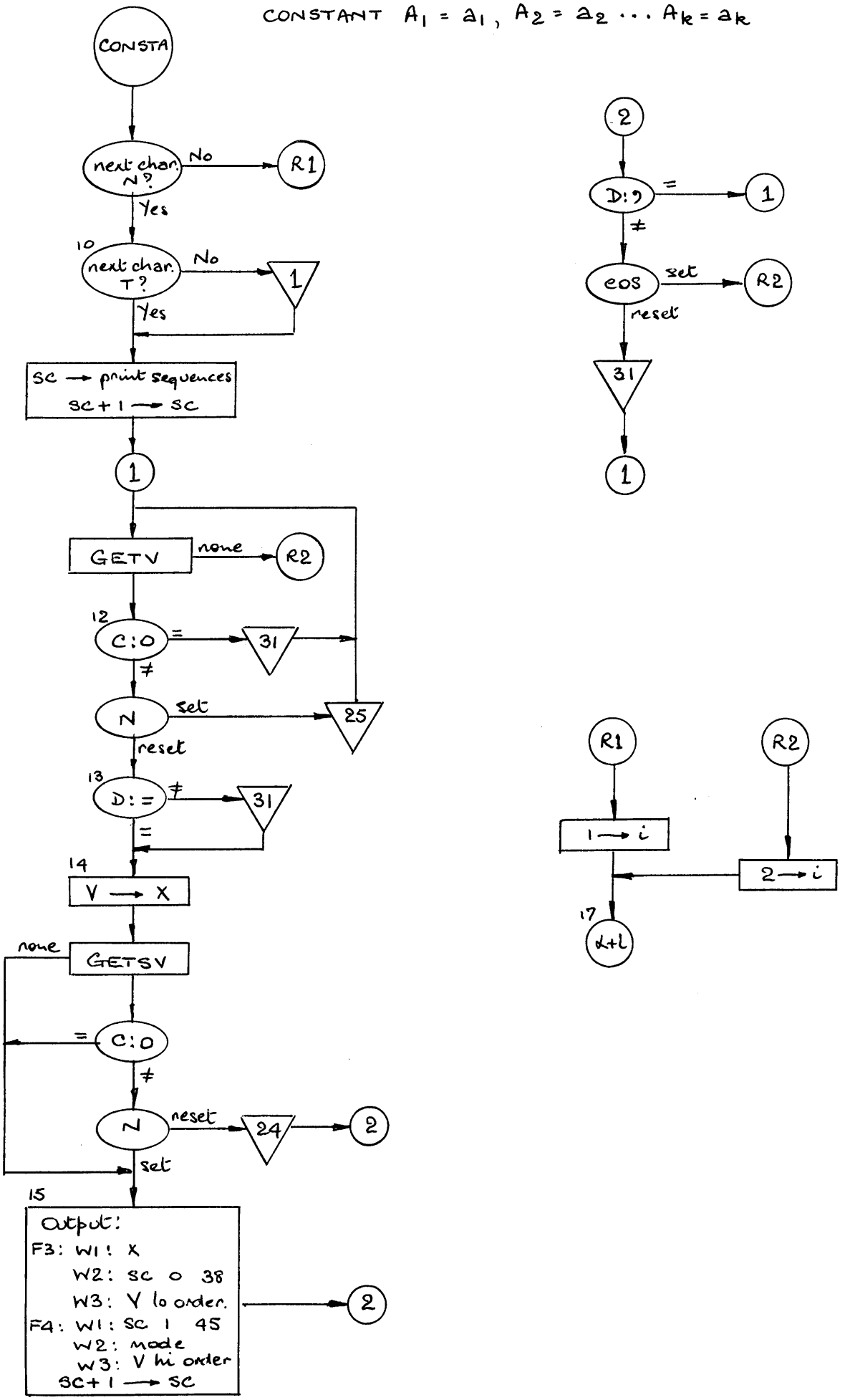


IF STATEMENT ANALYSIS IF7.



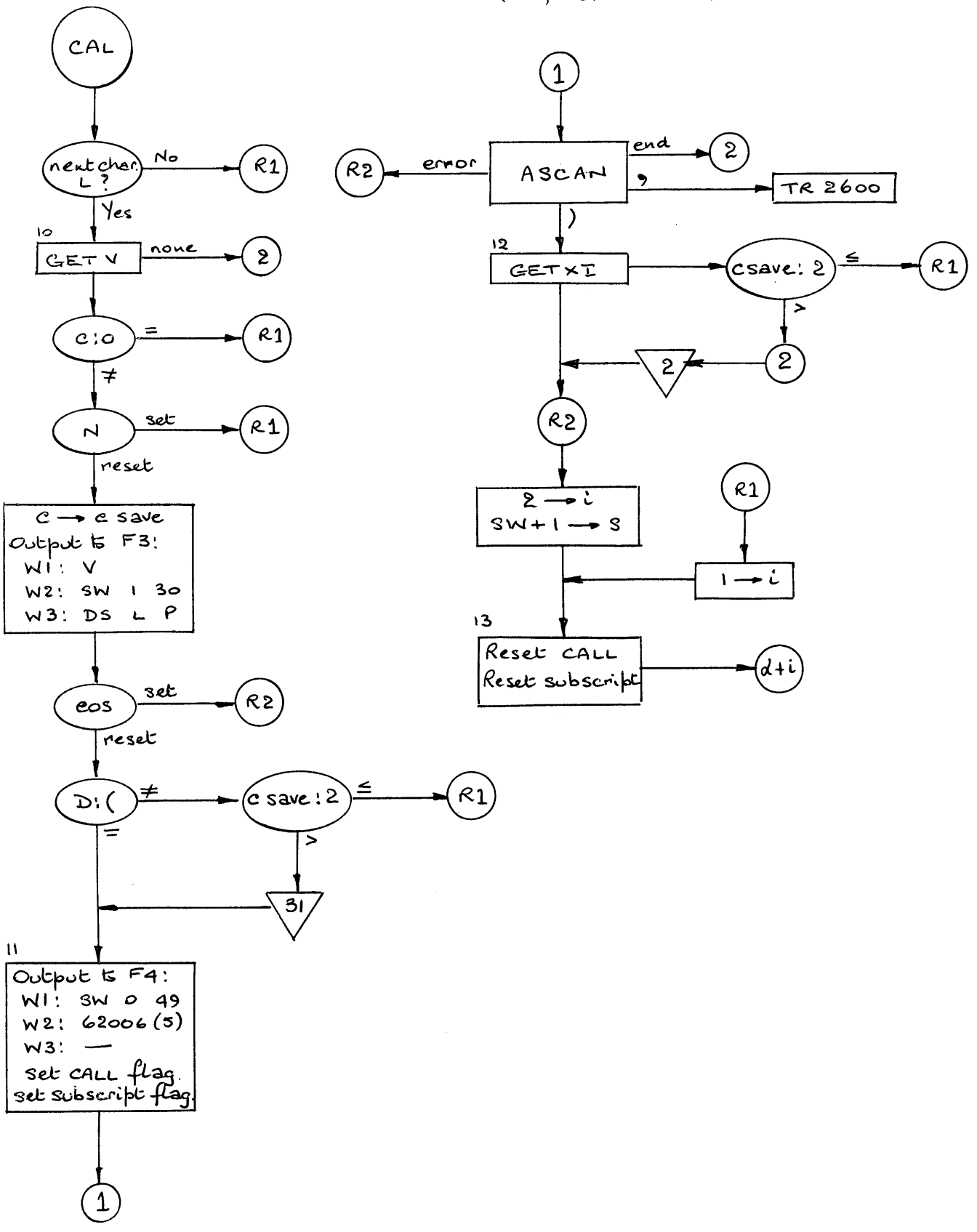
# LARC SCIENTIFIC COMPILER Phase I

CONSTANT $A_1 = a_1$, $A_2 = a_2 \cdots A_k = a_k$



# LARC SCIENTIFIC COMPILER Phase I

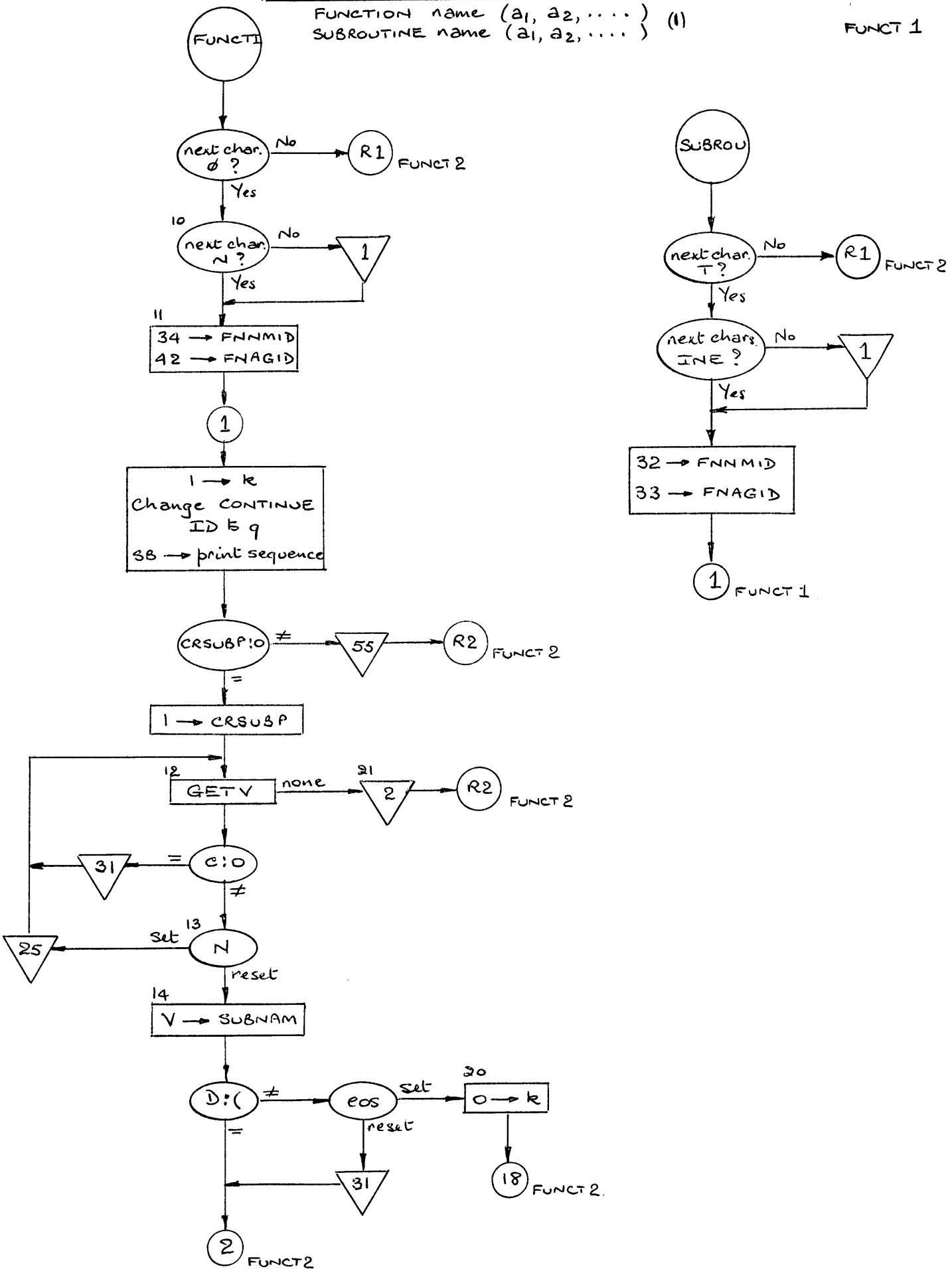
CALL $n(m_1, m_2, \ldots m_R)$

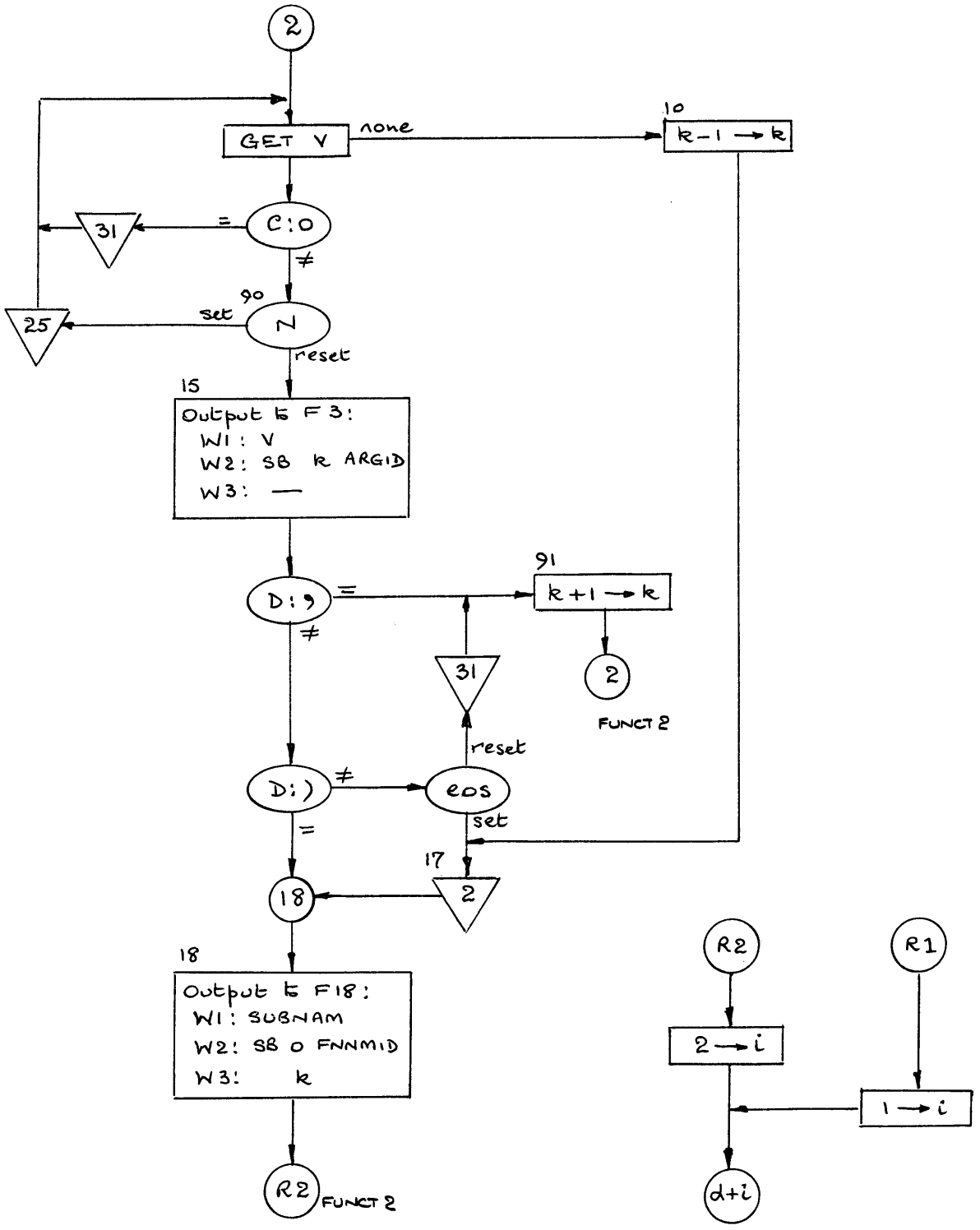


# LARC SCIENTIFIC COMPILER Phase I

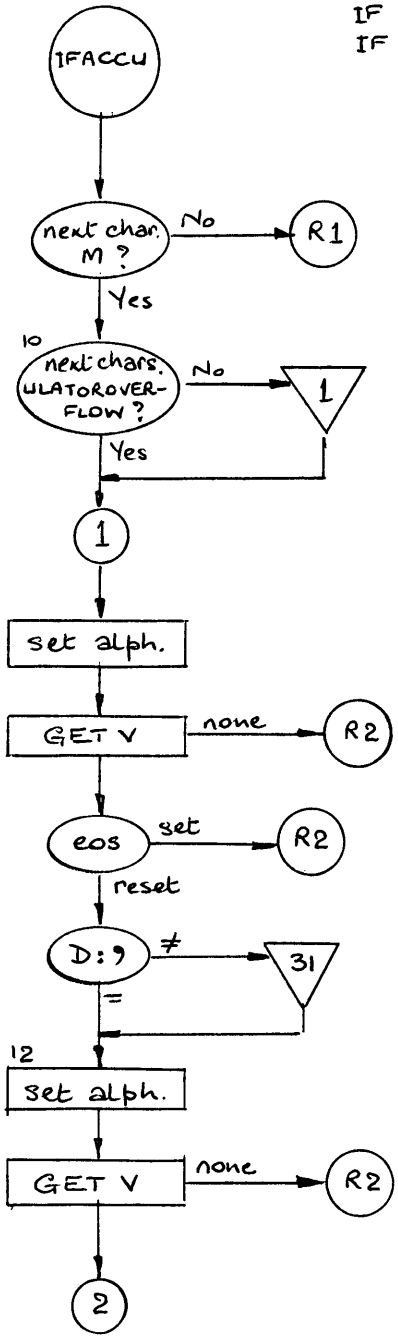
FUNCTION name $(a_1, a_2, \ldots)$ (1)
SUBROUTINE name $(a_1, a_2, \ldots)$

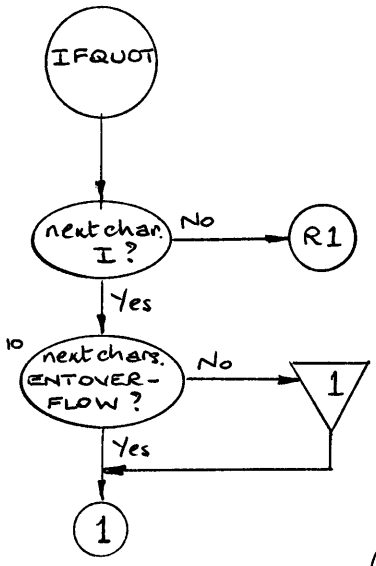
FUNCT 1

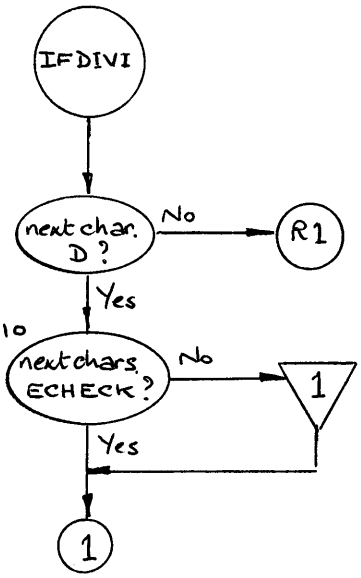


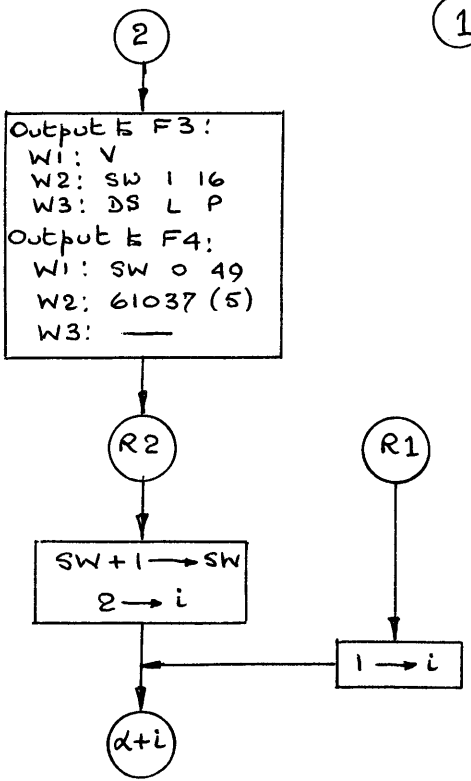

2
GET V
none
10
k-1 → k
31
=
C:0
≠
25
set
90
N
reset
15
Output to F 3:
W1 : V
W2: SB k ARGID
W3: —
91
k+1 → k
D:9
=
≠
31
2
FUNCT 2
reset
D:)
≠
eos
=
set
17
18
2
18
Output to F18:
W1: SUBNAM
W2: SB 0 FNNMID
W3: k
R2
FUNCT 2
R2
R1
2 → i
1 → i
d+i

IF ACCUMULATOR OVERFLOW $n_1, n_2$,
IF QUOTIENT OVERFLOW $n_1, n_2$
IF DIVIDE CHECK $n_1, n_2$

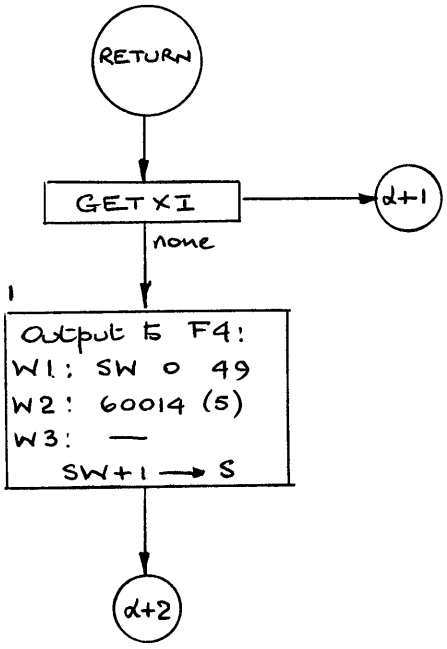


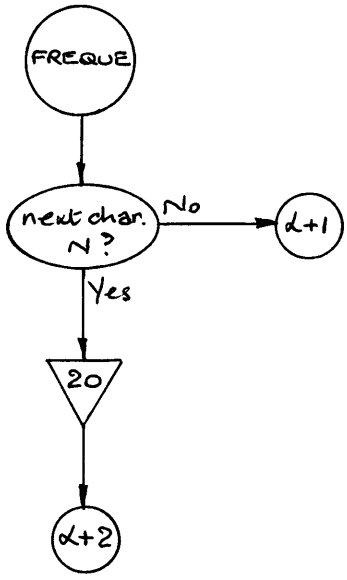



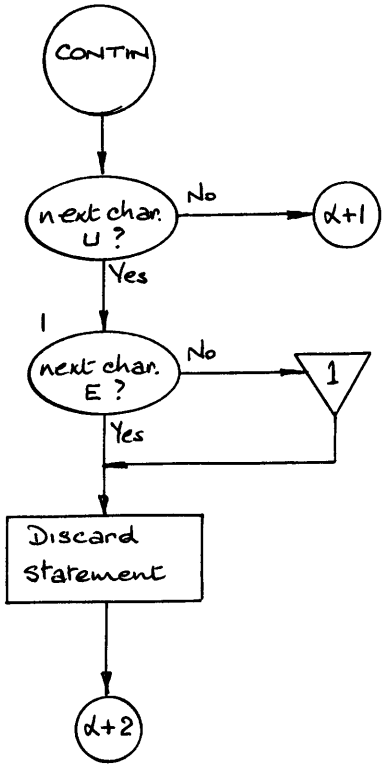



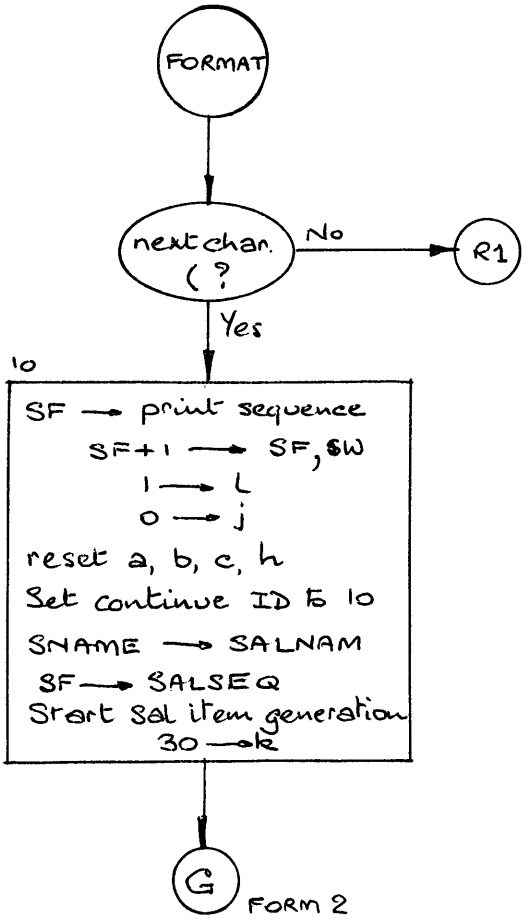



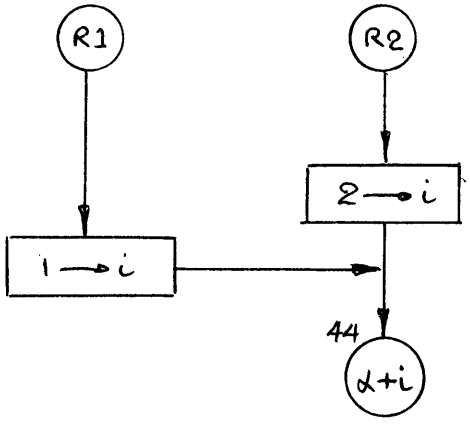
RETURN
FREQUENCY
CONTINUE

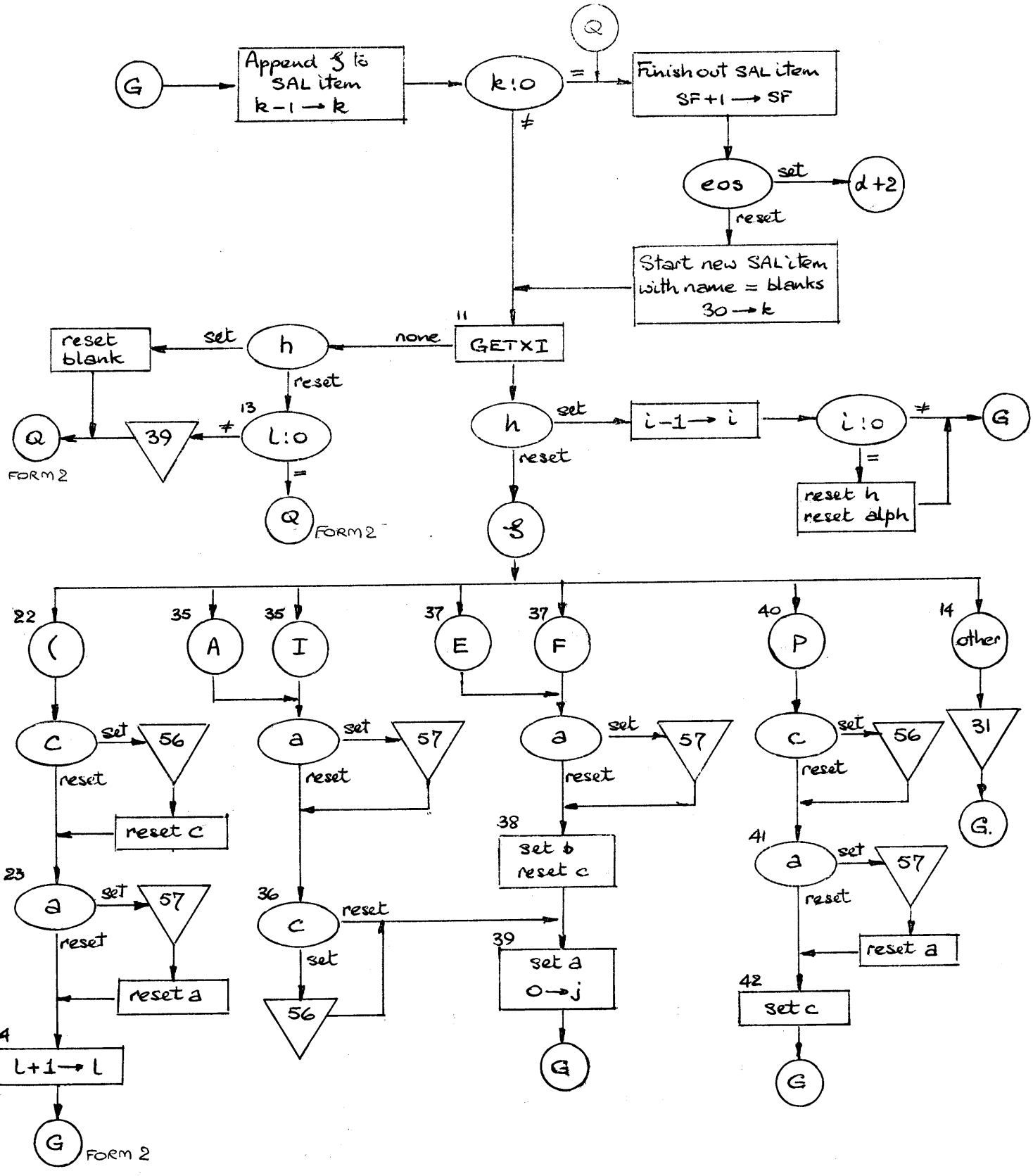


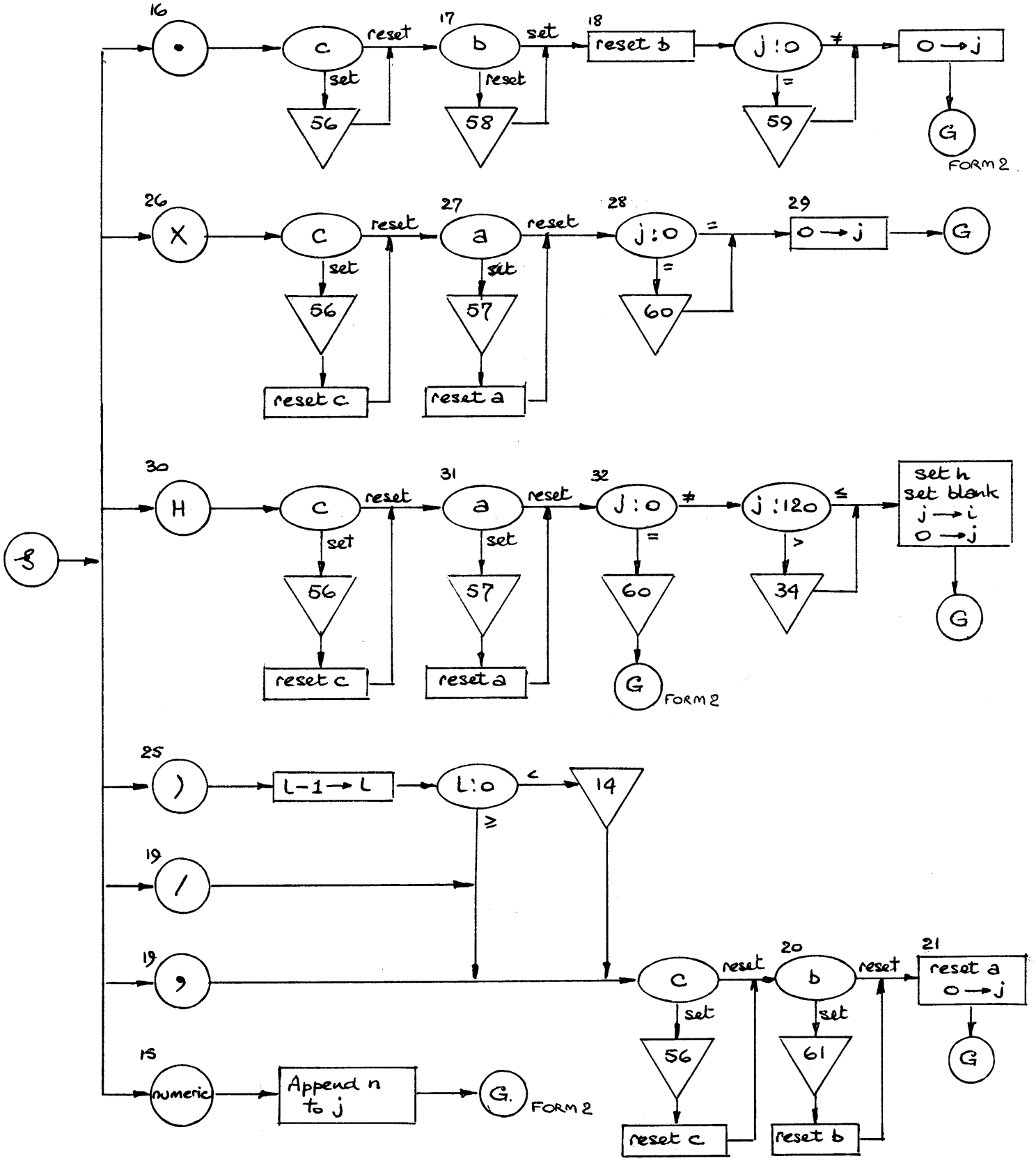



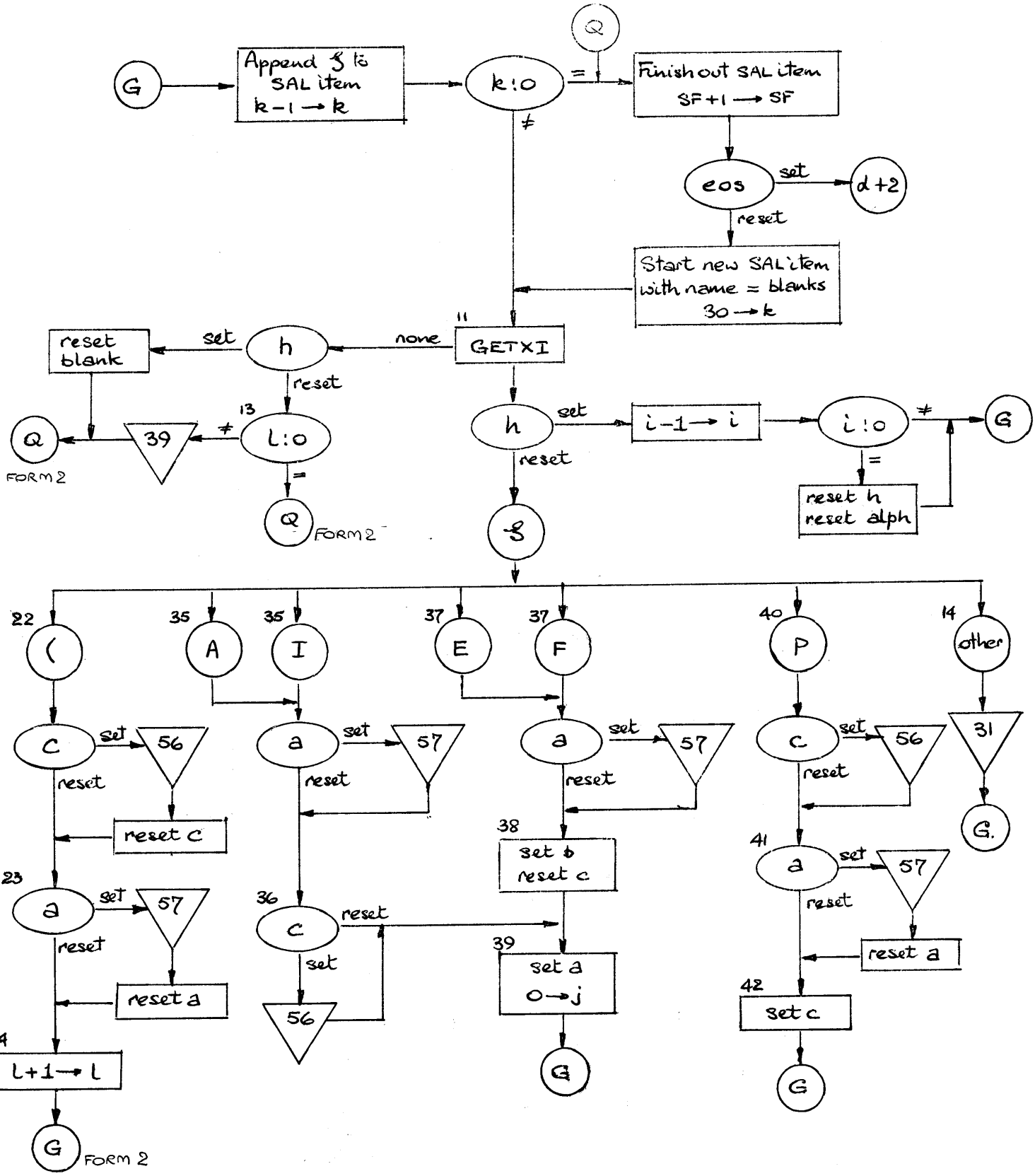




FORMAT
next char. ( ?
No
R1
Yes
10
SF → print sequence
SF+1 → SF, $W
1 → L
0 → j
reset a, b, c, h
Set continue ID to 10
SNAME → SALNAM
SF → SALSEQ
Start Sal item generation
30 → k
G
FORM 2

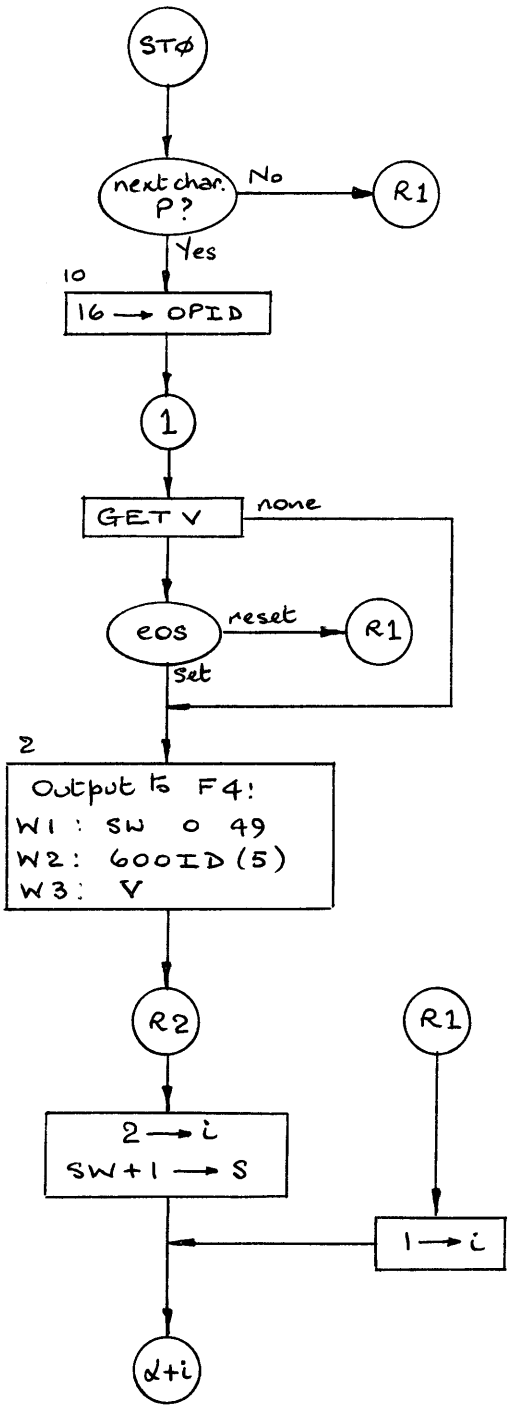

R1
R2
2 → i
1 → i
44
4+i

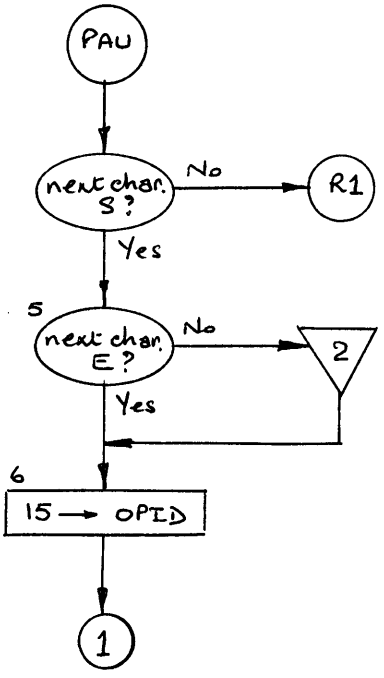

G
Append $ to SAL item
k-1 → k
k:0
=
Q
Finish out SAL item
SF+1 → SF
eos
set
d+2
reset
Start new SAL item
with name = blanks
30 → k
11
GETXI
none
h
set
reset blank
reset
13
L:0
≠
39
Q
FORM 2
=
Q FORM 2
h
set
i-1 → i
i:0
≠
G
=
reset h
reset alph
reset
$
22
(
35
A
35
I
37
E
37
F
40
P
14
other
C
set
56
reset
reset C
23
a
set
57
reset
reset a
24
L+1 → L
G
FORM 2
a
set
57
reset
36
C
reset
set
56
38
set b
reset c
39
set a
0 → j
G
a
set
57
reset
C
set
56
reset
31
G.
41
a
set
57
reset
reset a
42
set c
G


16
•
c
reset
set
56
17
b
set
reset
58
18
reset b
j : 0
≠
=
59
0 → j
G
FORM 2
26
X
c
reset
set
56
reset c
27
a
reset
set
57
reset a
28
j : 0
=
=
60
29
0 → j
G
30
H
c
reset
set
56
reset c
31
a
reset
set
57
reset a
32
j : 0
≠
=
60
G
FORM 2
j : 120
≤
>
34
set h
set blank
j → i
0 → j
G
25
)
L-1 → L
L : 0
<
≥
14
19
/
19
,
c
reset
set
56
reset c
20
b
reset
set
61
reset b
21
reset a
0 → j
G
15
numeric
Append n
to j
G.
FORM 2


G
Append $ to SAL item
k-1 → k
k:0
=
Q
Finish out SAL item
SF+1 → SF
≠
eos
set
d+2
reset
Start new SAL item
with name = blanks
30 → k
11
GETXI
none
h
set
reset blank
reset
13
l:0
≠
39
Q
FORM 2
=
Q
FORM 2
h
set
i-1 → i
i:0
≠
G
=
reset h
reset alph
reset
$
22
(
35
A
35
I
37
E
37
F
40
P
14
other
C
set
56
reset
reset C
a
set
57
reset
reset a
23
a
set
57
reset
reset a
24
l+1 → l
G
FORM 2
a
set
57
reset
38
set b
reset c
39
set a
0 → j
G
36
C
reset
set
56
C
set
56
reset
41
a
set
57
reset
reset a
42
set c
G
31
G.

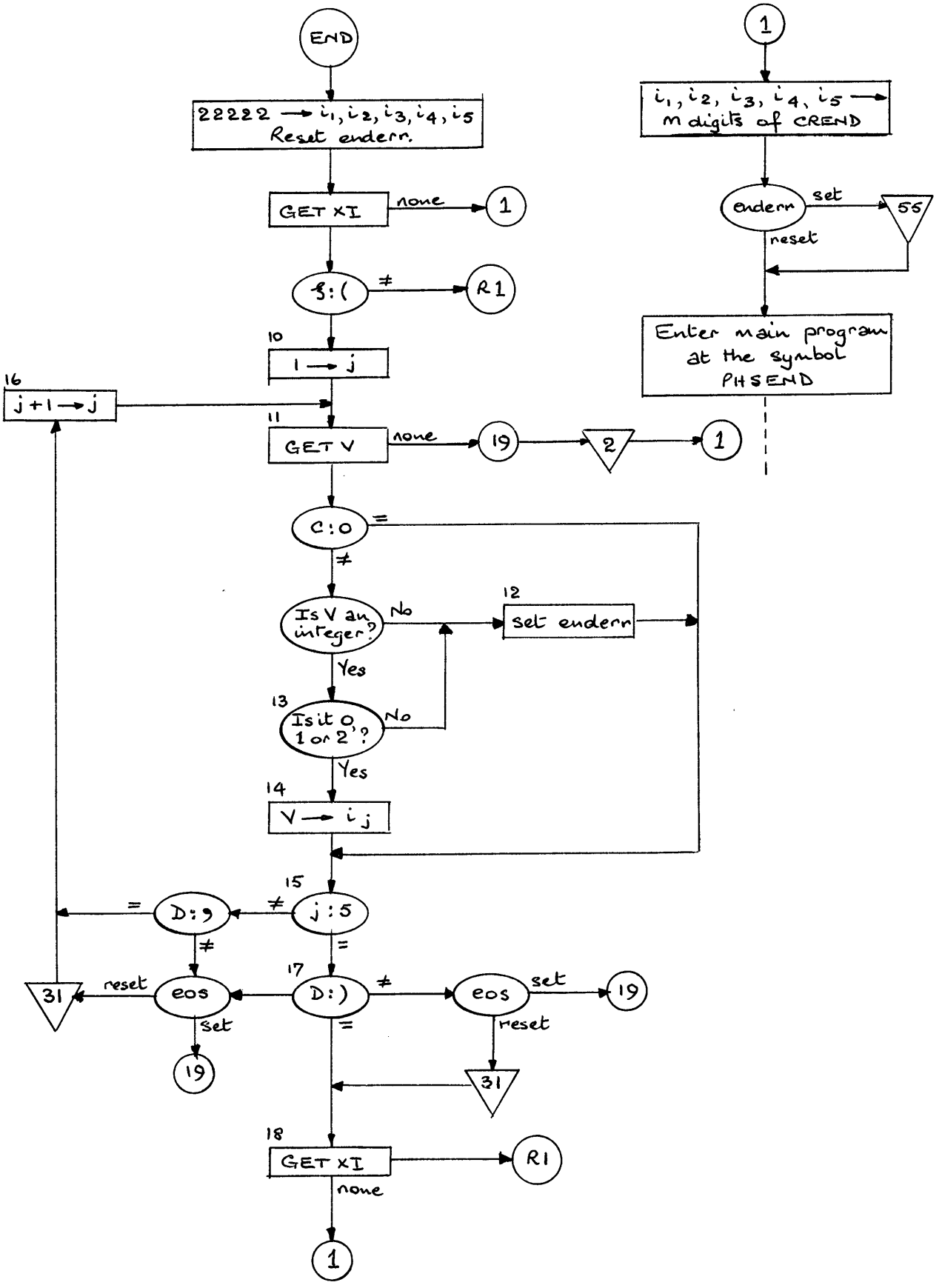
STOP n
PAUSE n



# LARC SCIENTIFIC COMPILER Phase I

END $(i_1, i_2, i_3, i_4, i_5)$
END



# PHASE II

## GENERAL DESCRIPTION

The functions of Phase II are:

1. To append information associated with a statement name, $N_i$, (continue item) to items generated from ASSIGN $N_i$ to M and GO TO M, (. . . $N_i$ . . . ) items,

2. To detect, error-mark and eliminate multiple uses of an alphanumeric word as the name of statements within an LSC source program,

3. To issue "collector" items for each built-in (open) and library (closed) subroutines, and

4. To attach pseudo-sequence numbers for most items written in File 3 by Phase II.

All items output by Phase II will be written in File 3, except error items which go to File 92.

A group of items to be analyzed in concert by Phase II consists of all items with the same name; there should be one continue item in each group. When all groups in File 2 have been processed, the function collector items are filed, library subroutine names are given dictionary references, and return is made to the control program.

## STATEMENT NAME (CONTINUE) ITEMS

A group which contains more than one continue item will generate only one continue item for File 3 and one error item for each of the other continue items. The continue item sent to File 3 will be:

a. The first item generated from a named executable statement (ID = 08), or

b. If none of these, the first item generated from either a named non-executable statement (ID = 09), or from a named SAL instruction (ID = 11), or

c. The first item generated from a FORMAT statement (ID = 10).

The sequence number of (08, 09, and 11) items in File 3 will be (CRS-4); the sequence number of (10) items will be (CRS-3). The original sequence number of these items has become the subsequence. (CRS) is the initial value sequence counter used for executable statements.

ASSIGN STATEMENTS

Each assign item (ID = 12) in File 2 generates three items in File 3 and an error item if a continue item is not within the group. The first of the items in File 3 is merely the assign item carried forward. Two assigned items (ID=13) are generated for each assign item. The first item has sequence (CRS-2) and sub-sequence equal to the sequence of the continue item which was assigned; the second item has the same sequence as in the (12) item. Each of these (13) items will be in an assigned variable group in Phase III and will contain in the third word the DO sequence, level, and plateau numbers of the appropriate continue item.

ASSIGNED GO TO STATEMENTS

Executable statement names which appear in "assigned go to" lists and in assign statements generate items (ID=18) which appear in assigned variable groups in Phase III. These items contain the DO sequence, level and plateau numbers of the appropriate continue item in word 3. An error item is generated if the listed assignment does not have a corresponding assign statement or continue item.

LIBRARY AND BUILT-IN FUNCTIONS

One Item (ID=44) is sent to File 3 for each possible built-in and library function which

may have been referenced in an LSC source program. The sequence number for these items is (CRS-1); the third word contains an embryonic mode word for use by later phases. This mode word contains the (a, b, c) digits and for built-in functions an assigned operator for Phase VII in the (ffff) digits.

PHASE II ITEM FORMATS

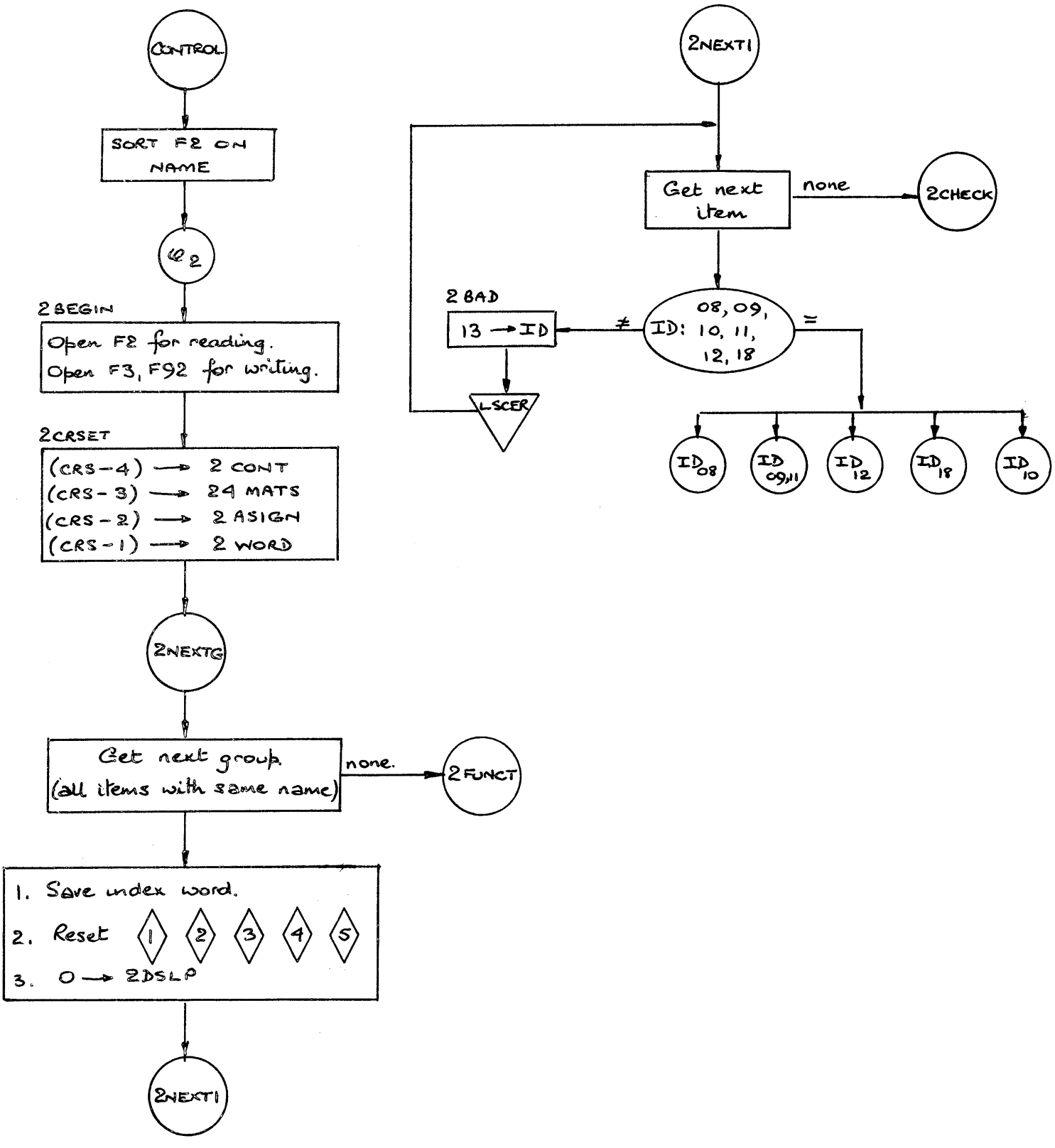
| From Phase I | | To Phase III | |
|---|---|---|---|
| 1. CONTINUE - executable statement | | | |
| W1 | NAME | W1 | NAME |
| W2 | S . O . 08 | W2 | (CRS-4) . S . 08 |
| W3 | DS , L , P | W3 | DS , L , P |
| ID = 09,10,11 are similar | | | |
| 2. ASSIGN NAME = TO M | | | |
| W1 | NAME | W1 | NAME |
| W2 | S . 1 . 12 | W2 | S . 1 . 12 |
| W3 | M | W3 | M |
| | | W1 | M |
| | | W2 | (CRS-2) . $S_c$ . 13 |
| $S_c$ = sequence of continue item | | W3 | DS , L , P |
| DS , L , P are from the continue item for NAME | | W1 | M |
| | | W2 | A . $S_c$ . 13 |
| | | W3 | DS , L , P |
| 3. GO TO M, (. . . $N_j$ . . . ) | | | |
| W1 | $N_j$ | W1 | M |
| W2 | S . j+1 . 18 | W2 | S . $S_c$ . 18 |
| W3 | M | W3 | DS , L , P |

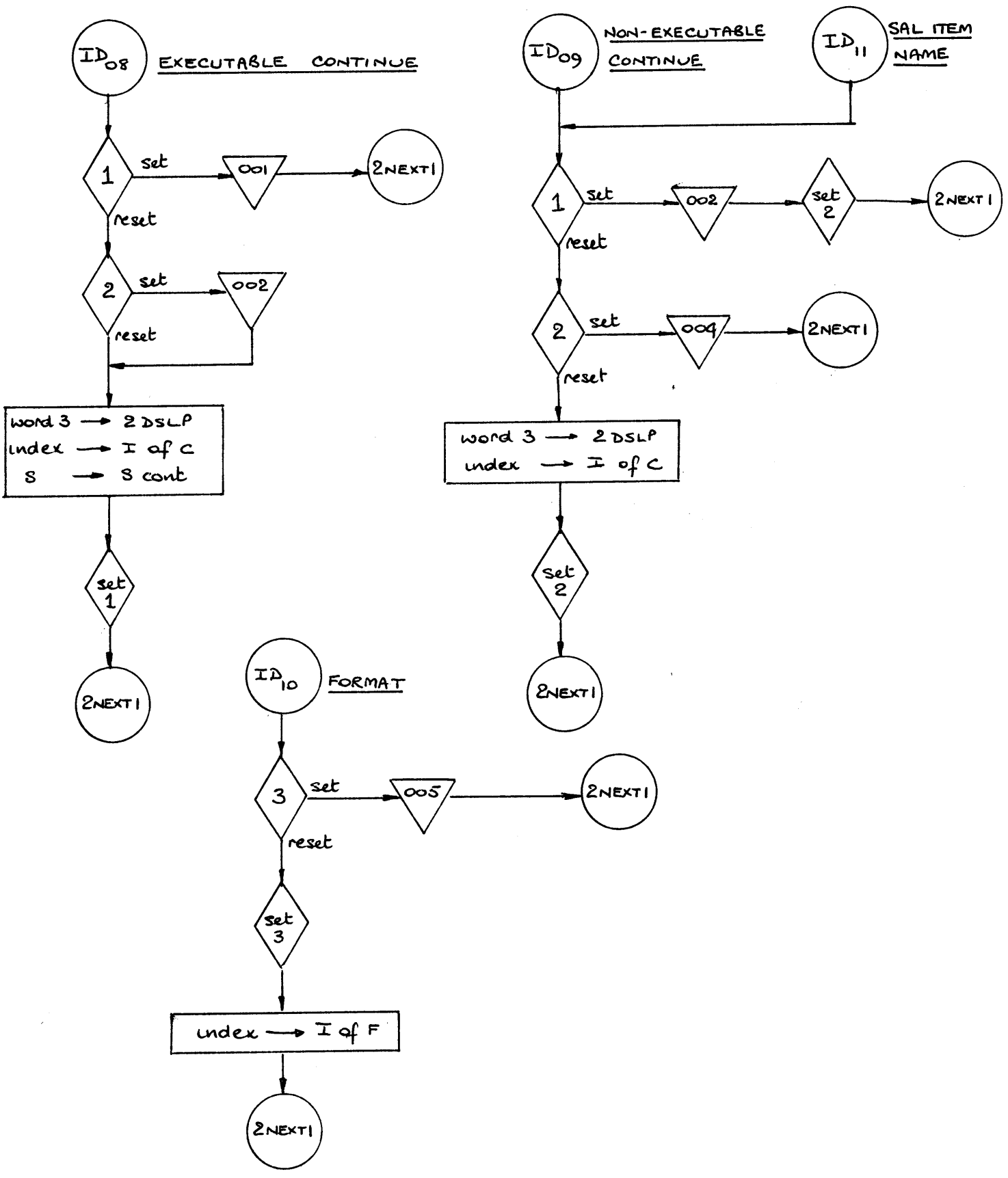
BUILT-IN FUNCTION ITEMS (to File 3)

| W1 | | W2 | W3 |
|---|---|---|---|
| ABS | (CRS-1) | 0000144 | 707000000012 |
| DIM | | 0000244 | 707000000085 |
| FLOAT | | 0000344 | 707000000009 |
| INT | | 0000444 | 707000000010 |
| MAXO | | 0000544 | 707000000086 |
| MAX1 | | 0000644 | 707000000086 |
| MINO | | 0000744 | 707000000087 |
| MIN1 | | 0000844 | 707000000087 |
| MOD | | 0000944 | 707000000083 |
| SIGN | | 0001044 | 707000000084 |
| XABS | | 0001144 | 707000000011 |
| XDIM | | 0001244 | 707000000095 |
| XFIX | | 0001344 | 707000000010 |
| XINT | | 0001444 | 707000000010 |
| XMAXO | | 0001544 | 707000000096 |
| XMAX1 | | 0001644 | 707000000096 |
| XMINO | | 0001744 | 707000000097 |
| XMIN1 | | 0001844 | 707000000097 |
| XMOD | | 0001944 | 707000000093 |
| XSIGN | | 0002044 | 707000000094 |
| MAX | | 0002144 | 707000000086 |
| MIN | | 0002244 | 707000000087 |
| XMAX | | 0002344 | 707000000096 |
| XMIN | (CRS-1) | 0022444 | 707000000097 |

LIBRARY FUNCTION ITEMS (to File 3)

| W1 | W2 | W3 |
|---|---|---|
| ACOS | (CRS-1) 0000044 | 82400000FFFF |
| ASIN | | |
| ATAN | | |
| CBRT | | |
| COS | | |
| COSH | | |
| COT | | |
| CSC | | |
| EXP | | |
| LOG | | |
| LOG10 | | |
| SEC | | |
| SIN | | |
| SINH | | |
| SQRT | | |
| TAN | | |
| TANH | (CRS-1) 0000044 | 82400000FFFF |


CONTROL
SORT F2 ON NAME
@2
2BEGIN
Open F2 for reading.
Open F3, F92 for writing.
2CRSET
(CRS-4) → 2 CONT
(CRS-3) → 24 MATS
(CRS-2) → 2 ASIGN
(CRS-1) → 2 WORD
2NEXTG
Get next group (all items with same name)
none.
2FUNCT
1. Save index word.
2. Reset 1 2 3 4 5
3. 0 → 2DSLP
2NEXTI
2NEXTI
Get next item
none
2CHECK
2BAD
13 → ID
≠
08, 09, ID: 10, 11, 12, 18
=
LSCER
ID08
ID09,11
ID12
ID18
ID10

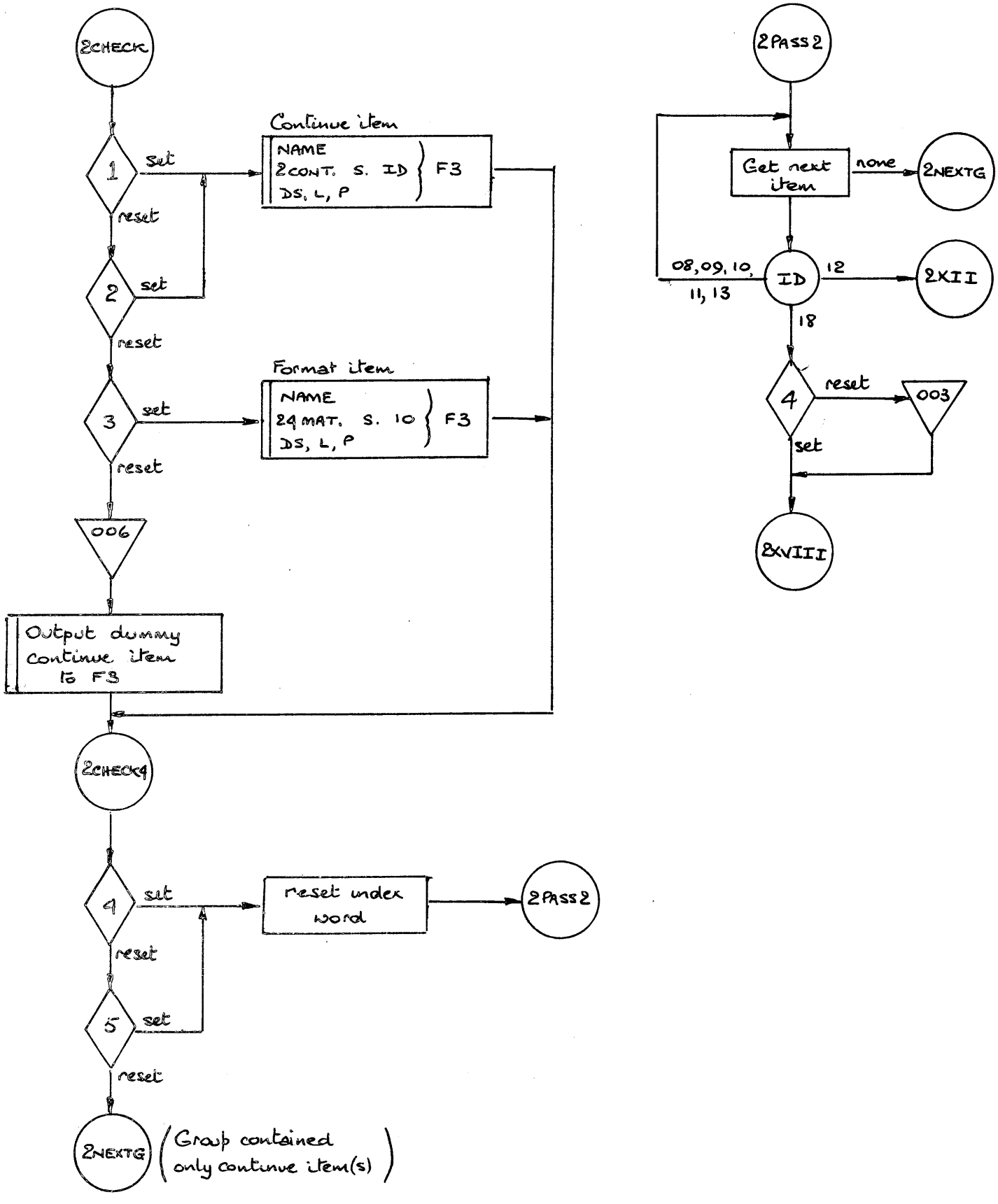


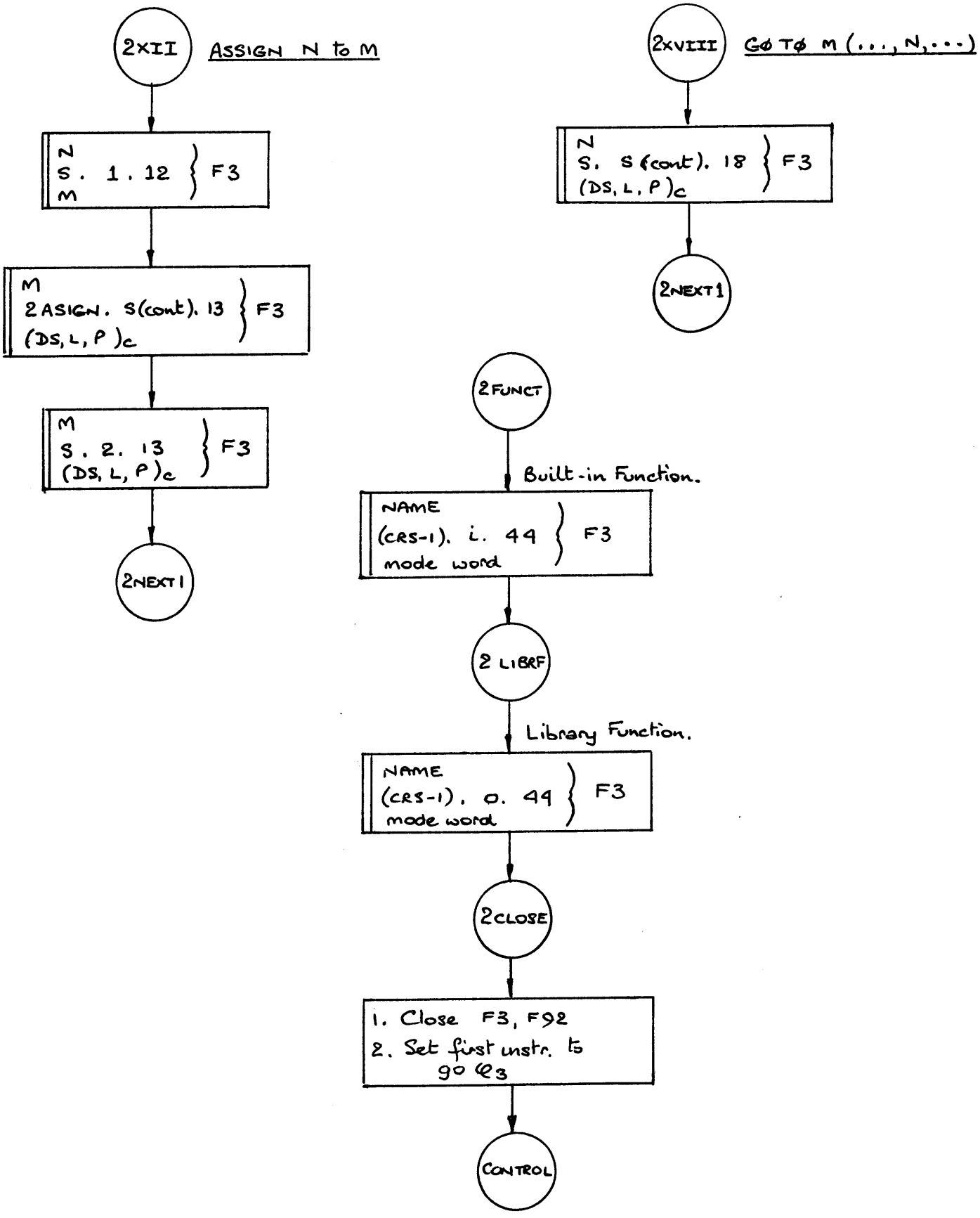

001 Multiple executable statements with same name.

002 Executable and non-executable statement have same name.

004 Multiple non-executable statements and/or SAL items have same name.

005 Multiple Format statements have same name.


2CHECK
1
set
reset
Continue item
NAME
2CONT. S. ID
DS, L, P
F3
2
set
reset
3
set
reset
Format item
NAME
2q MAT. S. 10
DS, L, P
F3
006
Output dummy continue item to F3
2CHECKq
q
set
reset
reset under word
2PASS2
5
set
reset
2NEXTG
Group contained only continue item(s)
2PASS2
Get next item
none
2NEXTG
08, 09, 10, 11, 13
ID
12
2XII
18
4
reset
003
set
8XVIII
003
No assign statement for listed assignment
006
No continue item for assign or assigned go to statement.


2XII
ASSIGN N to M
N
S. 1. 12
M
F3
M
2ASIGN. S(cont). 13
(DS, L, P)c
F3
M
S. 2. 13
(DS, L, P)c
F3
2NEXT1
2XVIII
GØ TØ M(..., N, ...)
N
S. S(cont). 18
(DS, L, P)c
F3
2NEXT1
2FUNCT
Built-in Function.
NAME
(CRS-1). i. 44
mode word
F3
2 LIBRF
Library Function.
NAME
(CRS-1). 0. 44
mode word
F3
2CLOSE
1. Close F3, F92
2. Set first instr. to
go @3
CONTROL

# PHASE III

## GENERAL DESCRIPTION

The function of Phase III is to analyze the use of all names in an LSC source program. This analysis involves the generation of mode words for functions and variables, the association of statement names with references to statements, and the detailed cross checking of the use of names. The important function of error detection in the compiler is also accomplished in Phase III through this process of cross referencing.

Before Phase III begins operation, File 3, which has been generated by Phases I and II, is sorted on the name of the item and the assigned sequence number. This sort makes it possible to avoid forming and searching tables to obtain information about a name. File 3 items have three words of the following form:

| | |
|---|---|
| Word 1: | Name (6 alphanumeric characters) |
| Word 2: | Digits 1-10: assigned sequence number |
| | Digits 11-12: item type identification (ID) |
| Word 3: | Special information |

Phase III begins its operation by requesting a group from the general file maintenance routine (FILE). A group of items in Phase III contains all items with the same name arranged on monotonically increasing order according to sequence numbers assigned by Phase I and Phase II. This ordering brings defining items to the top of a group enabling Phase III to complete mode definition by skimming off the top of a group rather than requiring a scan of the complete group, necessary if defining items had been homogenized with reference items.

The first scan analyzes items for determination of mode word information. Items are examined until the storage assignment, arithmetic type, and class of a group have been determined. These characteristics are translated into a, b, and c digits of the mode word; this word is completed; and the second scan, which produces items for File 4

is initiated. After all name-groups have been processed by Phase III, return is made to the control program.

DEFINITION HIERARCHY

The LSC set of sequence counters described in Phase I and Phase II is used to arbitrate errors arising from multiple definition of LSC names. Phase III will classify a group as belonging in one of the 16 possible categories as dictated by the first item in a group with ID digits listed below. Subsequent definitions and improper references will be considered offenses against the first definition.

| SEQ. | GROUP CLASSIFICATION | ID DIGITS |
|---|---|---|
| SB | FUNCTION | 34 |
| SB | SUBROUTINE | 32 |
| ST | PARAMETER | 01 |
| SD | DIMENSIONED VARIABLE | 02 |
| SC | CONSTANT | 38 |
| SA | ARITHMETIC FUNCTION | 23 |
| SA | DUMMY ARGUMENT OF ARITHMETIC FUNCTION | 24, note 2 |
| S-4; S | STATEMENT NAME | 08,09,11; 16; 35 |
| S-3; S | FORMAT NAME | 10; 15 |
| S-2 | ASSIGNED VARIABLE | 13 |
| S-1 | BUILT-IN FUNCTION | 44 |
| S-1 | LIBRARY FUNCTION | 44 |
| S | CALL | 30 |
| S | IMPLICIT FUNCTION | 47,25 |
| S | GENERAL VARIABLE | note 1 |
| S | INDUCTION VARIABLE | 19, note 2 |

Notes:

1. A group is classified as a general variable group if its first item with sequence (S) has valid ID digits different from those in the preceding table.

2. These groups are sub-groups and may appear within groups with different classifications.

Arguments of a FUNCTION (42) and SUBROUTINE (33) are considered as reference items although they are assigned sequence (SB).

COMMON (05), EQUIVALENCE (04) and mode specification (06, 07) items are adjectival items; that is they restrict, but do not themselves classify a group into one of the 16 categories.

PRONOUN GROUPS

If one item in the group has the ID of a PARAMETER statement, then the value of the parparameter is attached to all other items in the group. These items (except the PARAMETER item) are then filed in File 4 as "numeric" items. If the group has more than one PARAMETER item, an error is recorded and the first parameter given is used. Any item in the group which would not have its value supplied by a parameter generates an error (e.g., GO TO N may not have a parameter given for N). If a group contains an item with the ID of a CONSTANT statement, the group is treated in a similar manner.

STATEMENT NAME GROUP

If the group contains a "continue" item (a statement name), then all other names in the group must be references to that name. (GO TO NAME, etc.) If a group contains only a continue item, nothing is output from that group, as a result only those continue items which are referenced will be treated as "entry points" when computation of common subexpressions are eliminated in Phase VI.

Reference items (output to File 4) contain the sequence of the continue item in the third word.

In addition to reference items, special control items may be generated; these items are SUCCESSOR and TO and FROM items.

SUCCESSOR ITEMS

A successor item is generated whenever control leaves or enters a block. Phase III generates successor items when four types of items are encountered:

1. A listed assignment in an ASSIGNED GO TO statement,

2. SAL EXIT item,

3. A statement name used as an argument of a CALL,

4. A GO TO item which occurs within an assigned variable group, but the possible assignments are not listed. In this case, a successor is generated for each possible assignment.

The successor item contains the sequence numbers of the continue item (SCONT) and the reference item (SREF). These numbers are obtained in the following manner for the case listed above:

| | SCONT | SREF |
|---|---|---|
| 1. | Subsequence of an 18 item | Sequence of the preceding 17 item |
| 2. | Sequence of an 08 item | Sequence of a 35 item |
| 3. | Sequence of an 08 item | Sequence of a 31 item |
| 4. | Subsequence of each 13 item within the group. | Sequence of a 16 item |

The format of this successor item is contained in the item format part of the Phase III section. (A full description of successor items is in the Appendix).

TO AND FROM ITEMS

Possible transfers of control into, within, and from DO loops generate TO and FROM items. The generation of these items is subject to the following tests:

1. If $(DS,P)_C = (DS,P)_R$ , neither item is output.

2. If $(DS,P)_C \neq (DS,P)_R$ , and if $(DS)_c \neq 0$ a TO item is output.

3. If $(DS,P)_C \neq (DS,P)_R$ , and if $(DS)_R \neq 0$ a FROM item is output.

DS is the DO sequence number of the current loop
P is a count of the number of preceding DO statements
sub C denotes a continue item
sub R denotes a reference item

The DS and P numbers of continue and reference items are contained in the third word of these items. The occurrence of items for which TO/FROM tests are made is identical to the four occurrences for which successor items are generated; in addition all ordinary references to statement names (ID = 16) which are not in an assigned variable group are tested for TO/FROM generation. These TO and FROM items are used by Phase IV for DO loop analysis.

The format of TO and FROM items is contained in the item format part of the Phase III section.

PSEUDO-CONTINUE ITEM

The pseudo-continue item (ID = 13) generated in Phase II, will be processed according to the type of reference made to it as a statement name. However, in the same group, the name may be referred to as a variable; if it is, then the remainder of the items (not statement name references) are processed accordingly. In this case, the "ASSIGN A, TO B" statement is treated as a "definition point" for the variable B. (Definition point is described below.) A group which contains an assigned variable may, therefore, have both statement referencing items and variable referencing items without generat-

ing error messages. Note, however, that such a variable may not be subscripted; an error item will be recorded if such a variable is subscripted.

VARIABLE GROUP

If the group obtained does not contain any specified defining items, it is assumed to represent a variable. All occurrences of the variable in the source program will be in the group. The function of some of the items is to indicate information about the variable (e.g., it is floating point, double precision, 3-dimensional, is in an Equivalence or Common statement, etc.). When the mode is not specified, its first letter determines its arithmetic mode according to the LSC naming conventions. This information about the variable is collected to form a single mode word for the variable. This mode word is attached to every item in the group generated from an executable statement.

If the variable occurs in a dimension statement, Phase III does not know what the maximum dimensions are since these may be parameters which are assigned in a different group by a Parameter statement, however the number of dimensions is known. The number of dimensions is assigned to the mode word as is the location in the dimension table where the maximum dimensions for the variable are to be found. (The actual maximum dimensions will be stored there by Phase IV).

Later, in the compiler (Phase VI), an analysis is performed to avoid the recomputation of common subexpressions within a "block." In order to perform this analysis, it is necessary to know the last definition point (DP) of each occurrence of a variable. The last definition point shows where the last point (with the smallest sequence) in the program was that the variable was defined. A defining item may come from the following sources: (1) An Input Statement, (2) an Argument of a CALL, (3) the occurrence of a variable on the left hand side of an equation (a store item) or an ASSIGN statement. A defining item will have its sequence number (DP) assigned to every other item below it in the group until another defining item in the group appears, then the sequence number of the new defining item is used for DP, etc. (Recall that a group is sorted on the Phase I assigned sequence number in File 3).

Although the main function of Phase III, with respect to a variable, is generating a mode word and assigning definition points to each occurrence of the variable in an executable statement, some items require additional processing. These are:

1. Induction Variable (the controlling variable of a DO loop).

   In the statement: DO $\mu$, I = L, M, N

   I is the unduction variable. Every occurrence of the quantity, I, in the range of the DO (all statements from the DO, down to, and including $\mu$) must be recognized and associated with the induction variable in the DO statement. It is possible, in a source program, that several other induction variables named "I" are controlled by different DO loops. The variable, I, may also appear outside the range of a DO. Any other item (not a defining item) in the group whose sequence number lies in this range is identified as being associated with that particular induction variable. This information is retained in the mode word for that item as well as a new name for the induction variable of a DO. In the case of induction variables, the DO statement is the definition point for the variable. Phase III will error any defining item for the induction variable in the range of a DO. (i.e., a defining item between a 19 and a 20 item).

2. Dummy Variables in an Arithmetic Function Definition

   Phase I will have associated with each dummy variable $X_i$ in:

   ABCF ($X_1, X_2, \ldots X_n$) = expression ($X_1, X_2, \ldots X_n$) an identification that they are dummy variables. During Phase III any occurrence of $X_i$ in the sequence range for the function definitions will be renamed, but a name $X_i$ occurring elsewhere in the program will not be confused with a dummy variable. Furthermore, a name may be the dummy variable of several arithmetic functions, it will be unambiguously renamed for each function. The arithmetic mode of the variable, $X_i$, will be defined in the same manner as other variables.

3. Variable storage items are output for non-dimensioned, non-common, non-equivalenced variables.

Further checks performed in Phase III.

1. If N appears in: FUNCTION N(. . .)
then N must also appear in the group as a non-subscripted store item or in an input statement list.

2. If N appears in: CALL N(. . .)
then every occurrence of N in the group must be from a CALL N(. . .).

3. If N appears in: SUBROUTINE N(. . .)
then there must be only that one item in the N group (i.e., N may not occur anywhere else in the program).

4. A name will be indicated as a subprogram reference ( in its mode word) if it is followed by a left parenthesis but does not occur in a DIMENSION statement.

5. If the first item of a group is a built-in or library function item which was generated in Phase II, the group is processed accordingly. Library function names have been sent to File 91 by Phase I; built-in function names are dropped, since these functions are considered to be unique operators after Phase III.

ERROR PROCEDURES

The recognition of an error in the source language causes a two-word error item to be sent to File 92 and an appropriate item to File 4; the File 4 item generally contains the usual sequence information in word one, the mode word for the group being processed in word two, and the third word of the item in File 3 in the third word. If the error

detected would create anomalies in later phases. Phase III will output a necessary item to allow the compilation to be completed.

OUTPUT FROM PHASE III

Items output to File 4 have a three-word format:

| | | |
|---|---|---|
| Word 1: | Digits 1-5 | Sequence number |
| | Digits 6-10 | Subsequence number |
| | Digits 11-12 | Item identification |
| Word 2: | Mode word | |
| Word 3: | Special information dependent upon item type. | |

1. References to variables

ID = 14, 21, 22, 25, 26, 27, 28, 29, 46

Word 3 contains the definition point (DP) right adjusted.

2. Referenced statement

ID = 08, 11

Word 3 contains (DS, L, P)
If any non-executable statements (ID = 09) were references, their ID was changed to 08.

3. Statement references

A. ID = 16

Word 3 contains the sequence of the continue item right

adjusted. If this item occurs in an assigned group, or should be assigned the D digit of the mode word is 1.

B. ID = 12

Word 3 contains the alphanumeric name of the assignee.

C. ID = 35

This SAL EXITS item dropped (see SUCCESSOR).

4. FORMAT statement

ID = 10

This item is dropped.

5. Format references

ID = 15

Word 3 contains (DS, L, P)

6. PARAMETER

ID , 01

Word 3 contains the value of the parameter in floating point notation. The B digit of the mode word is 1; the C digit, 2. Thus, a parameter must be an integer and need not appear in a mode specifier statement (ID = 06).

7. Parameter references

ID = 45

Word 3 contains the value of the parameter in floating point notation.
Valid references, ID = 03 or 14, are changed to 45.
Missing parameter values are set equal to 1.0

8. CONSTANT statements

ID = 38

Word 3 contains the value of the low order part of the constant. It is not checked for being non-zero, nor is an improper mode word B digit changed. Constant references are treated like variable references.

9. Mode specifier

ID = 06, 07

These items are dropped.

10. DIMENSION

ID = 02

Word 3 contains the number of dimensions in fixed point, right adjusted. If more than one dimension declaration is made for a variable, all save the first have ID = 50.

11. EQUIVALENCE and COMMON

ID = 04 and 05

Word 3 contains zeros

12. Induction variables

A. ID = 19

| | |
|---|---|
| Word 1 | DS, 1, 19 |
| Word 2 | mode word for non-induction variable |
| Word 3 | O . (Se) . 00 |

Se = sequence of 20 item or, 0———0 for nested DO's on same induction variable).

13. References to induction variable (within DO loop)

A. ID = 07

| | |
|---|---|
| Word 1 | S . SS . 07 |
| Word 2 | 411 0000 0FFFF |
| Word 3 | DS , L , P |

Valid references (ID = 14, 28) are changed to 07. Arguments of a CALL, ID = 31, are unchanged.

14. CALL

ID = 30

Word 3 contains (DS , L , P) (see 21).

15. Arguments of CALL, unsubscripted

A. ID = 31

These items are treated as references to variables (ID =14), statement names (ID = 16), or formats (ID = 15) as appropriate.

Arguments of CALL, subscripted

B. ID = 41

These items are treated as references to dimensioned variables (ID = 25).

16. SUBROUTINE

ID = 32

Word 3 contains the number of arguments
The name becomes the SAL LABEL for the object program. (See "OUTPUT TO FILE 91" - below).

17. Argument of SUBROUTINE

ID = 33

References to an argument will contain the argument's order number (within the list of arguments) in the mode word.

Word 3 contains the definition point.

18. FUNCTION

ID = 34

Word 3 contains the number of arguments.
The name becomes the SAL LABEL for the object program. (See "OUTPUT TO FILE 91" - below).

19. Arguments of a FUNCTION

ID = 42

References to an argument will contain the argument's order number (within the list of arguments) in the mode word.

Word 3 contains the definition point.

20. Store expression item

ID = 39

Word 3 contains zeros.
(The mode word of this item is required to properly complete the Polish notation analysis in Phase IV).

21. Library Request Function

ID = 44

This item is output to File 4 only for CALLed and implicit functions. It provides proper mode information for referenced functions. Word 3 contains the alphanumeric name of function to be included in the object program.

22. Name of Arithmetic Statement Function

ID = 23

Word 3 contains the number of arguments. (See "OUTPUT TO FILE 91" - below.

23. Argument of Arithmetic Statement Function

This item is dropped; however, its subsequence number becomes a pseudo-name (UFFFF of mode word) of references to this variable which are in the function skeleton appearing on the right hand side of the function statement.

24. Variable Storage Item

For all non-dimensioned, non-equivalence, non-common variables, a storage item is generated with the following format:

| | |
|---|---|
| Word 1 | (SD-5) . i . 06 |
| Word 2 | mode word |
| Word 3 | 0————0 |

(SD-5) is the DIMENSION sequence counter less 5;
(i) is a counter denoting the $i^{th}$ such variable.

25. SUCCESSOR Items (non-extensible. - see Phase VIII).

| | | |
|---|---|---|
| Word 1 | (SREF) . 0 . 03 | sequence of reference |
| Word 2 | 0200000 (SCONT) | sequence of continue |
| Word 3 | 0————————01 | |

26. TO Item

| | | | |
|---|---|---|---|
| A. | Word 1 | (DSc) . 00007 . 40 | DO sequence of continue. |
| | Word 2 | (DSr) 0————0 | DO sequence of reference. |
| | Word 3 | (Sc) 0————0 | |

FROM Item

| | | | |
|---|---|---|---|
| B. | Word 1 | (DSr) . 00006 . 43 | |
| | Word 2 | (DSc) 0————0 | |
| | Word 3 | ($S_R$) 0————0 | |

27. Function Reference

ID = 47

If a group contains only 47 and/or 25 items it will be treated as a reference to a compiled function.

Word 3 contains zero. (see 21).

28. Reference to assigned variable) from "ASSIGNED GO TO")

ID = 17

Word 3 contains (DS , L , P)
The mode word D digit is 1.

29. Listed Assignment

ID = 18

These items are dropped; they produce SUCCESSOR and may produce TO and FROM items.

30. Assignee

ID = 13

Word 3 contains the DP, which is the sequence of this item.
The mode word D digit is 1.

OUTPUT TO FILE 91

A. File 91 is the dictionary file for LSC; it contains all the names (i.e., symbols) used in the source program and names generated by LSC. Phase III will file all names it encounters (one for each group) with two exceptions.

1. Names of unreferenced statements.

2. Names of library and of all built-in functions.

B. The name of an ARITHMETIC STATEMENT FUNCTION, FUNCTION, or SUBROUTINE when filed in File 91, will be immediately followed by (n) items, one for each argument of the subprogram. These list names are generated by converting the File 91 entry number (CRDREF) to alpha-numeric and appending AR.

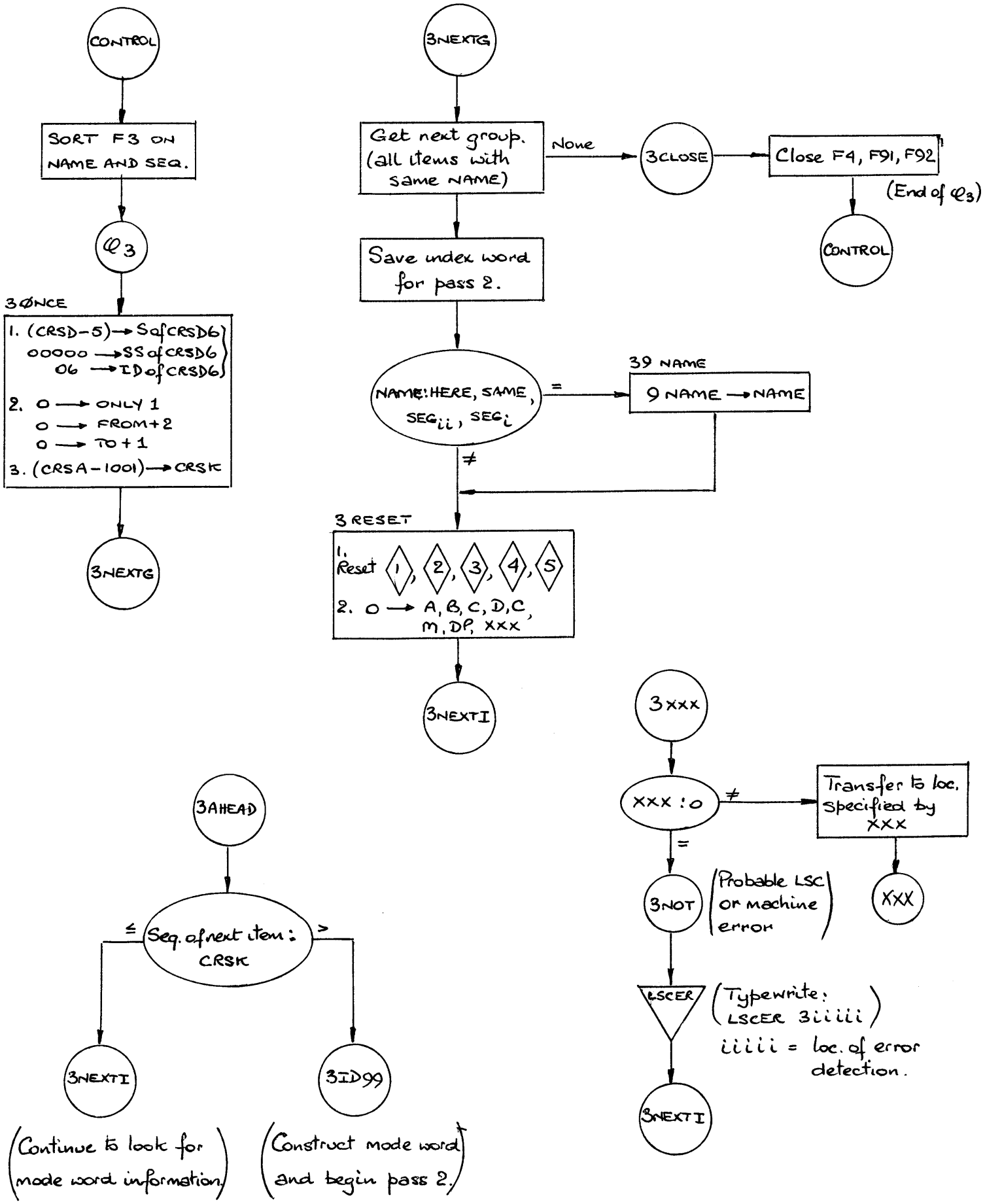
e.g., FUNCTION NAME F (A , B)

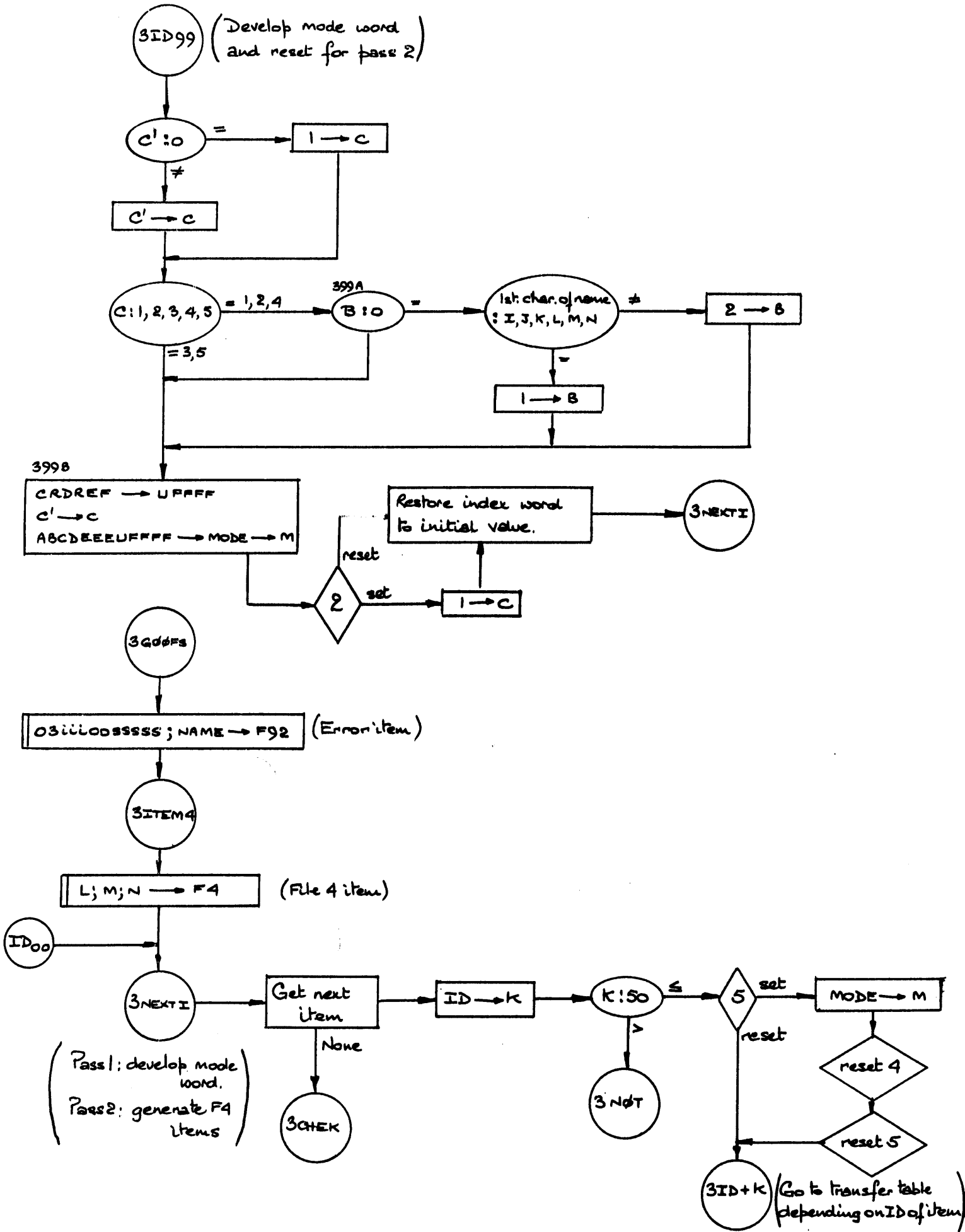
| | | |
|---|---|---|
| FILE 91 | 66 | NAME |
| | 67 | 67AR |
| | 68 | 68AR |

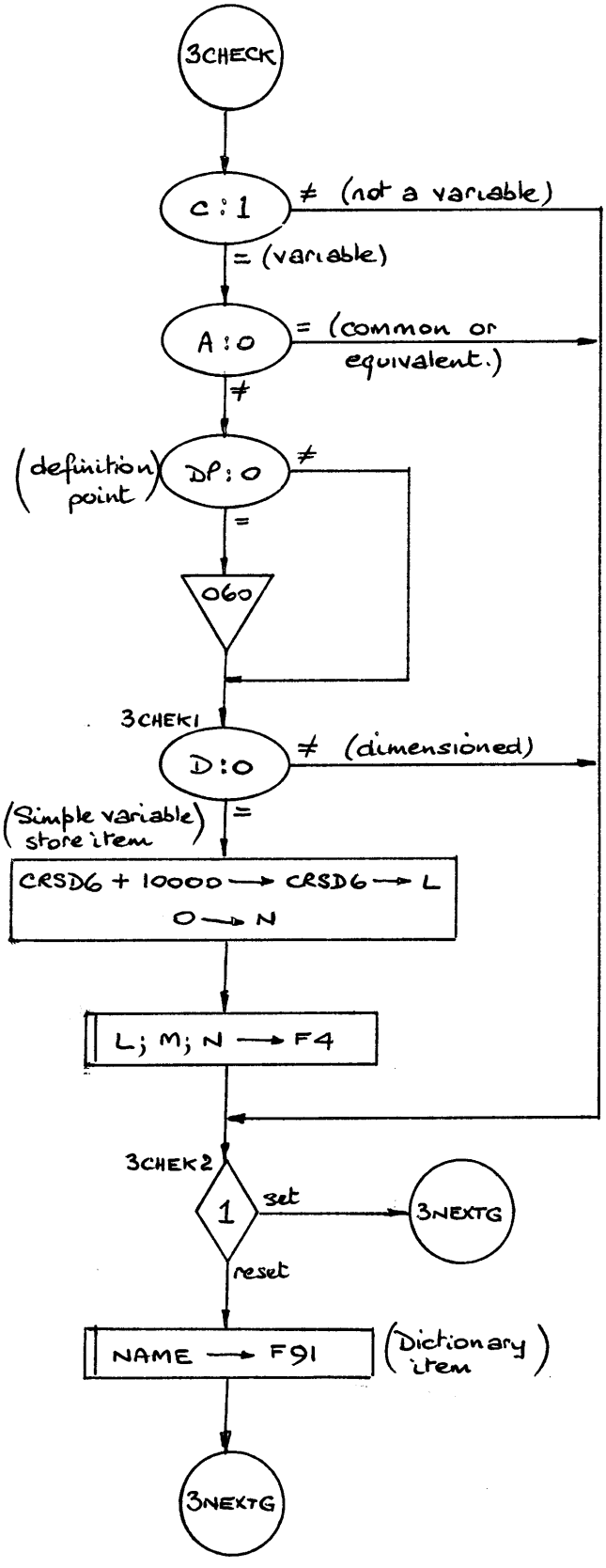
C. LSC names which are incompatible with SAL are prefaced with a 9 by Phase III. These names are; HERE, SAME, and SEG ii. The name of a FUNCTION will be attached to the first executable statement of the subprogram; all references to this name within the subprogram will be prefaced with an 8 to avoid having a statement name and variable with the same name.

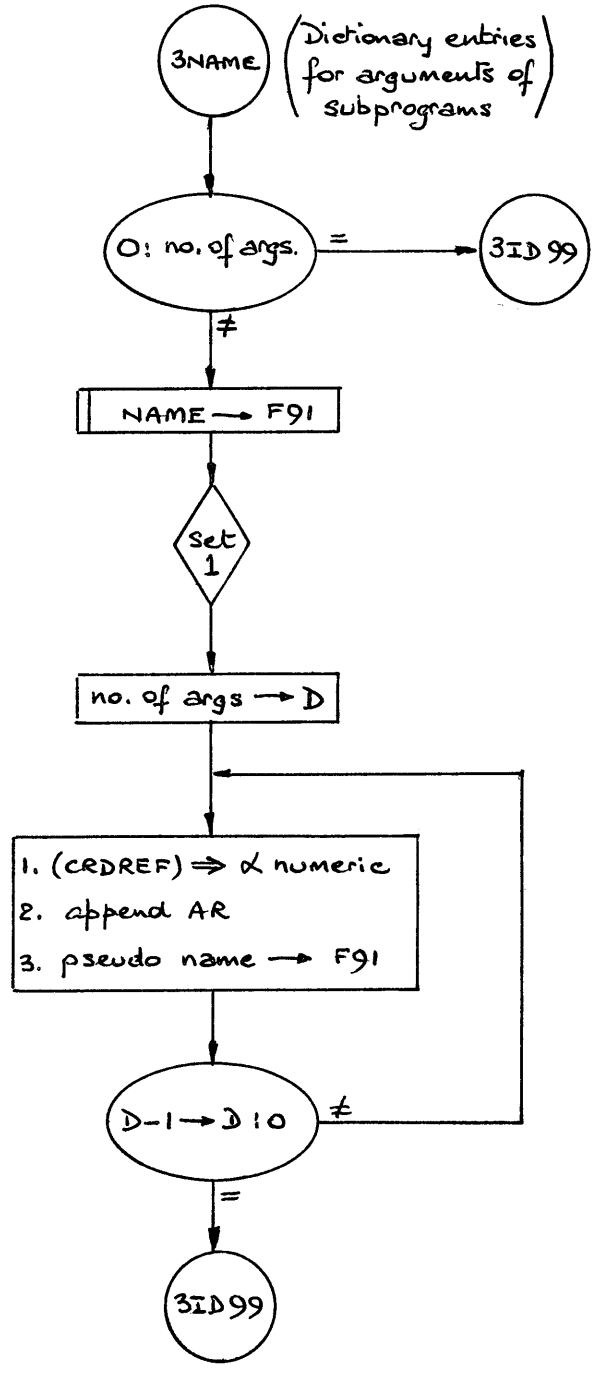
OUTPUT TO FILE 92

Error items (described in the Phase IX report) are output as two-word items. They will contain the alphanumeric name of the group in question and hence will facilitate understanding the source error (or probable error). A complete list of all conditions which will cause an error item to be output is found in the Phase IX listing (the "parallel codedit").


CONTROL
SORT F3 ON NAME AND SEQ.
℘3
3ØNCE
1. (CRSD-5) → SofCRSD6
00000 → SSofCRSD6
06 → IDofCRSD6
2. 0 → ONLY 1
0 → FROM+2
0 → TO+1
3. (CRSA-1001) → CRSK
3NEXTG
3NEXTG
Get next group. (all items with same NAME)
None
3CLOSE
Close F4, F91, F92
(End of ℘3)
CONTROL
Save index word for pass 2.
NAME:HERE, SAME, SEGii, SEGi
=
39 NAME
9 NAME → NAME
≠
3 RESET
1. Reset ◇1, ◇2, ◇3, ◇4, ◇5
2. 0 → A, B, C, D, C, M, DP, XXX
3NEXTI
3AHEAD
≤
Seq. of next item : CRSK
>
3NEXTI
3ID99
(Continue to look for mode word information.)
(Construct mode word and begin pass 2.)
3XXX
XXX : 0
≠
Transfer to loc. specified by XXX
XXX
=
3NOT
(Probable LSC or machine error)
LSCER
(Typewrite: LSCER 3iiiii
iiiii = loc. of error detection.)
3NEXTI

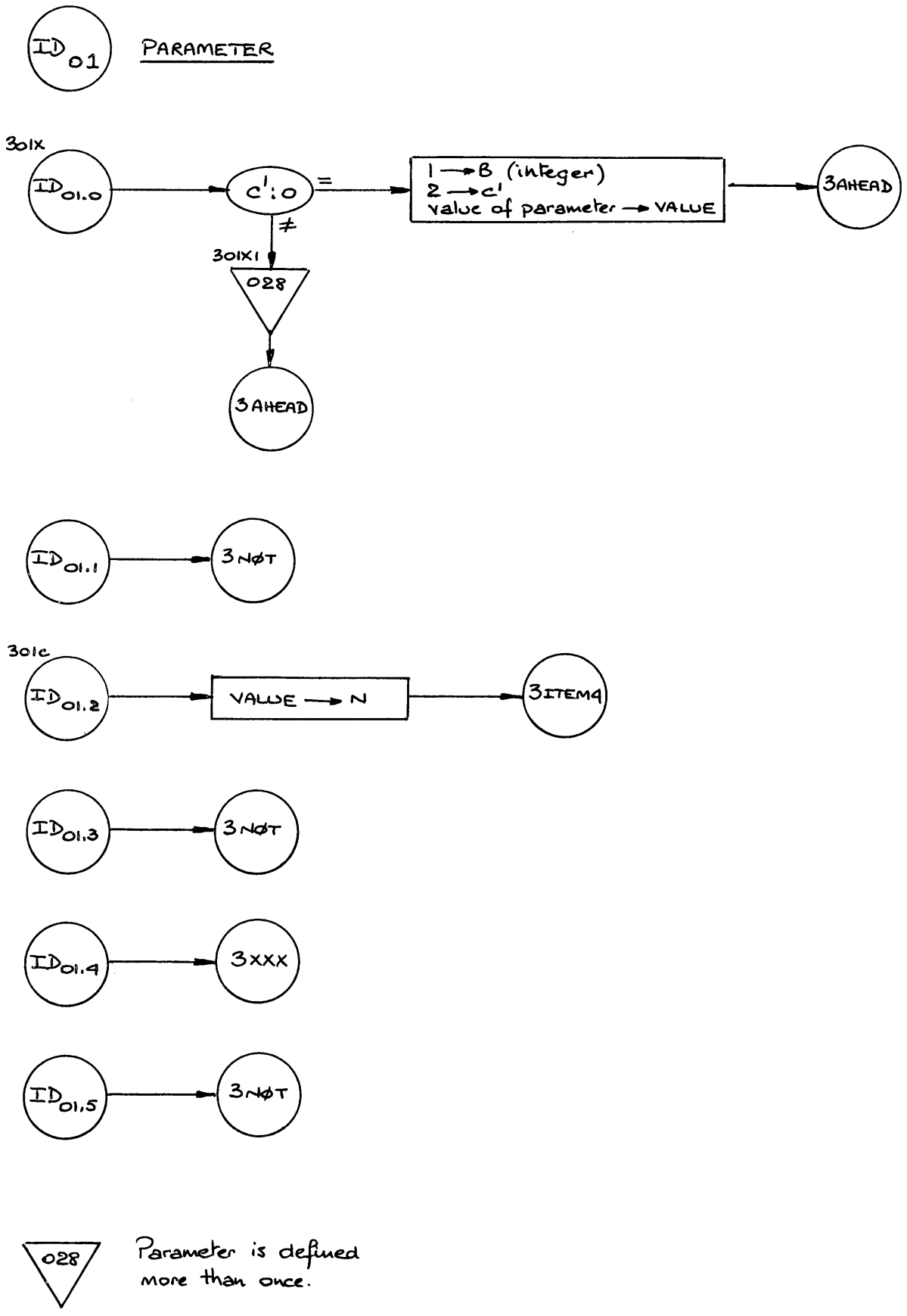

3ID99
(Develop mode word and reset for pass 2)
C' :0
=
1 → C
≠
C' → C
C: 1, 2, 3, 4, 5
= 1, 2, 4
= 3, 5
3999A
B :0
=
1st char. of name : I, J, K, L, M, N
≠
2 → B
1 → B
3998
CRDREF → UFFFF
C' → C
ABCDEEEEUFFFF → MODE → M
reset
2
set
1 → C
Restore index word to initial value.
3NEXTI
3GØØFS
O3iiiiOOSSSSS ; NAME → F92
(Error item)
3ITEM4
L; M; N → F4
(File 4 item)
IDOO
3NEXTI
Get next item
None
ID → K
K :50
>
3NØT
≤
5
set
reset
MODE → M
reset 4
reset 5
3ID+K
(Go to transfer table depending on ID of item)
3CHEK
(Pass 1: develop mode word.
Pass 2: generate F4 items)



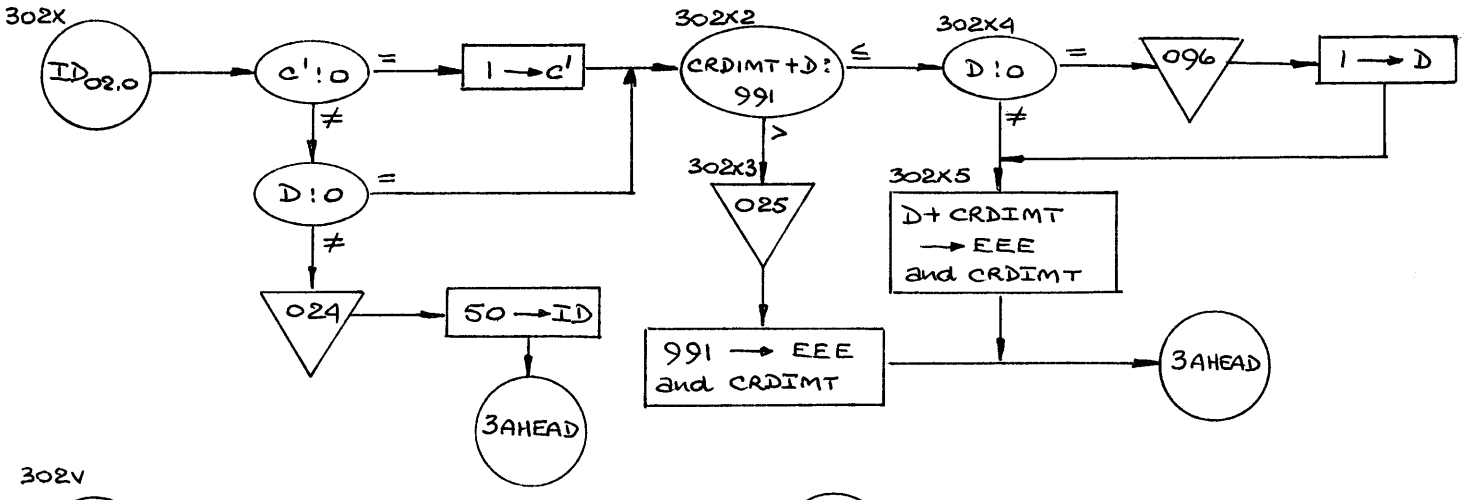



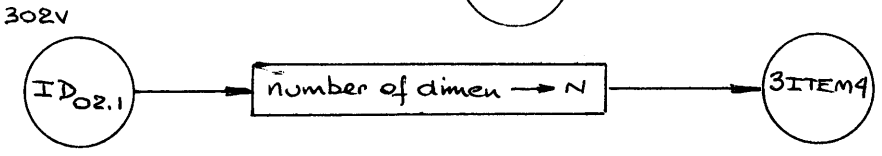

e.g. If the name of a subprogram, which has three arguments, were filed as the 88th item in the dictionary; the 89th, 90th and 91st entries would be, respectively: 89AR, 90AR and 91AR.

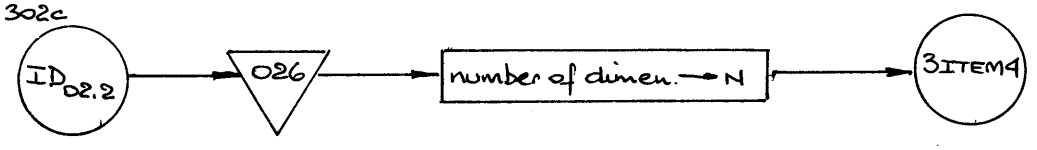
060 No definition point for variable

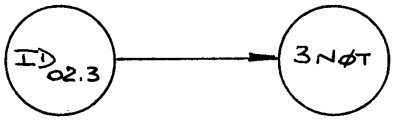

ID 01
PARAMETER
301X
ID 01.0
c' : 0
=
1 → B (integer)
2 → c'
value of parameter → VALUE
3AHEAD
≠
301X1
028
3AHEAD
ID 01.1
3NØT
301C
ID 01.2
VALUE → N
3ITEM9
ID 01.3
3NØT
ID 01.4
3XXX
ID 01.5
3NØT
028
Parameter is defined more than once.

$ID_{02}$ DIMENSION



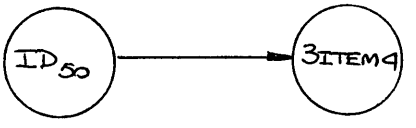











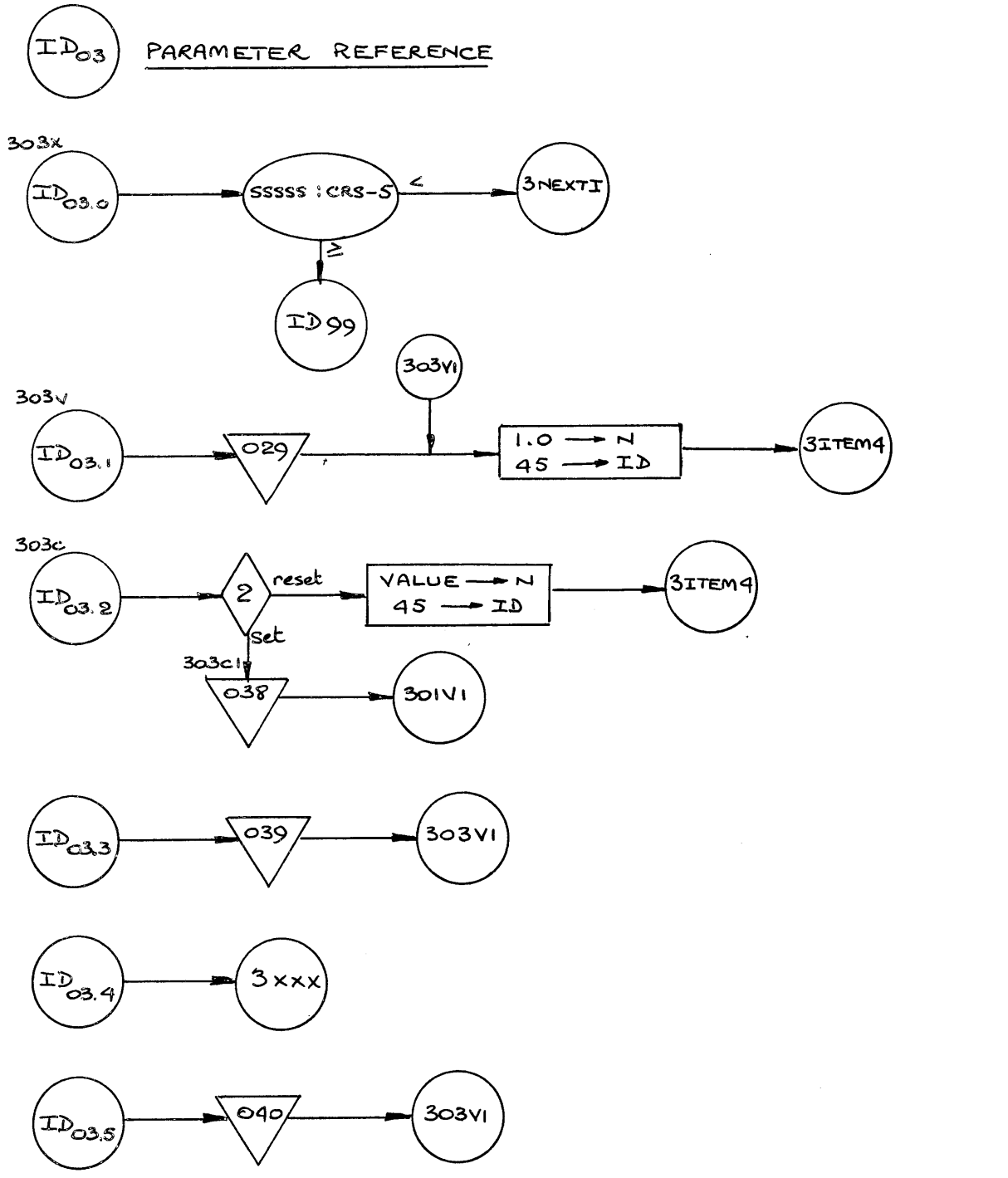


024 Name appears in more than one DIMENSION statement.

026 Dimension variable and parameter have same name.

025 Dimension table exceeded.

096 0 — Dimensioned variable.

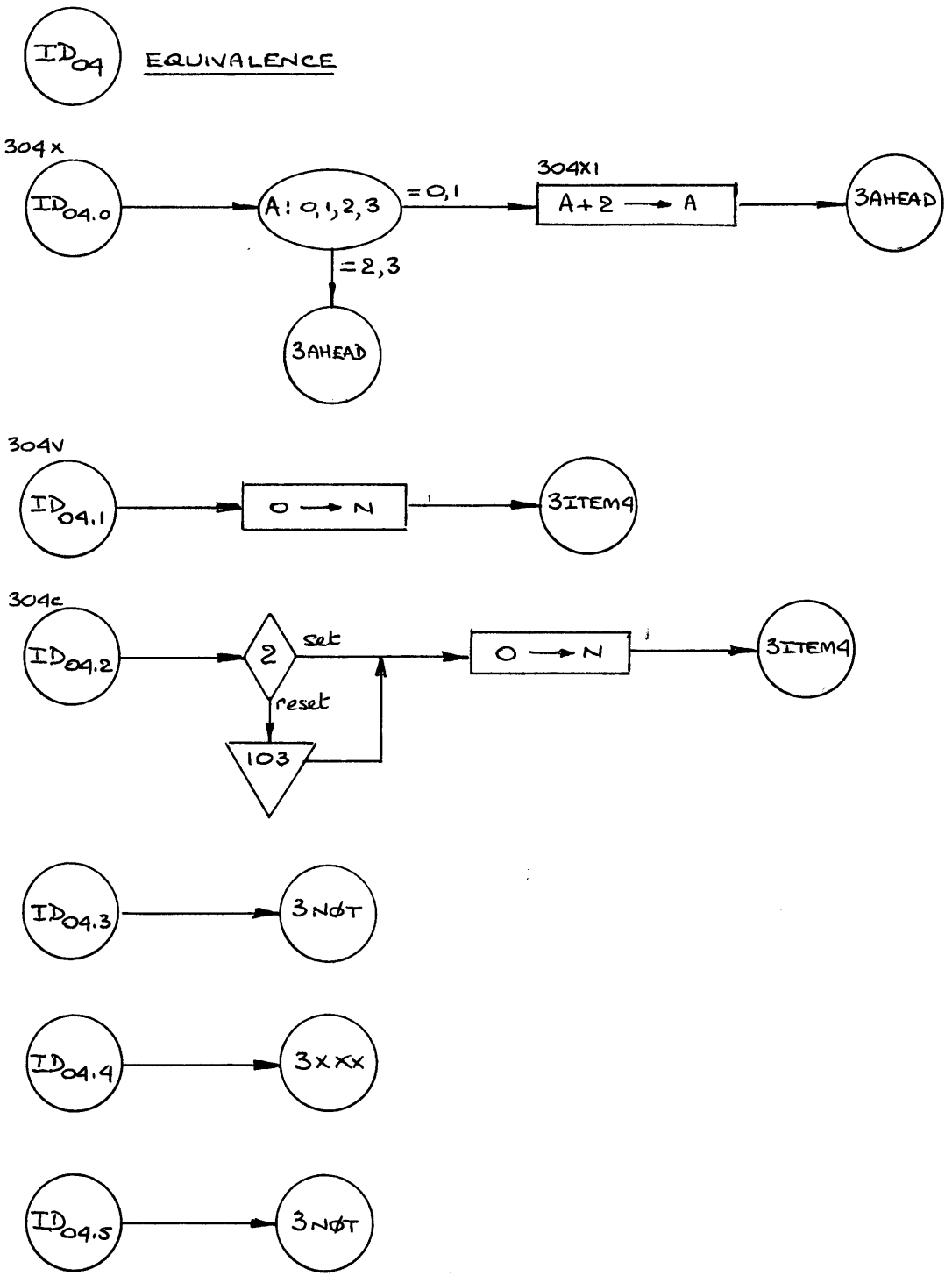



029 Name must be defined by PARAMETER

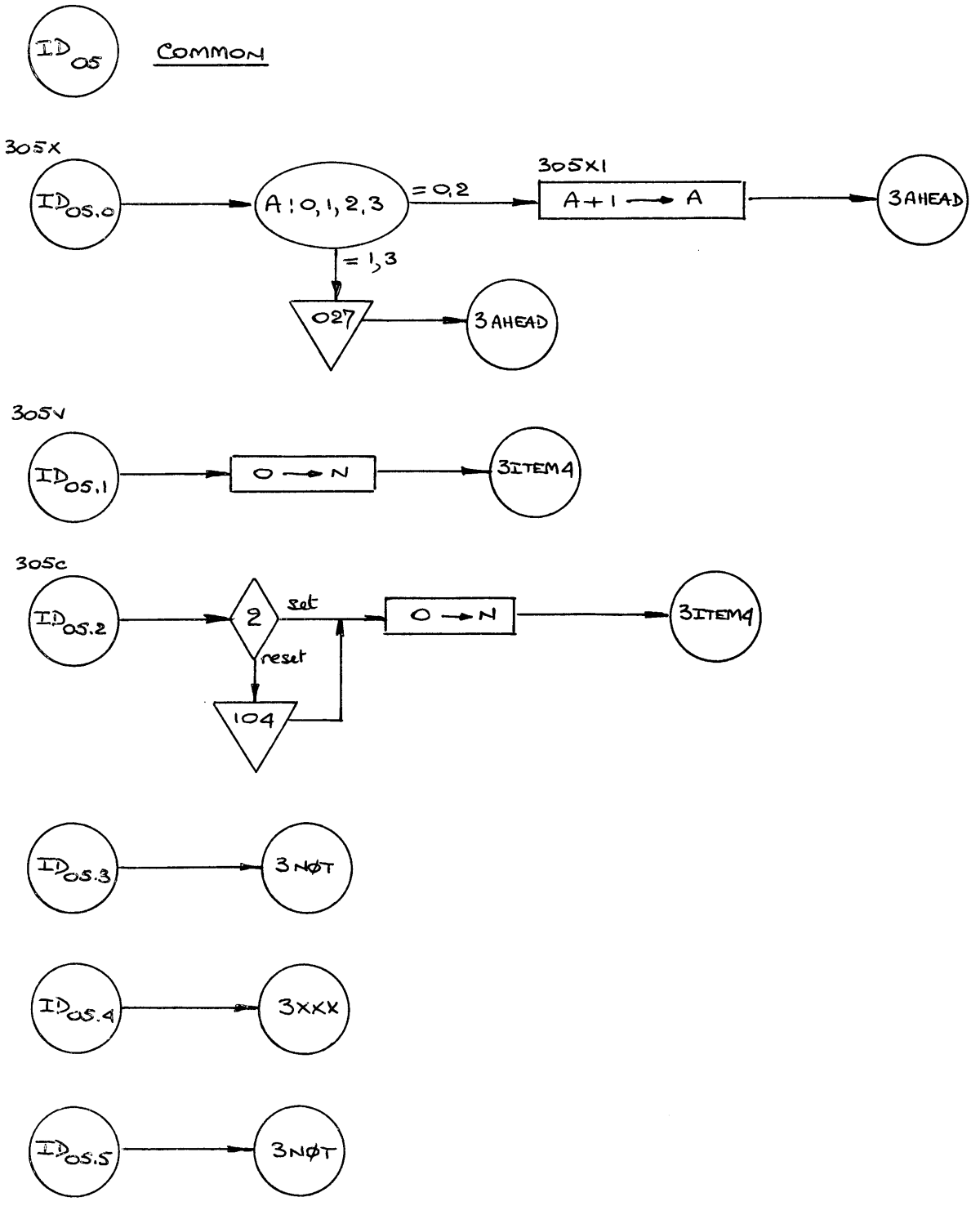
039 Statement name used as parameter reference.

038 Parameter reference for defined CONSTANT

040 Format name referenced as parameter.

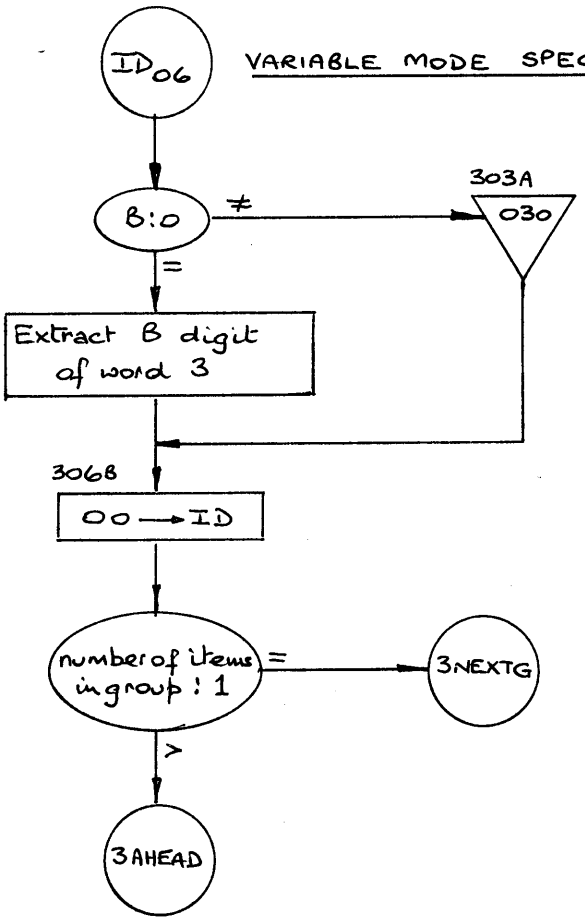


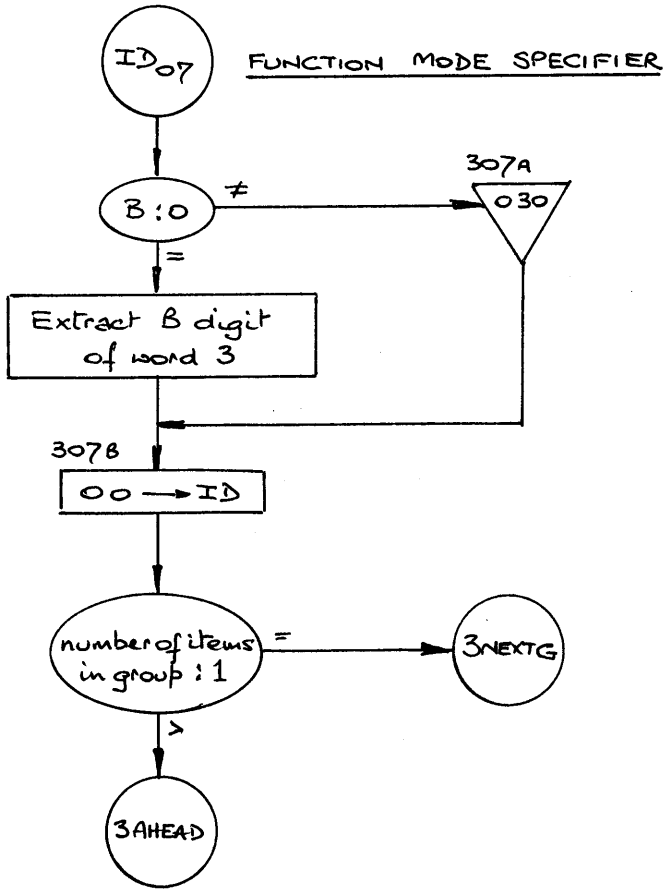

103 Parameter in equivalence list.




027 Name in more than one COMMON statement.

104 Parameter in COMMON statement.

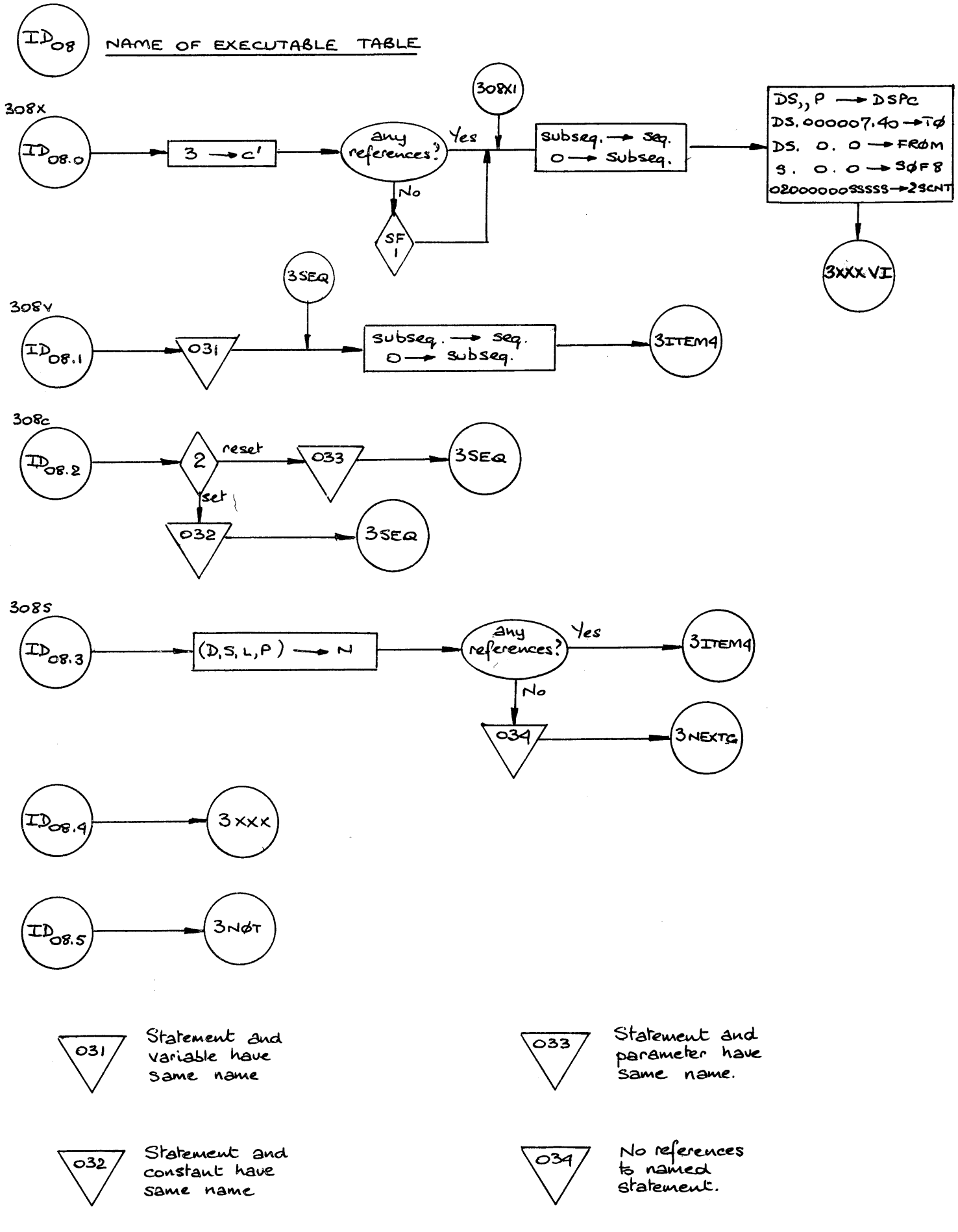


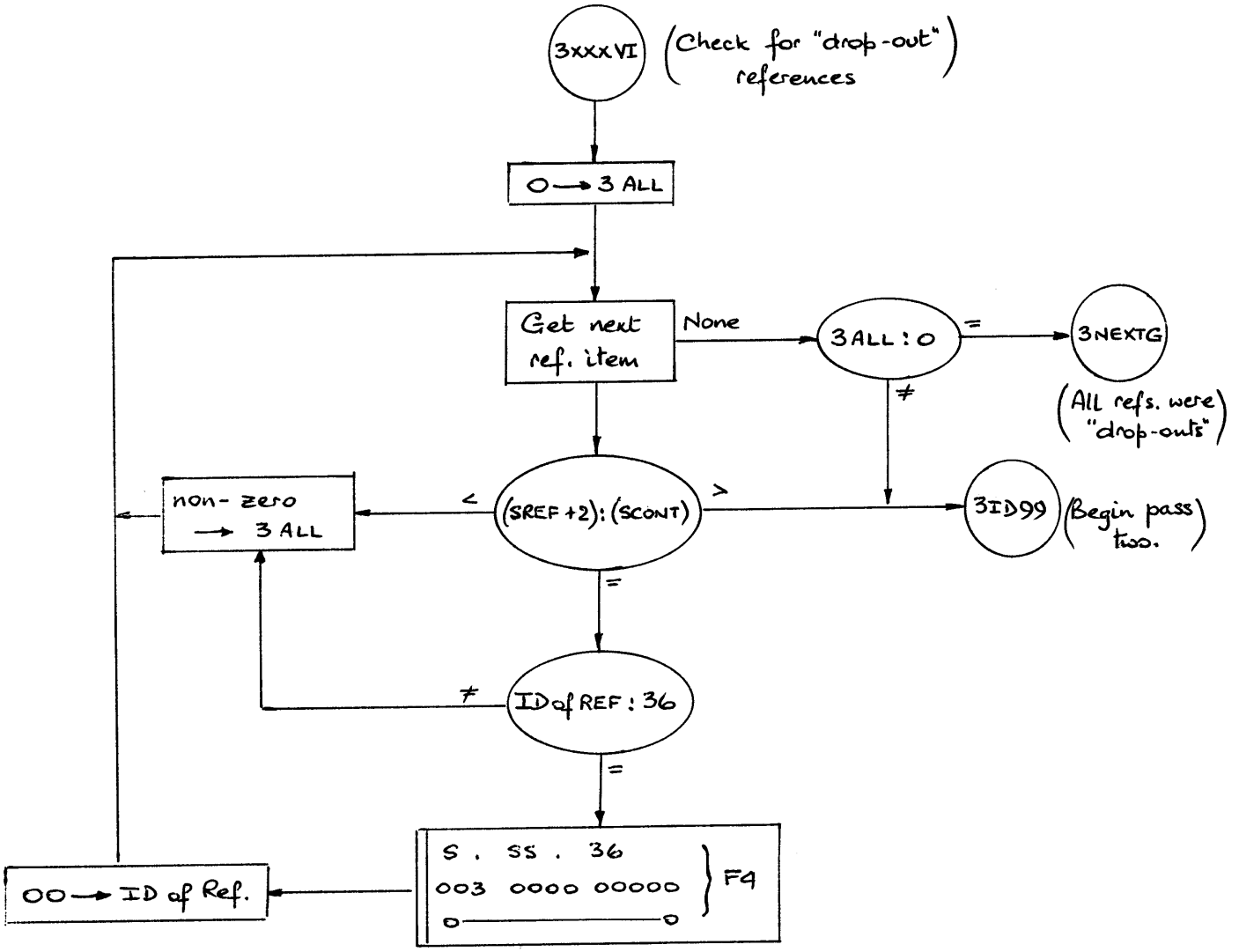



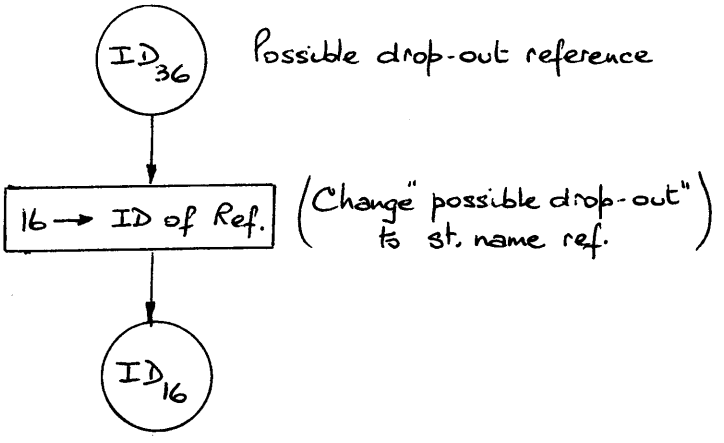


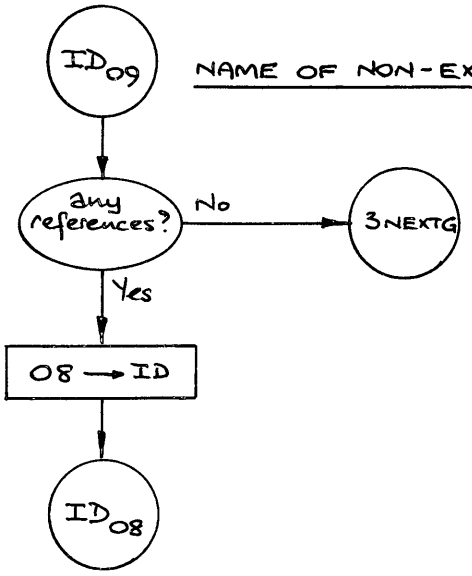


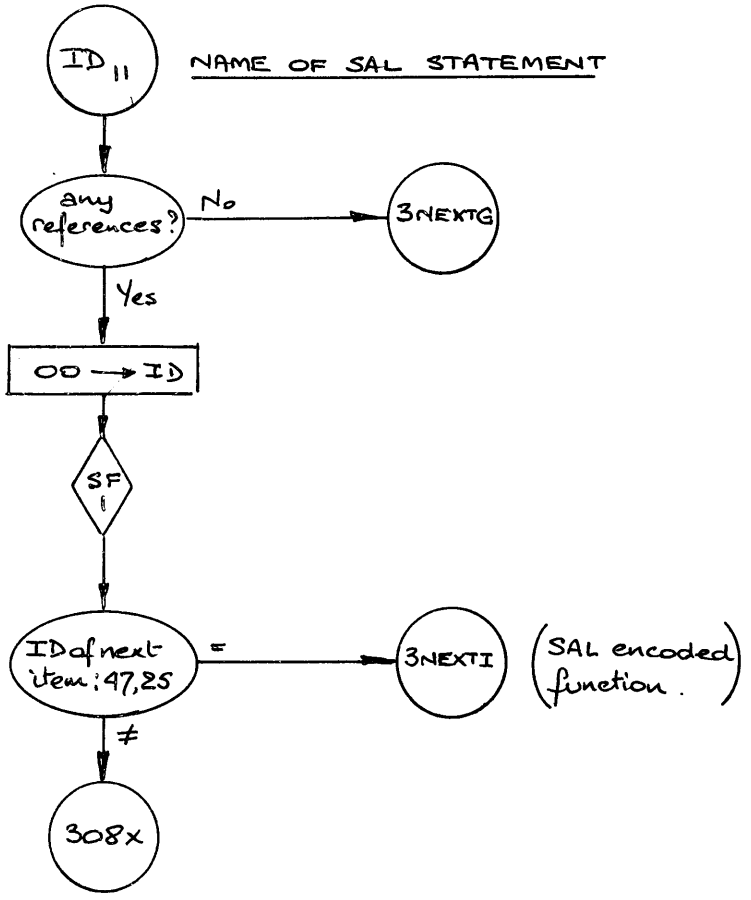
Name appears in more than 1 mode specifier statement.

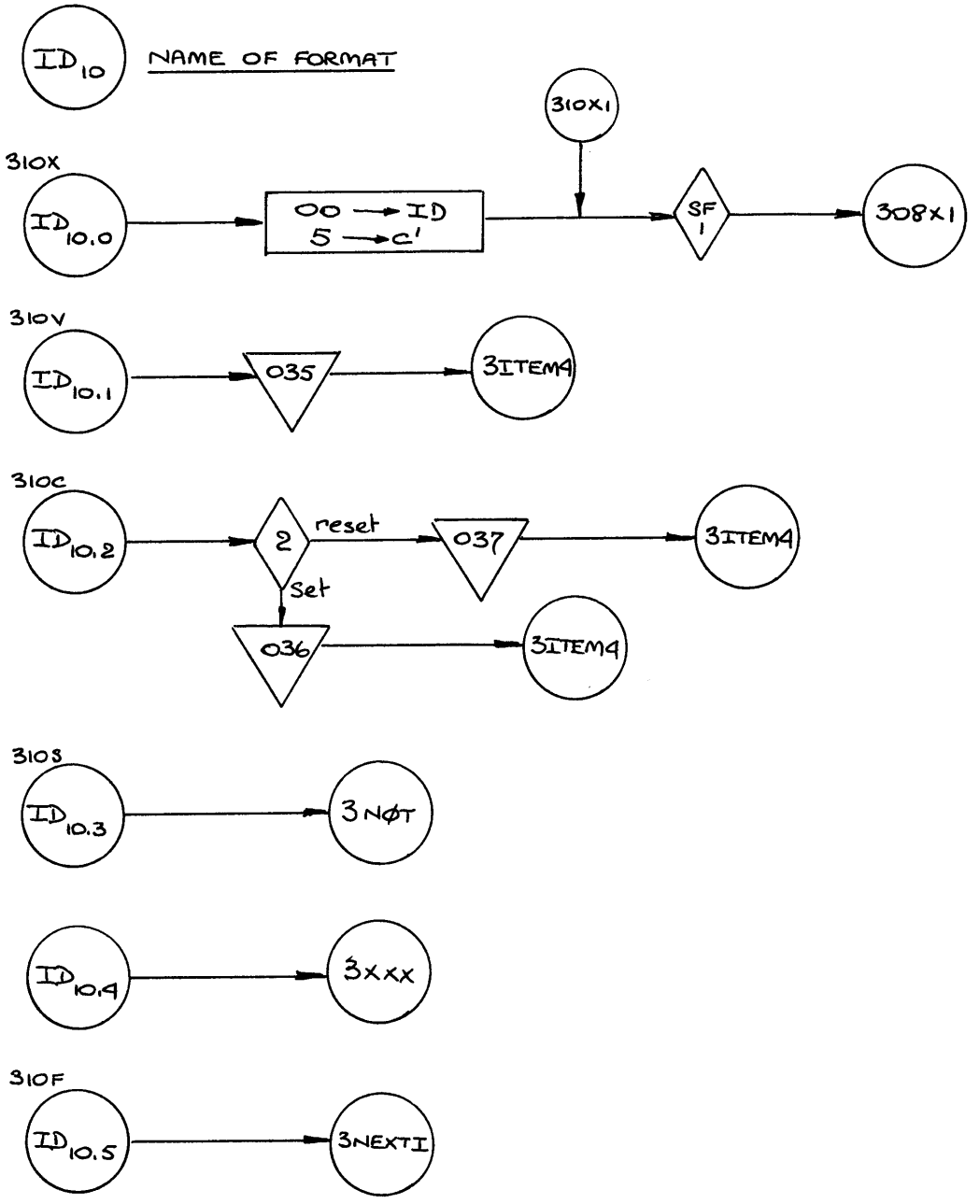

ID08
NAME OF EXECUTABLE TABLE
308X
ID08.0
3 → c'
any references?
Yes
No
SF 1
308X1
subseq. → seq.
0 → subseq.
DS,, P → DSPC
DS. 000007.40 → TØ
DS. 0. 0 → FRØM
3. 0. 0 → SØF8
02000000SSSSS → 2SCNT
3XXX VI
3SEQ
308V
ID08.1
031
subseq. → seq.
0 → subseq.
3ITEM4
308C
ID08.2
2
reset
033
3SEQ
set
032
3SEQ
3085
ID08.3
(D,S,L,P) → N
any references?
Yes
3ITEM4
No
034
3NEXTG
ID08.4
3XXX
ID08.5
3NØT
031
Statement and variable have same name
033
Statement and parameter have same name.
032
Statement and constant have same name
034
No references to named statement.

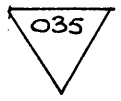

3XXXVI
(Check for "drop-out" references
0 → 3 ALL
Get next ref. item
None
3ALL : 0
=
3NEXTG
(All refs. were "drop-outs"
≠
non-zero → 3 ALL
<
(SREF+2) : (SCONT)
>
3ID99
(Begin pass two.
=
≠
ID of REF : 36
=
S . SS . 36
003 0000 00000
F4
00 → ID of Ref.


ID36
Possible drop-out reference
16 → ID of Ref.
(Change "possible drop-out" to st. name ref.
ID16


ID09
NAME OF NON-EXECUTABLE STATEMENT
any references?
No
3NEXTG
Yes
08 → ID
ID08


ID11
NAME OF SAL STATEMENT
any references?
No
3NEXTG
Yes
00 → ID
SF
IDofnext item;47,25
=
3NEXTI
(SAL encoded function.
≠
308x



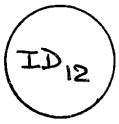

Variable and format have same name



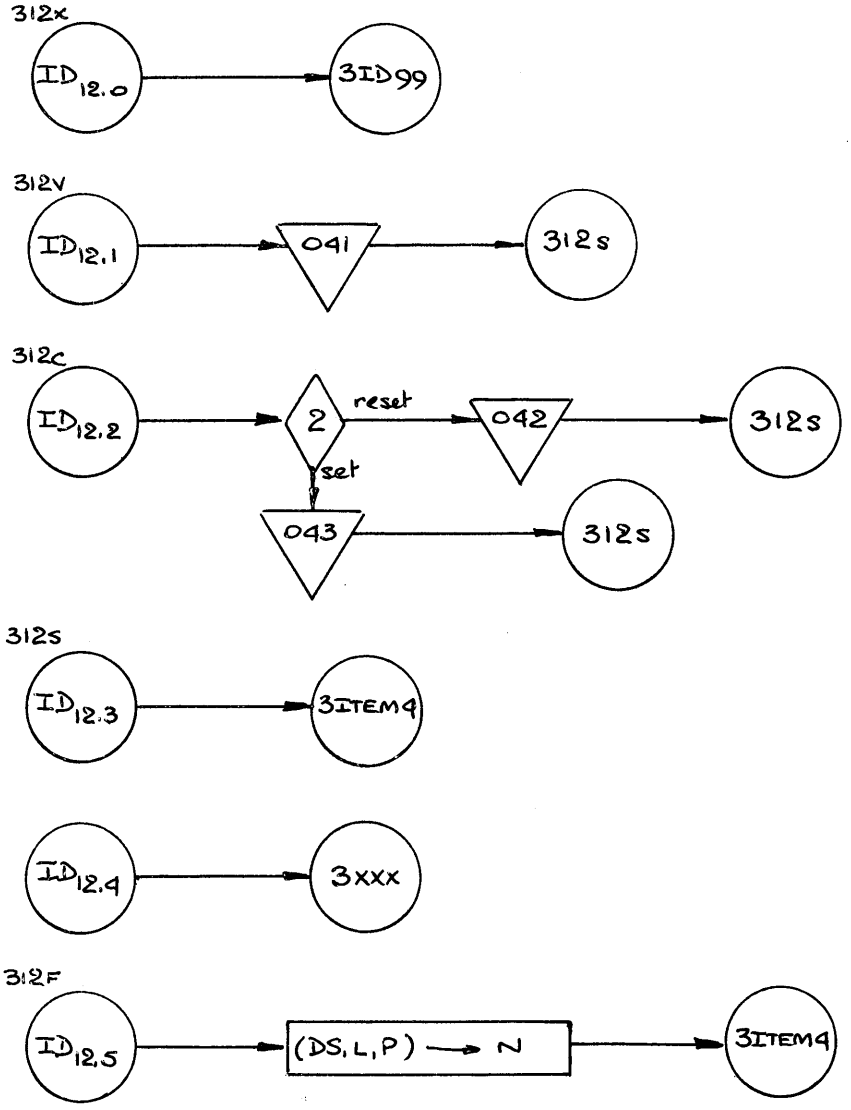

Statement name and parameter have same name




Constant and format have same name



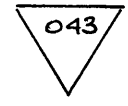

ASSIGNED ST. NAME REFERENCES







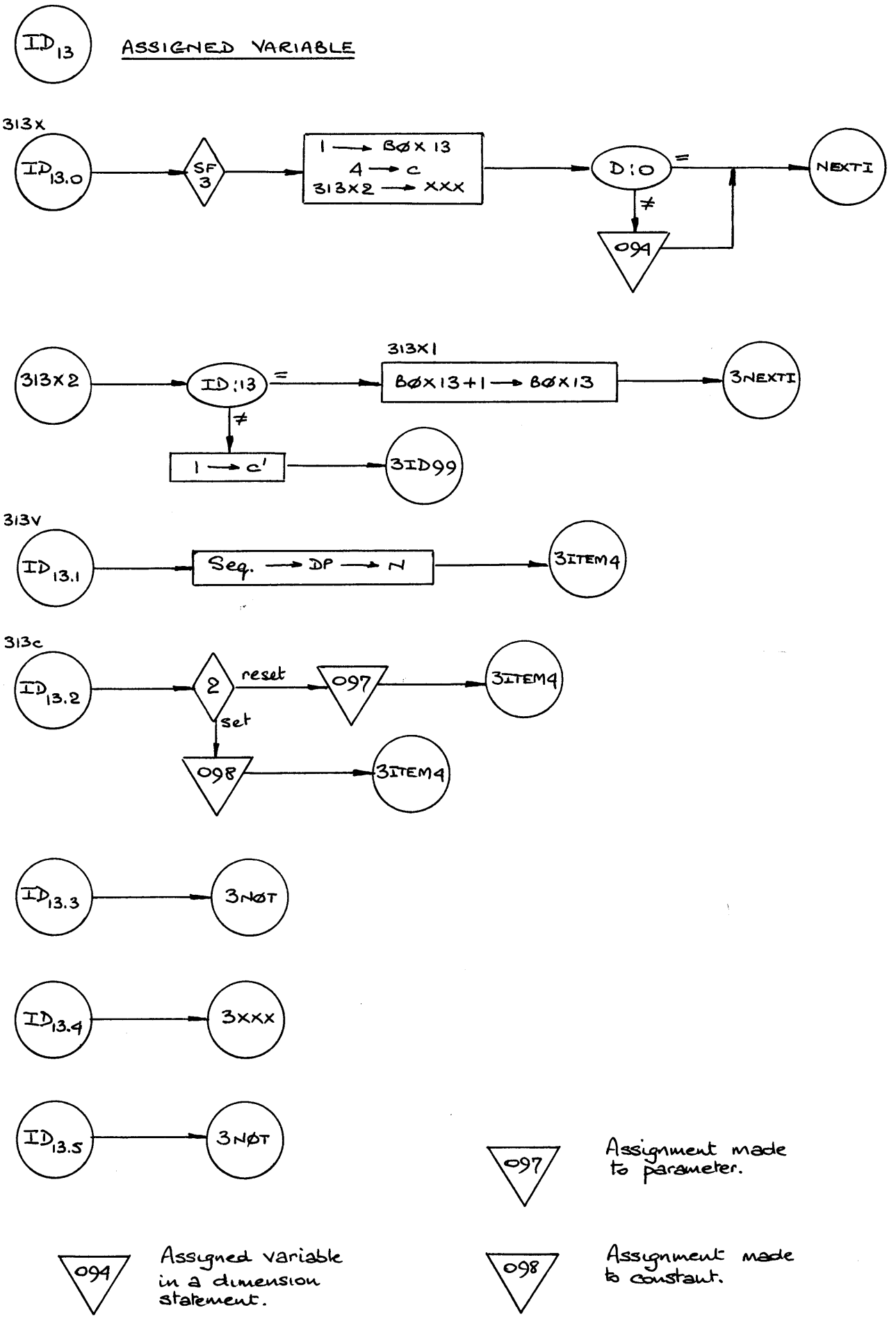
Variable was assigned as statement reference

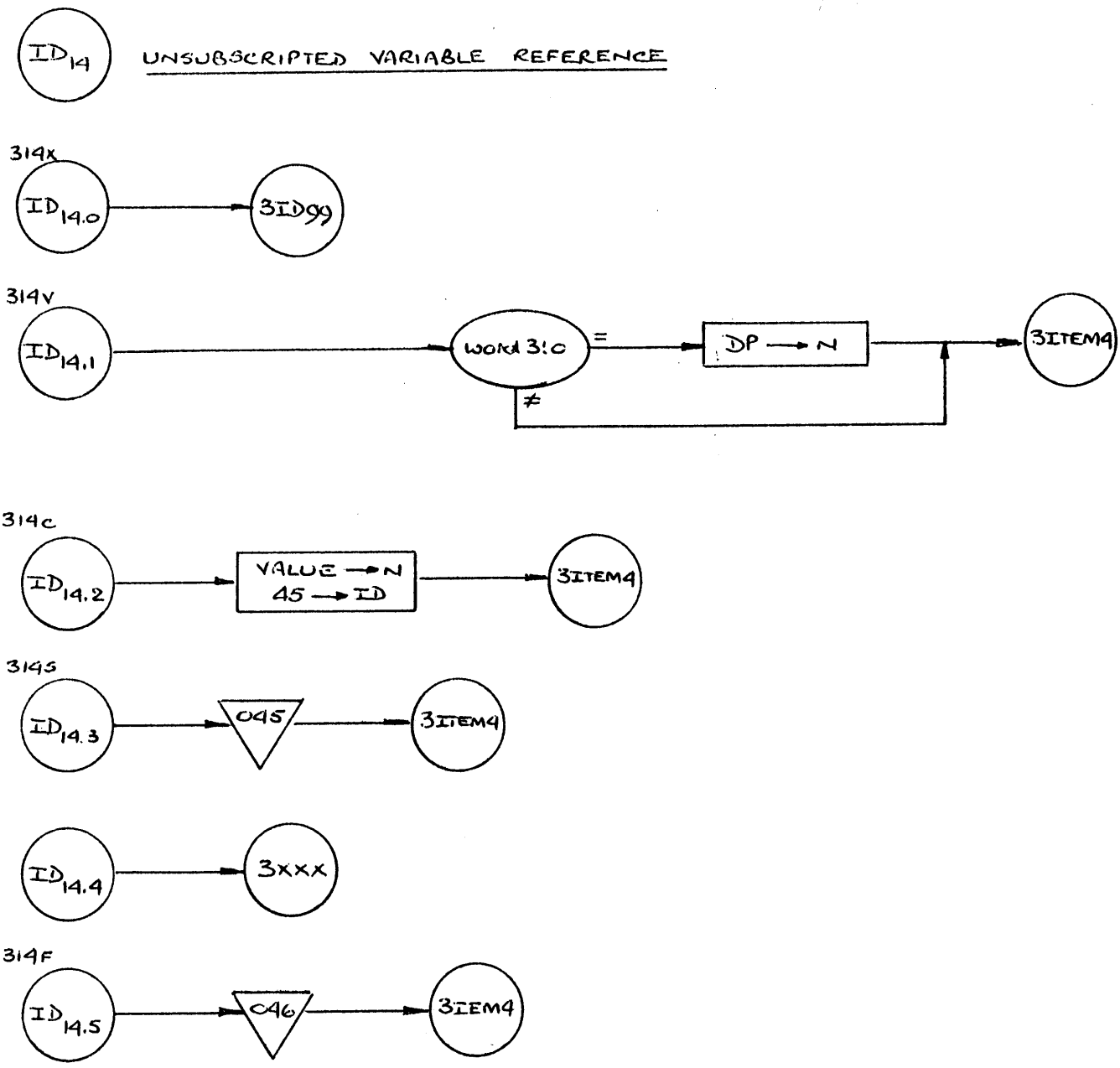



Constant was assigned as statement reference.

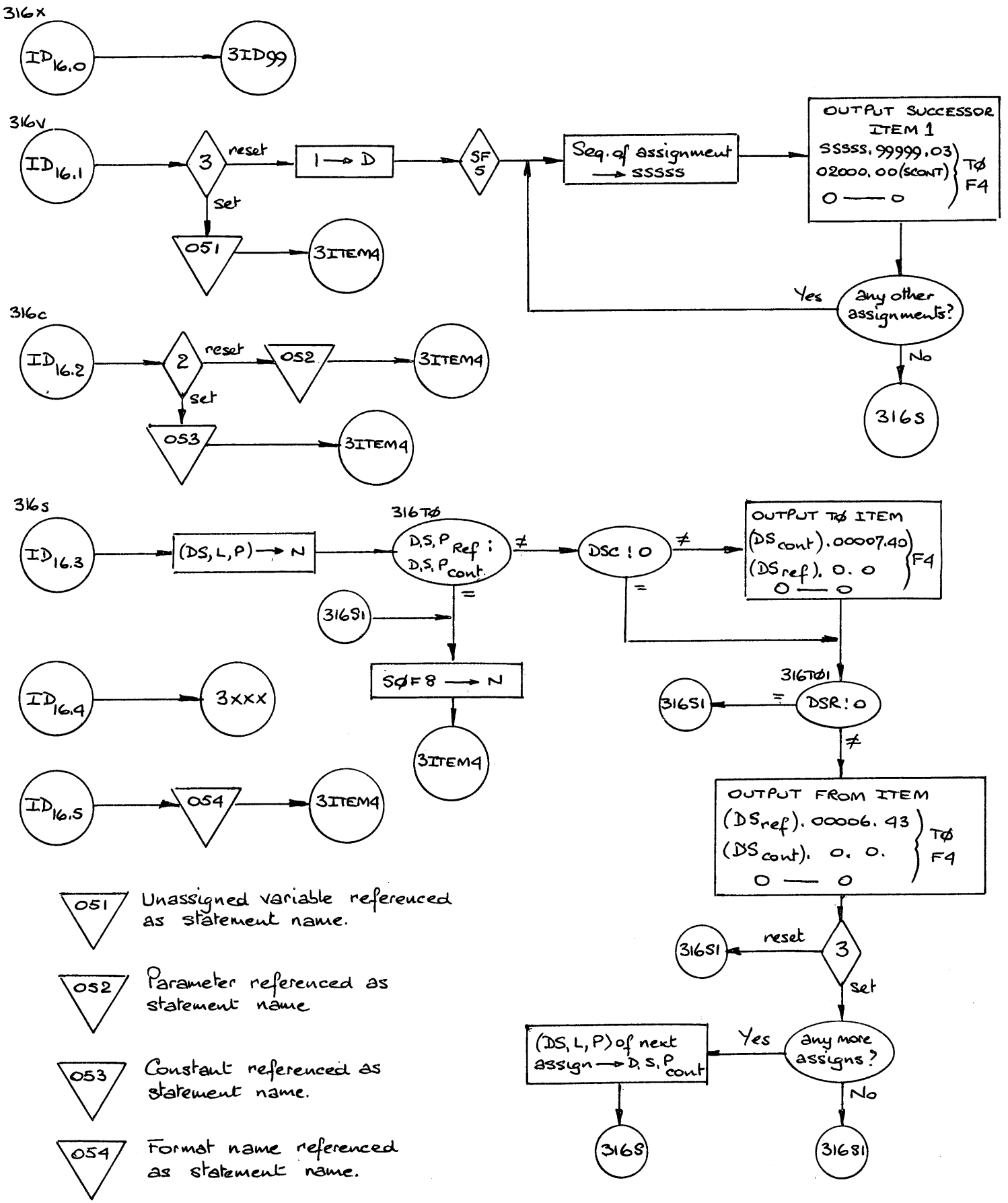


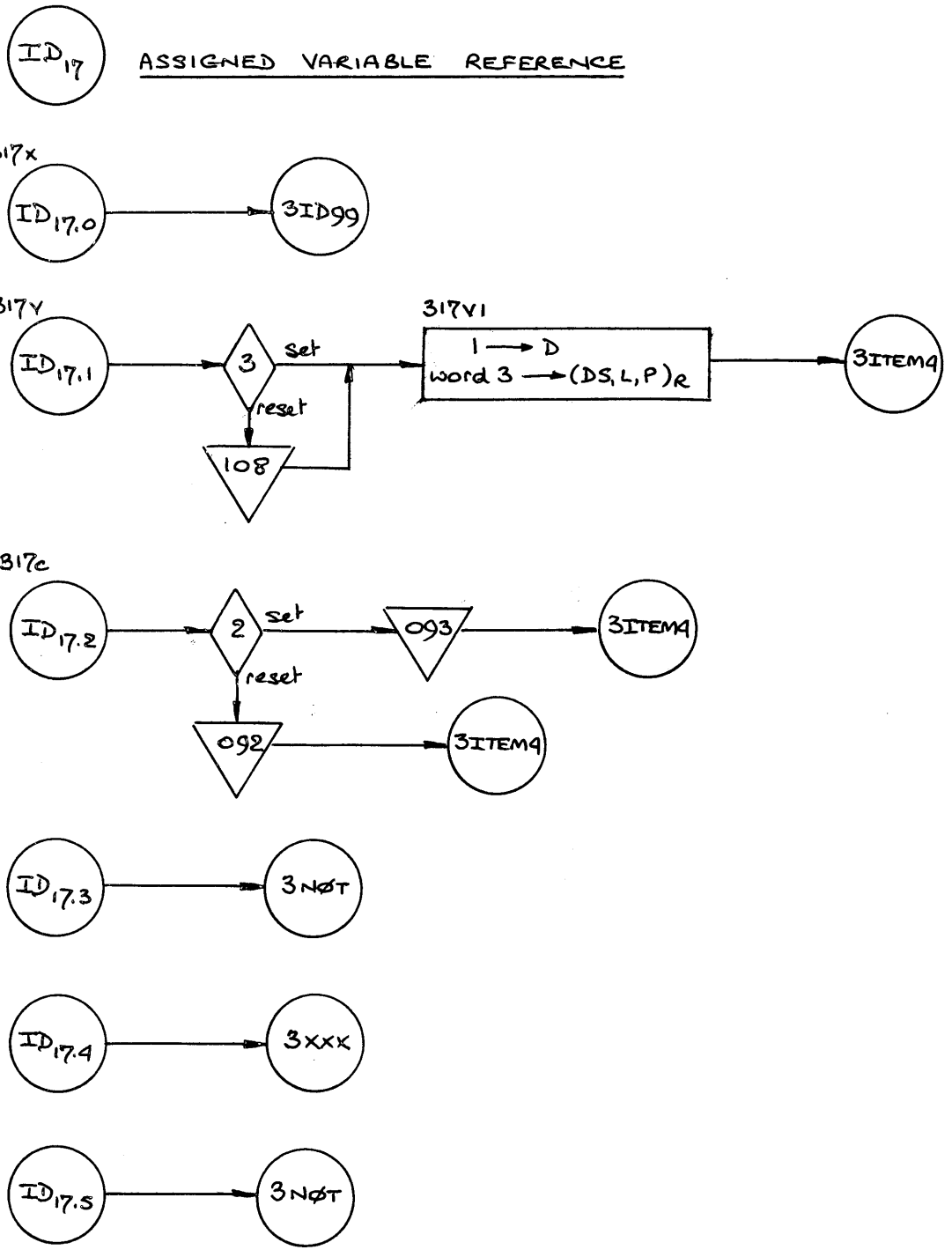

Parameter was assigned as statement reference.


ID13
ASSIGNED VARIABLE
313X
ID13.0
SF 3
1 → BØX 13
4 → C
313X2 → XXX
D:0
=
≠
NEXTI
094
313X2
ID:13
=
313X1
BØX13+1 → BØX13
3NEXTI
≠
1 → C'
3ID99
313V
ID13.1
Seq. → DP → N
3ITEM4
313C
ID13.2
2
reset
set
097
098
3ITEM4
ID13.3
3NØT
ID13.4
3XXX
ID13.5
3NØT
097
Assignment made to parameter.
094
Assigned variable in a dimension statement.
098
Assignment made to constant.


ID14
UNSUBSCRIPTED VARIABLE REFERENCE
314X
ID14.0
3IDGG
314V
ID14.1
WORD 3:0
=
DP → N
3ITEM4
≠
314C
ID14.2
VALUE → N
45 → ID
3ITEM4
3145
ID14.3
045
3ITEM4
ID14.4
3XXX
314F
ID14.5
046
3IEM4
045
Statement name referenced as variable
046
Format name referenced as variable

3ID16 STATEMENT NAME REFERENCE

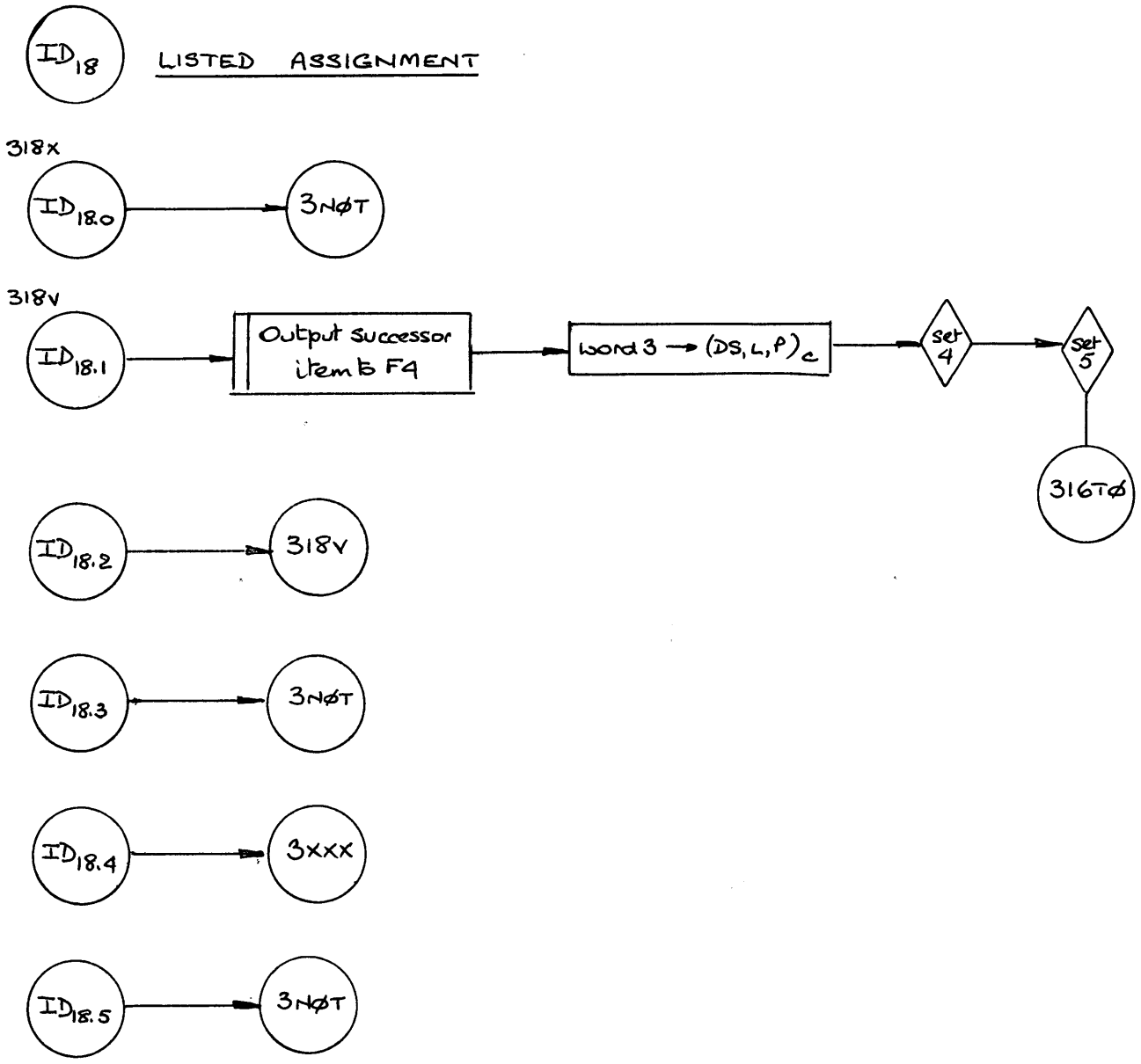


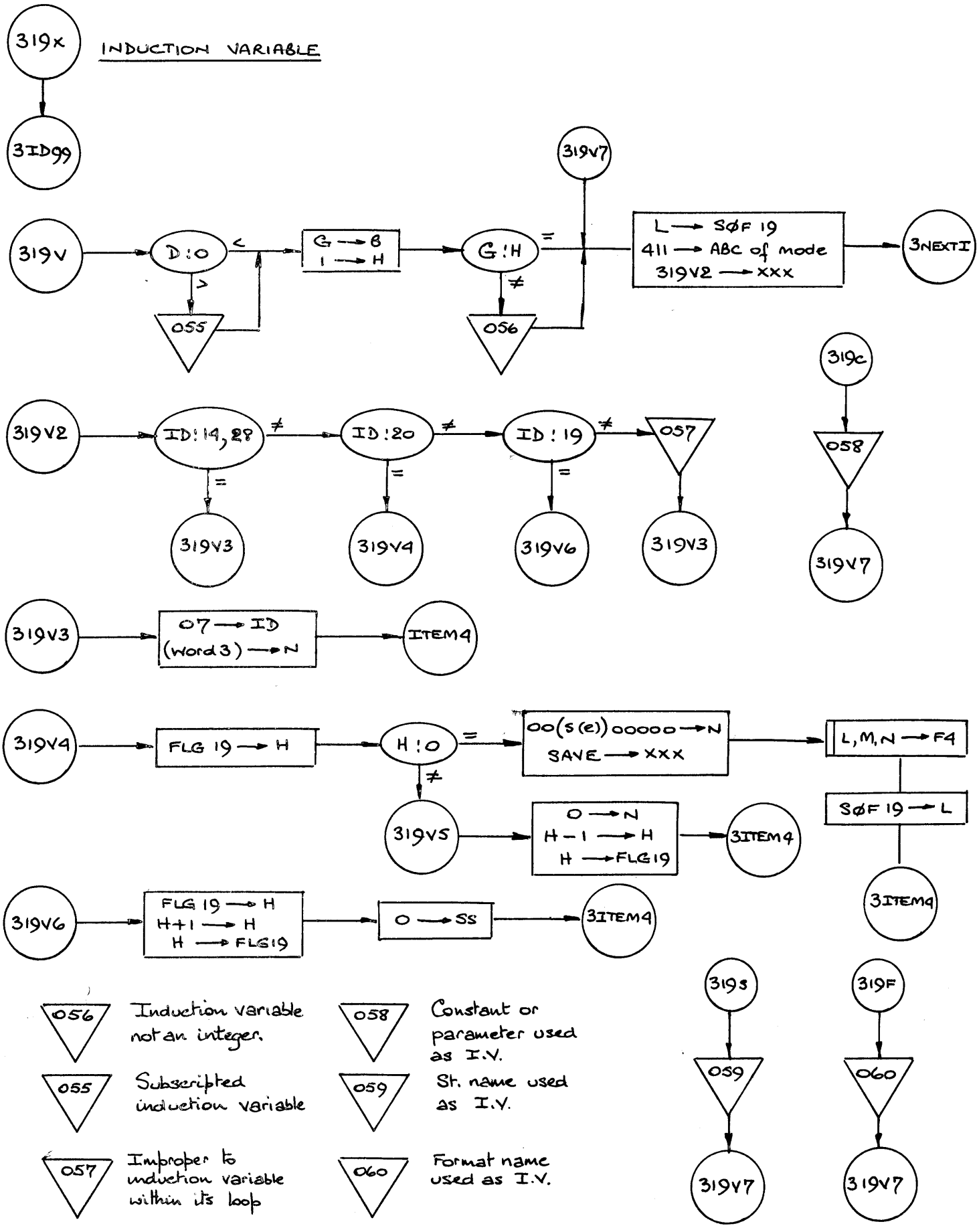



108 No assignment for variable

093 Assignment made to constant.

092 Assignment made to parameter.


ID18
LISTED ASSIGNMENT
318X
ID18.0
3NØT
318V
ID18.1
Output Successor
item b F4
word 3 → (DS, L, P)c
set 4
set 5
316TØ
ID18.2
318V
ID18.3
3NØT
ID18.4
3XXX
ID18.5
3NØT


319x
INDUCTION VARIABLE
3IDgg
319V
D:0
<
>
055
G → B
1 → H
G!H
=
≠
056
319V7
L → SØF 19
411 → ABC of mode
319V2 → XXX
3NEXTI
319V2
ID:14,28
≠
ID:20
≠
ID:19
≠
057
=
=
=
319V3
319V4
319V6
319V3
319c
058
319V7
319V3
07 → ID
(word3) → N
ITEM4
319V4
FLG 19 → H
H:0
=
≠
00(s(e))00000 → N
SAVE → XXX
L,M,N → F4
SØF 19 → L
3ITEM4
319V5
0 → N
H-1 → H
H → FLG19
3ITEM4
319V6
FLG 19 → H
H+1 → H
H → FLG19
0 → SS
3ITEM4
056
Induction variable not an integer.
055
Subscripted induction variable
057
Improper to induction variable within its loop
058
Constant or parameter used as I.V.
059
St. name used as I.V.
060
Format name used as I.V.
319S
059
319V7
319F
060
319V7

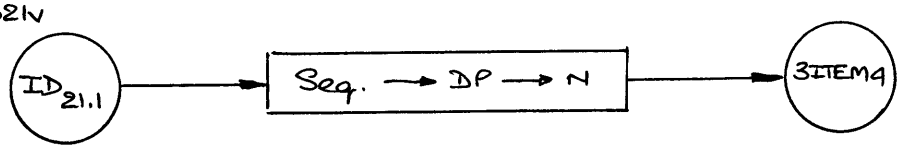



STORED UNSUBSCRIPTED VARIABLE

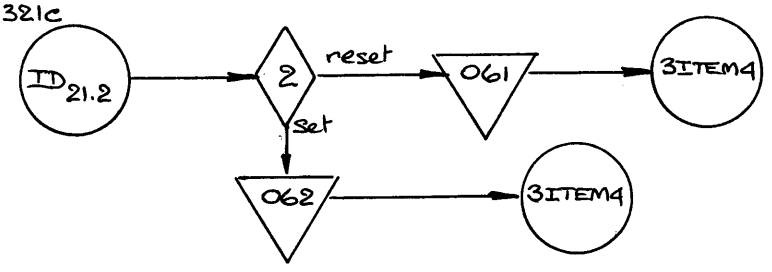
321X



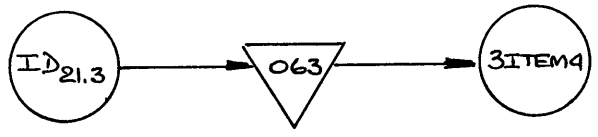

321V

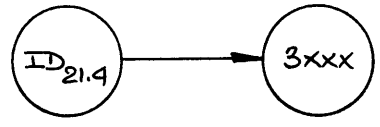



321C

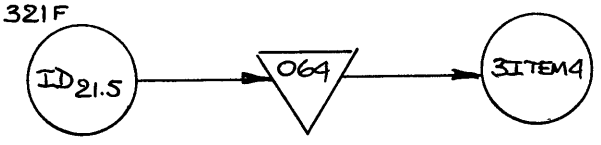



321S




321F







Parameter appears as store item




Statement name appears as store item.




Constant appears as store item.




Format name appears as store item.

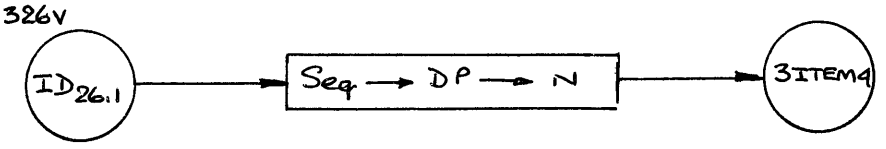



INPUT-LISTED UNSUBSCRIPTED VARIABLE

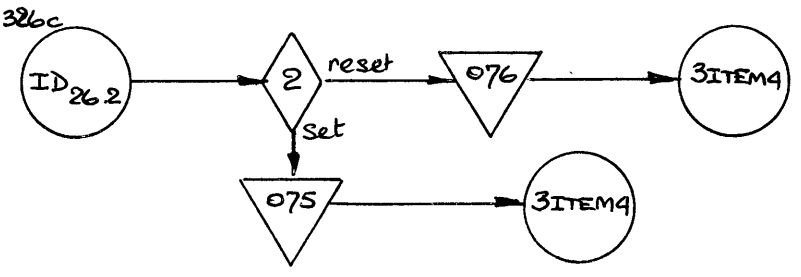
326X



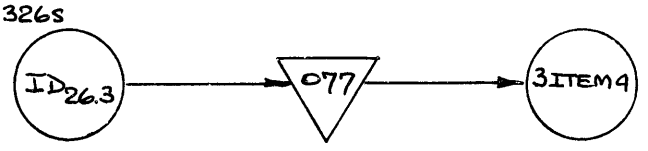

326V




326C



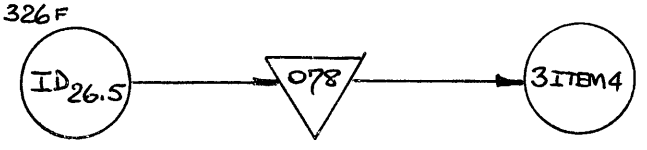

326S







326F







Parameter in input list



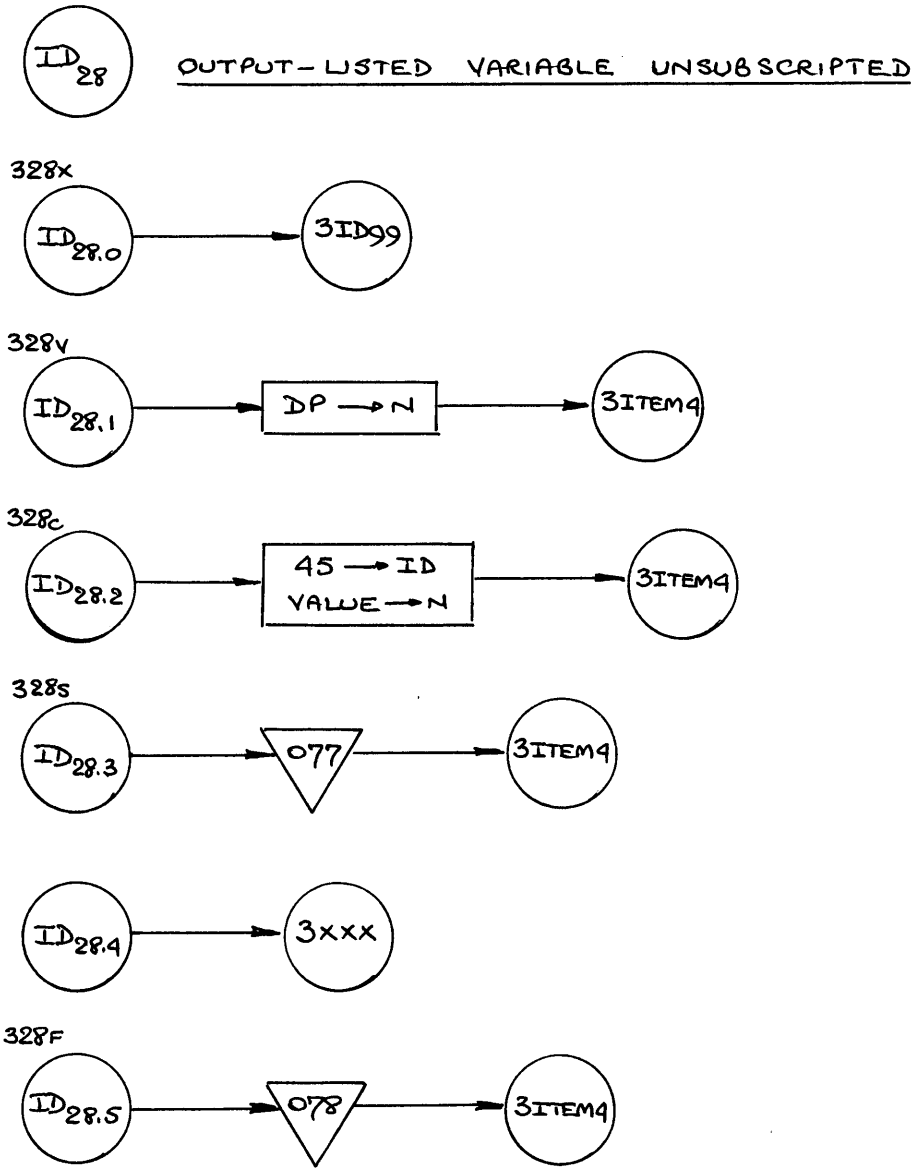

St. name in I/ϕ list




Constant in input list.




Format name in I/ϕ list.



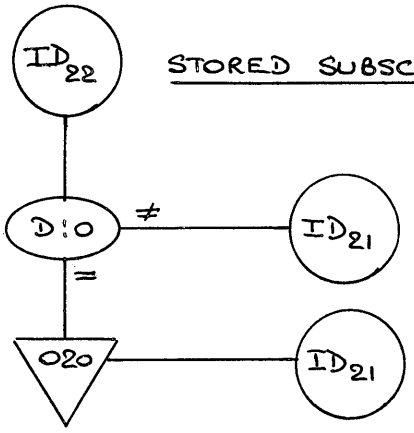




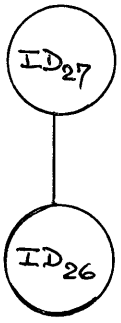
St. name in I/ø list.



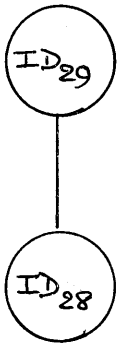

Format name in I/ø list.




STORED SUBSCRIPTED VARIABLE



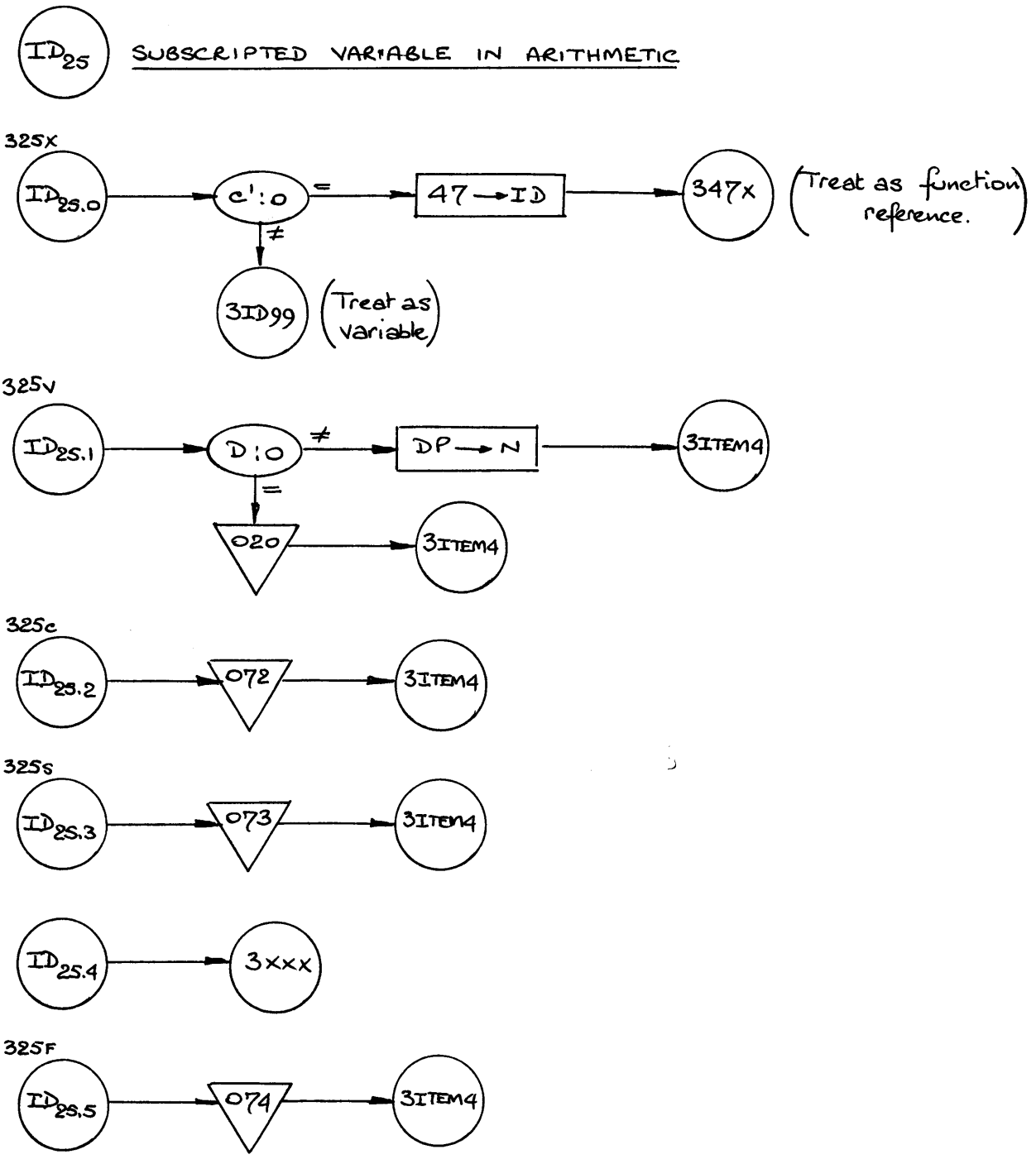

INPUT-LISTED SUBSCRIPTED VARIABLE



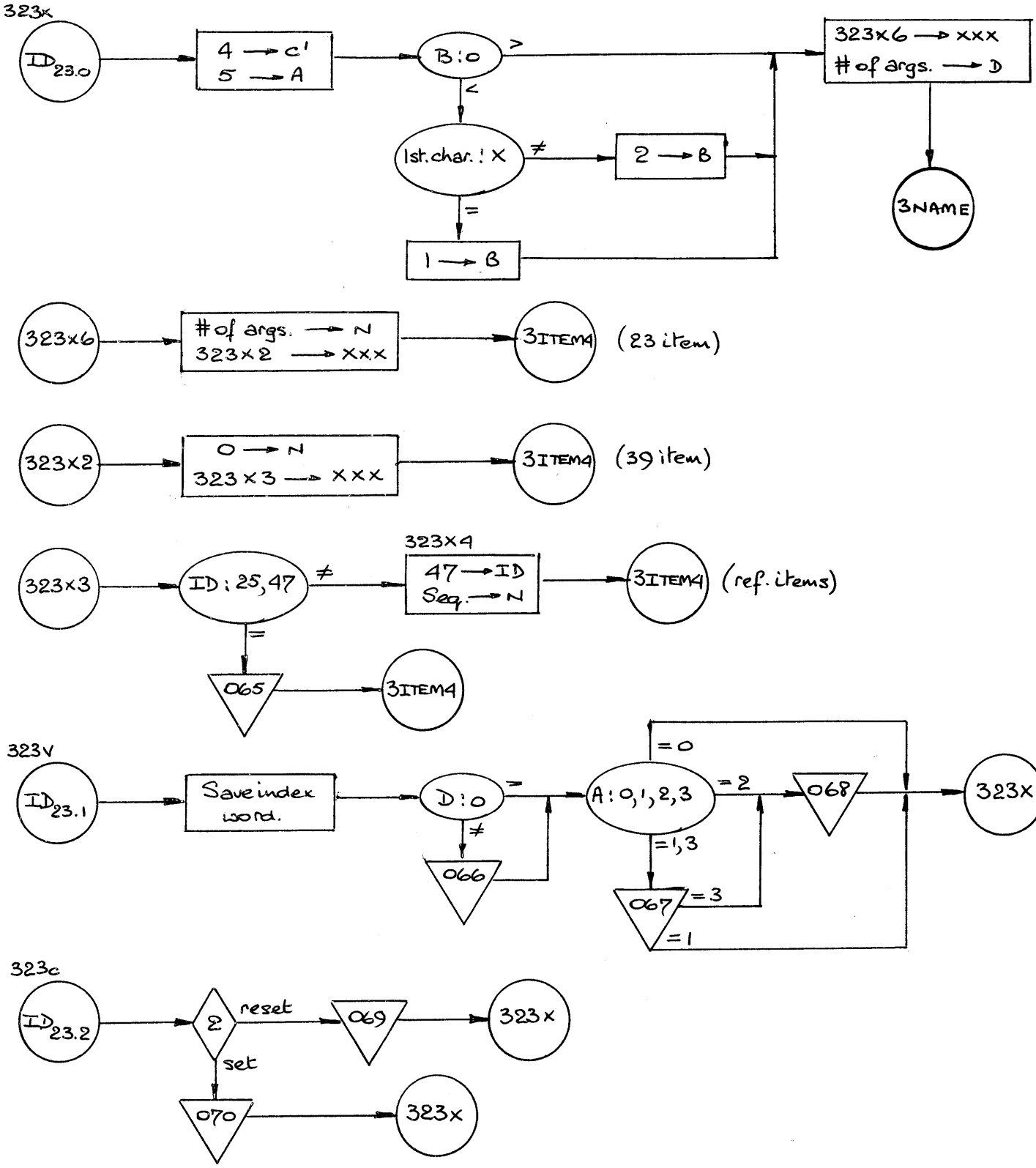

OUTPUT-LISTED SUBSCRIPTED VARIABLE




Subscripted reference to non-dimensioned variable.


ID25
SUBSCRIPTED VARIABLE IN ARITHMETIC
325X
ID25.0
C':0
=
47→ID
347X
(Treat as function reference.)
≠
3ID99
(Treat as variable)
325V
ID25.1
D:0
≠
DP→N
3ITEM4
=
020
3ITEM4
325C
ID25.2
072
3ITEM4
325S
ID25.3
073
3ITEM4
ID25.4
3XXX
325F
ID25.5
074
3ITEM4
072
Subscripted reference to parameter
074
Subscripted reference to format name.
073
Subscripted reference to statement name

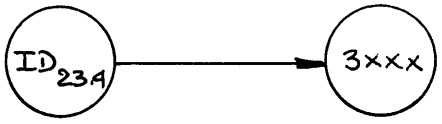
ID23 ARITHMETIC STATEMENT FUNCTION page 1






## ARITHMETIC STATEMENT FUNCTION page 2.










Improper reference to arithmetic st. function.



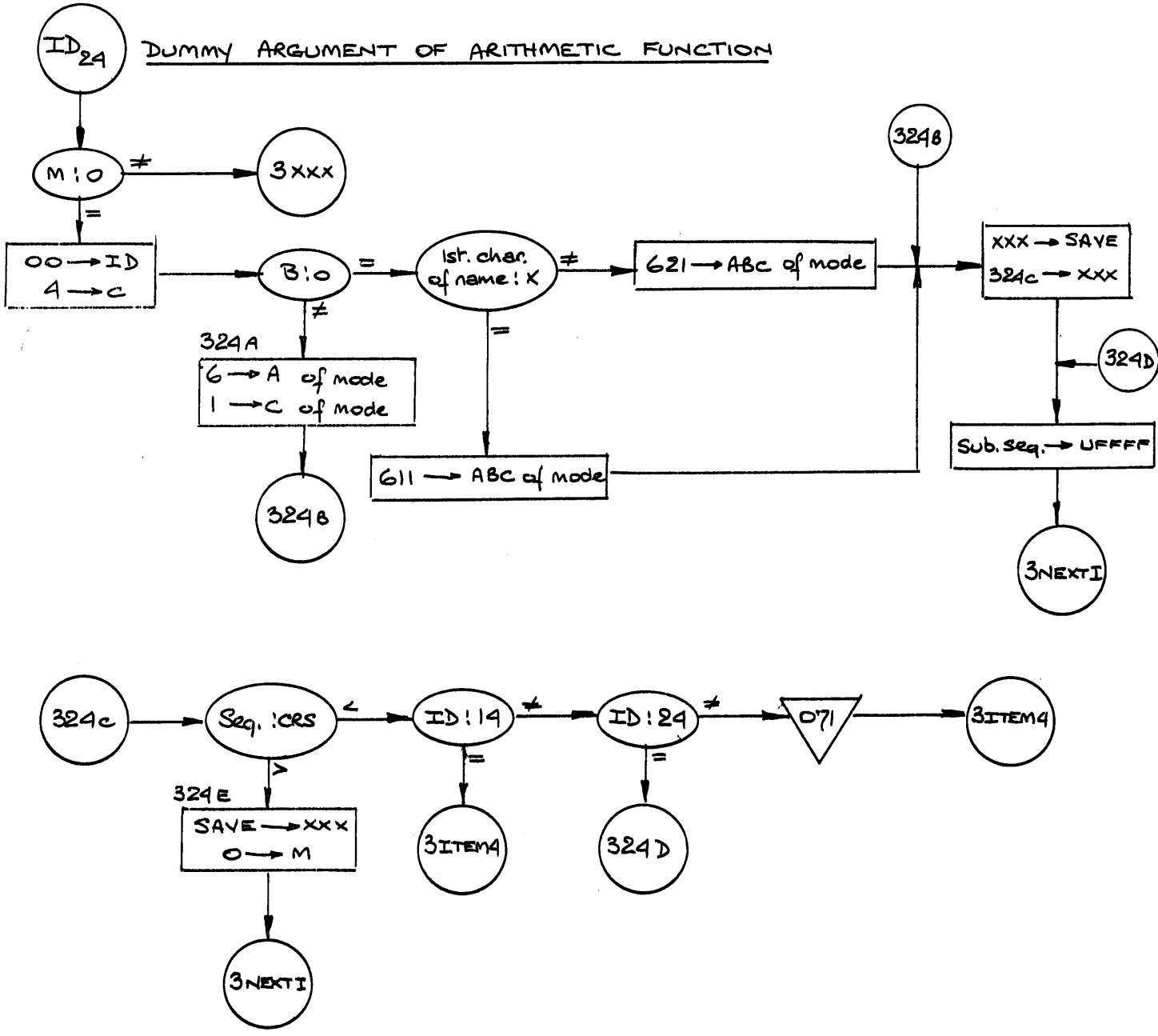

Arithmetic function name in dimension st.



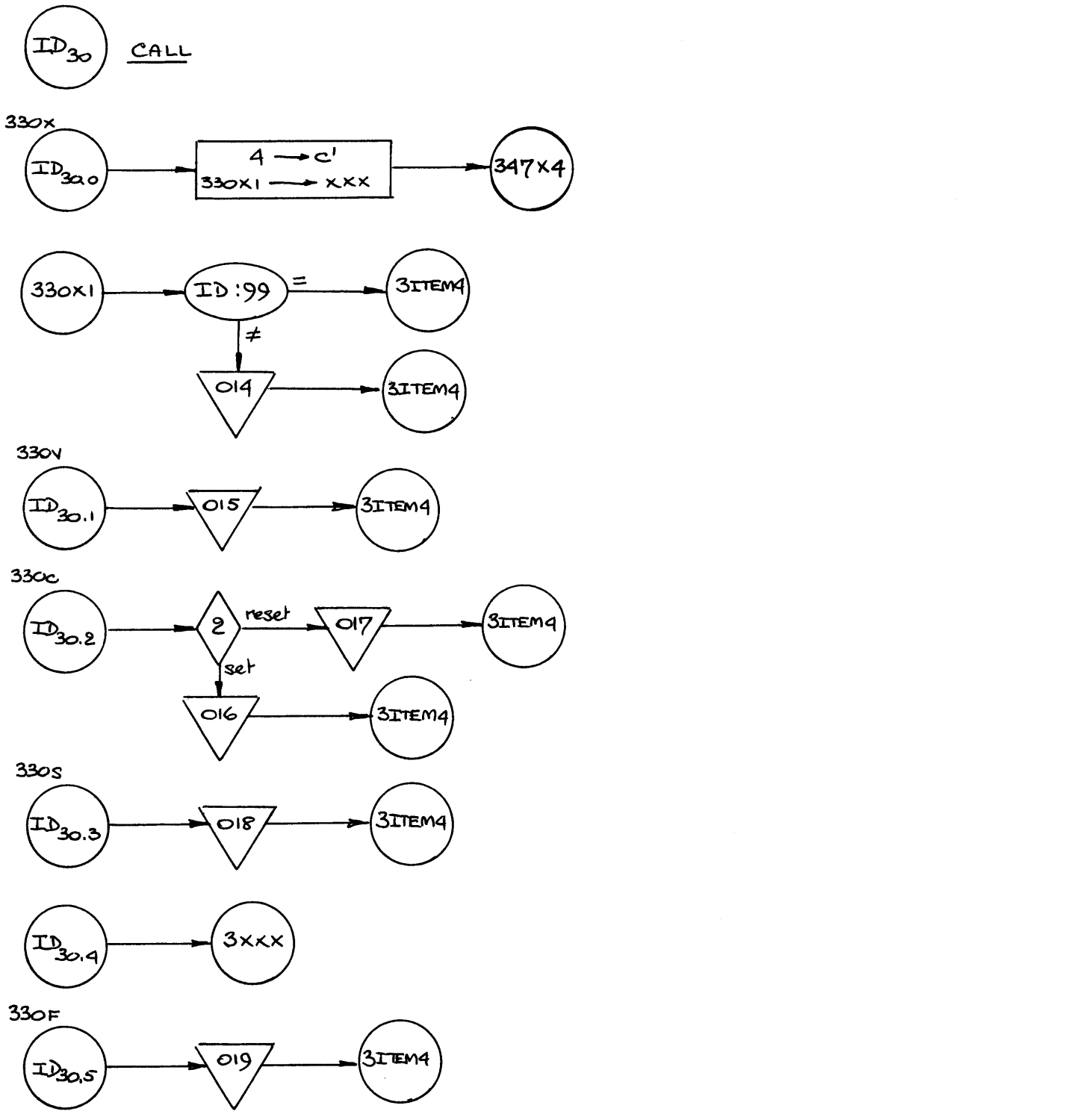

Arithmetic function name in common st.

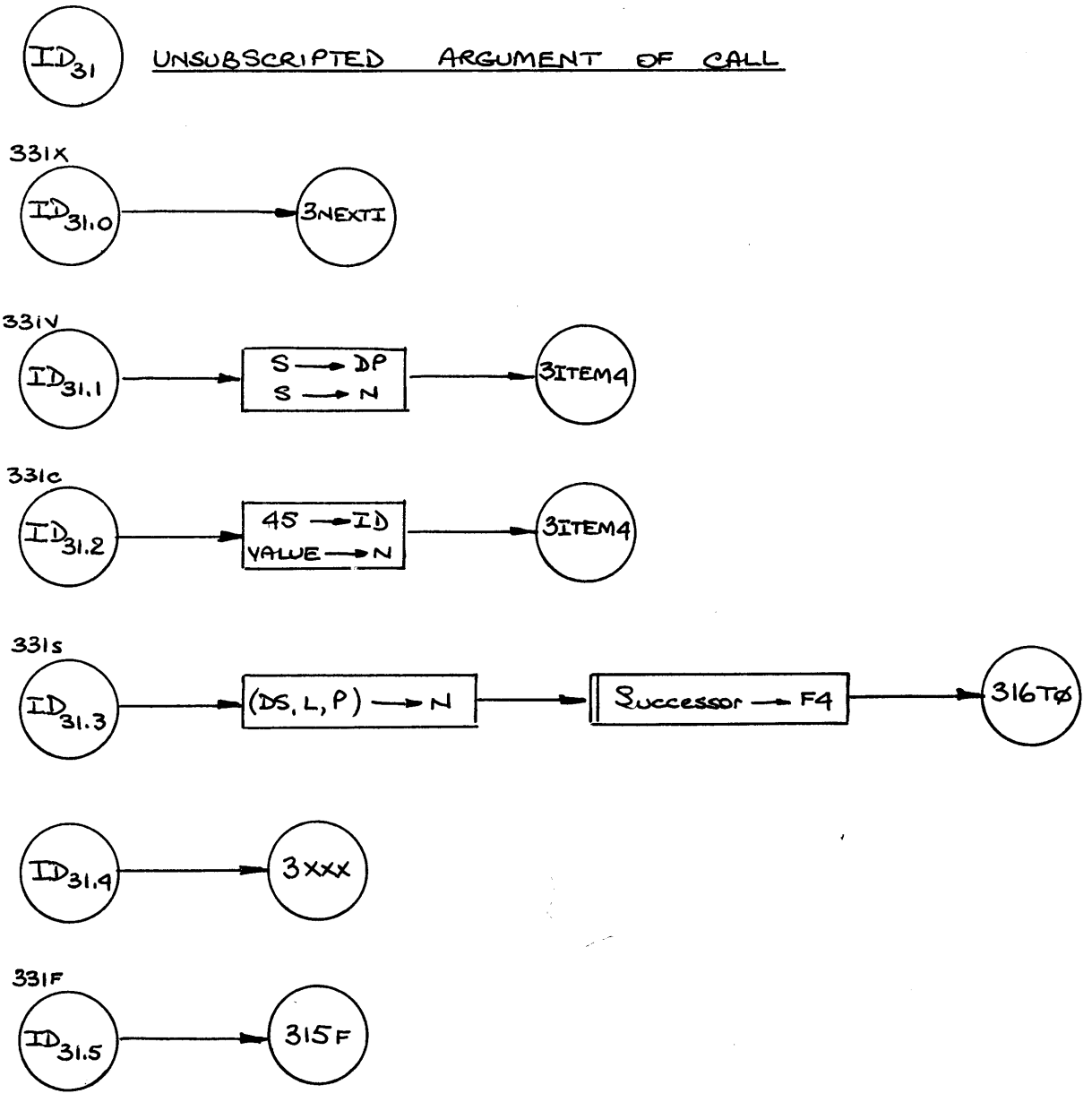


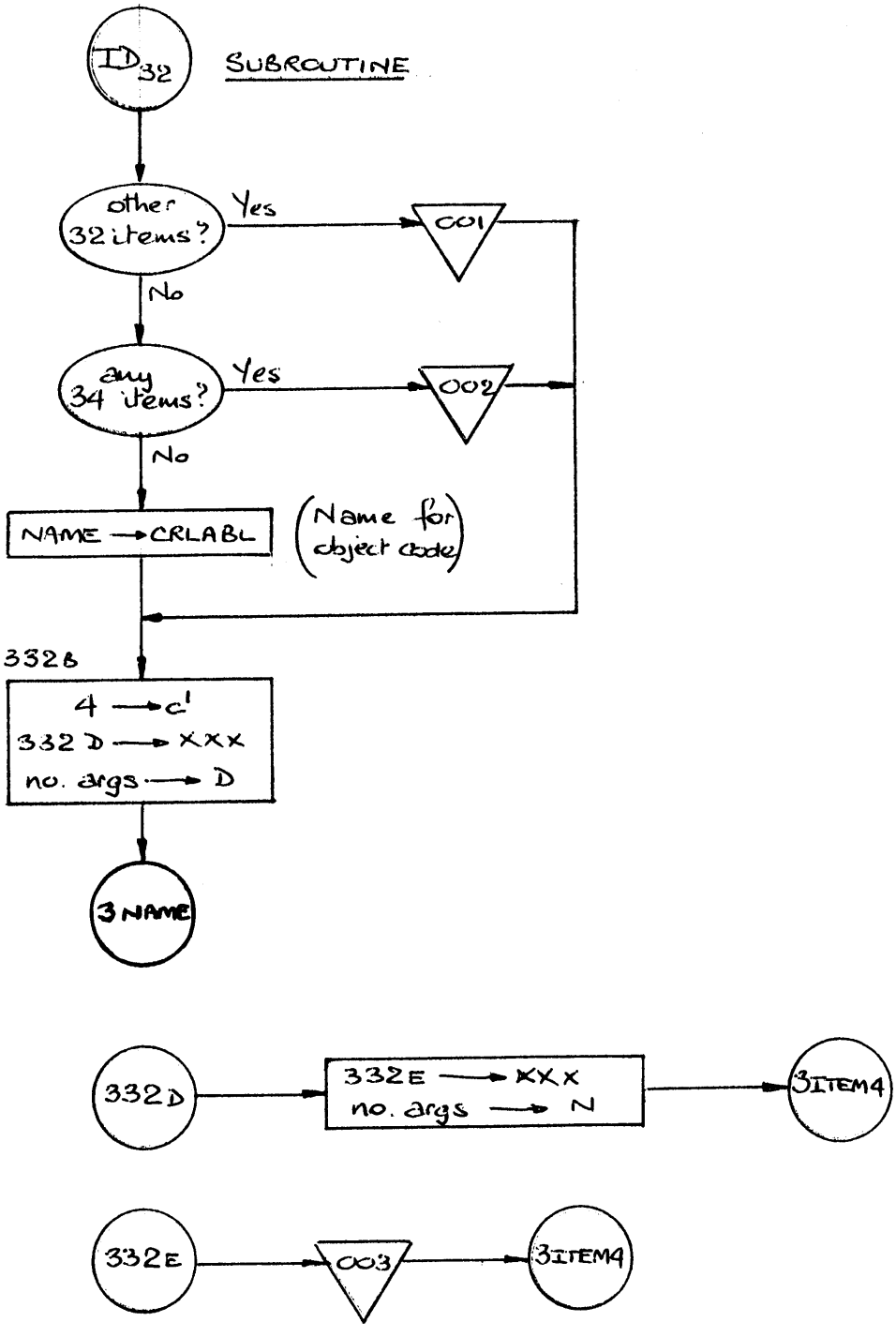

Arithmetic function name in equivalence st.




071 Improper use of dummy argument of arithmetic function.


ID30 CALL
330X
ID30.0
4 → C'
330X1 → XXX
347X4
330X1
ID:99
=
3ITEM4
≠
014
3ITEM4
330V
ID30.1
015
3ITEM4
330C
ID30.2
2
reset
017
3ITEM4
set
016
3ITEM4
330S
ID30.3
018
3ITEM4
ID30.4
3XXX
330F
ID30.5
019
3ITEM4
014 Improper use of called subroutine
016 Constant referenced as called subroutine
018 St. name referenced as called subroutine
015 Variable referenced as called subroutine
017 Parameter referenced as called subroutine
019 Format referenced as called subroutine


ID31
UNSUBSCRIPTED ARGUMENT OF CALL
331X
ID31.0
3NEXTI
331V
ID31.1
S → DP
S → N
3ITEM4
331C
ID31.2
45 → ID
VALUE → N
3ITEM4
331S
ID31.3
(DS, L, P) → N
Successor → F4
316TØ
ID31.4
3XXX
331F
ID31.5
315F







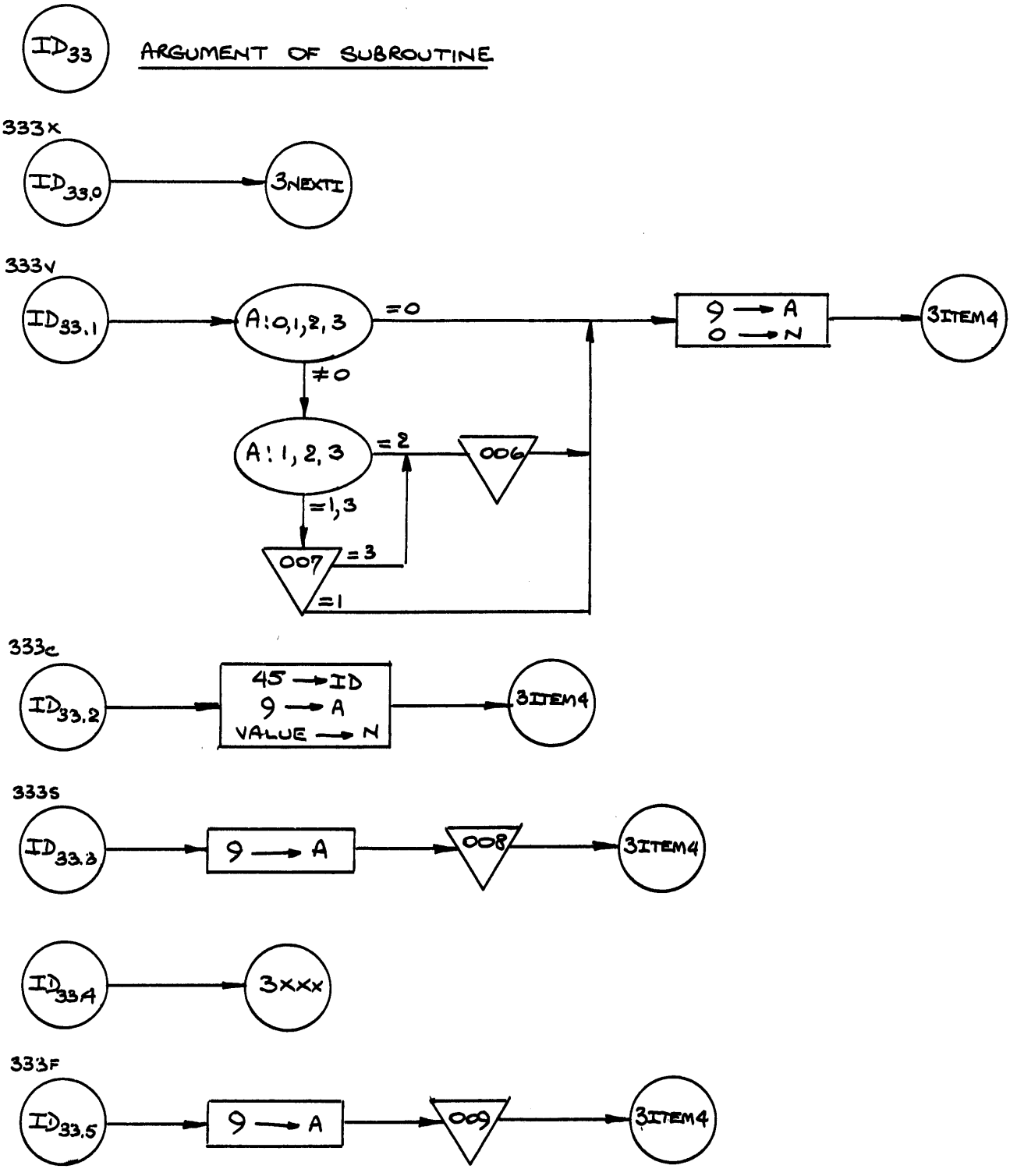
Multiple subroutine statements.




Subroutine and function statement.




Name of subroutine used elsewhere in program.

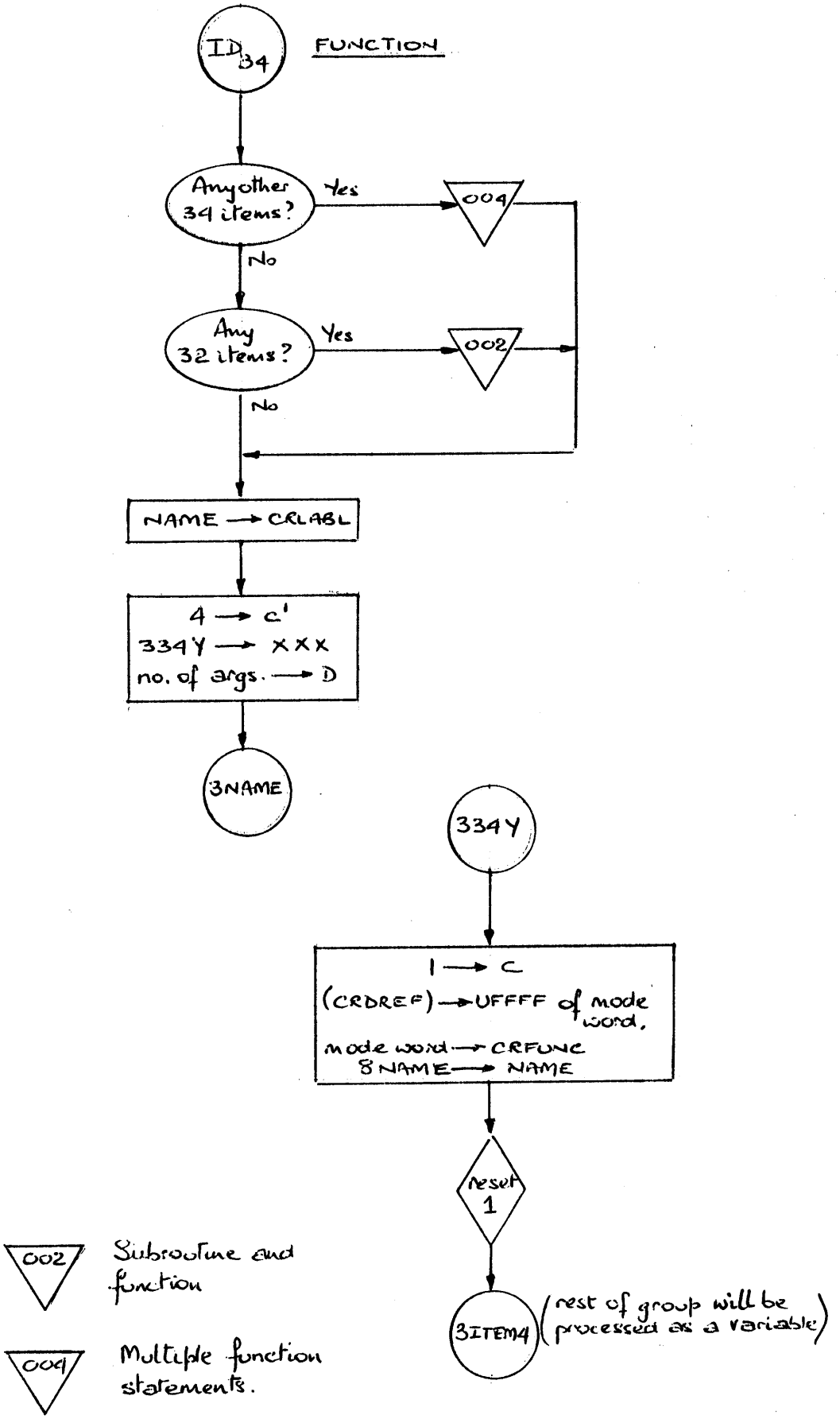


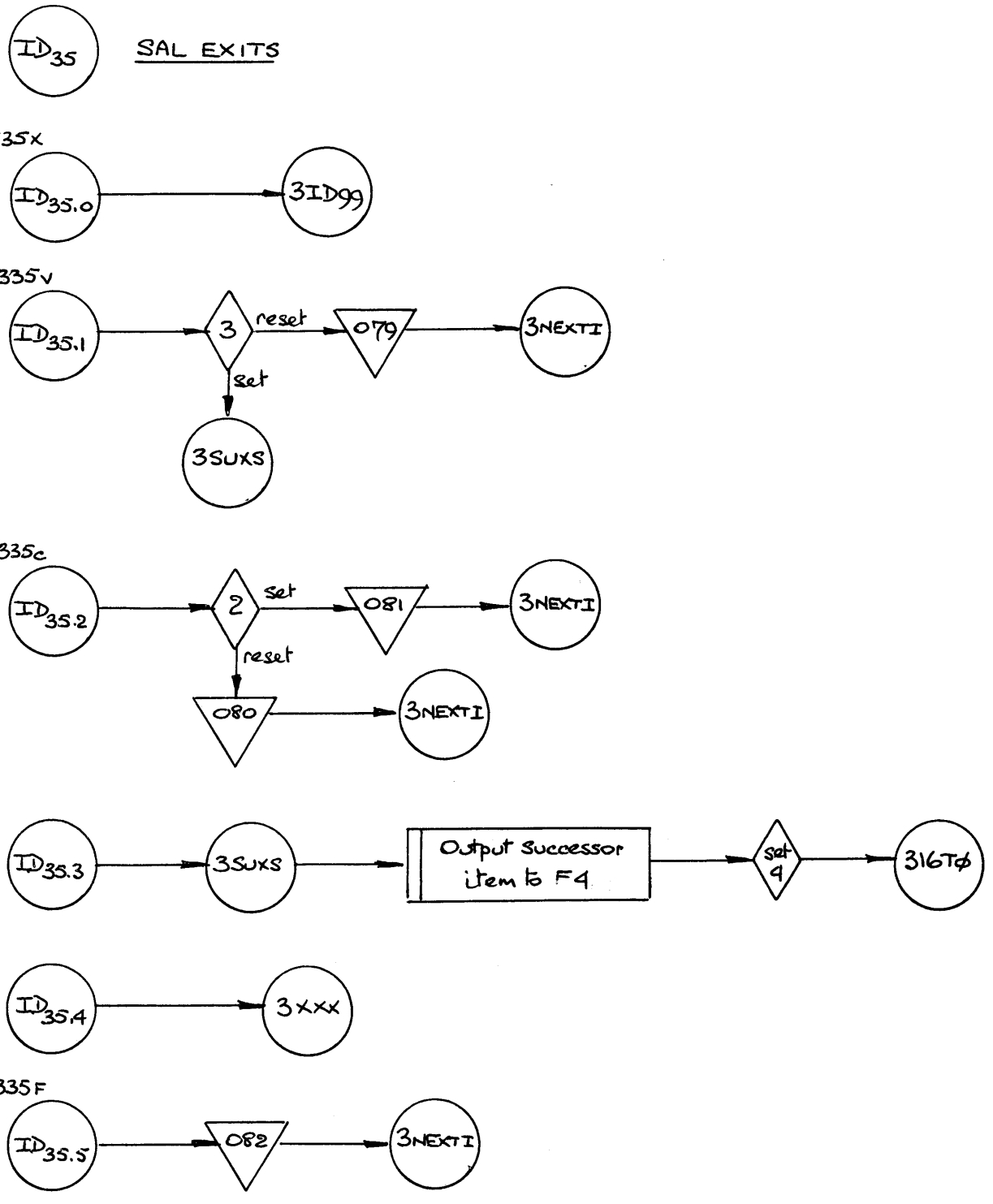

006 Argument of subroutine in common

007 Argument of subroutine in equivalence.

008 Argument of subroutine is statement name

009 Argument of subroutine is format name.


ID 34
FUNCTION
Any other 34 items?
Yes
004
No
Any 32 items?
Yes
002
No
NAME → CRLABL
4 → c'
334Y → XXX
no. of args. → D
3NAME
334Y
1 → C
(CRDREF) → UFFFF of mode word.
mode word → CRFUNC
& NAME → NAME
reset 1
3ITEM4
(rest of group will be processed as a variable)
002
Subroutine and function
004
Multiple function statements.

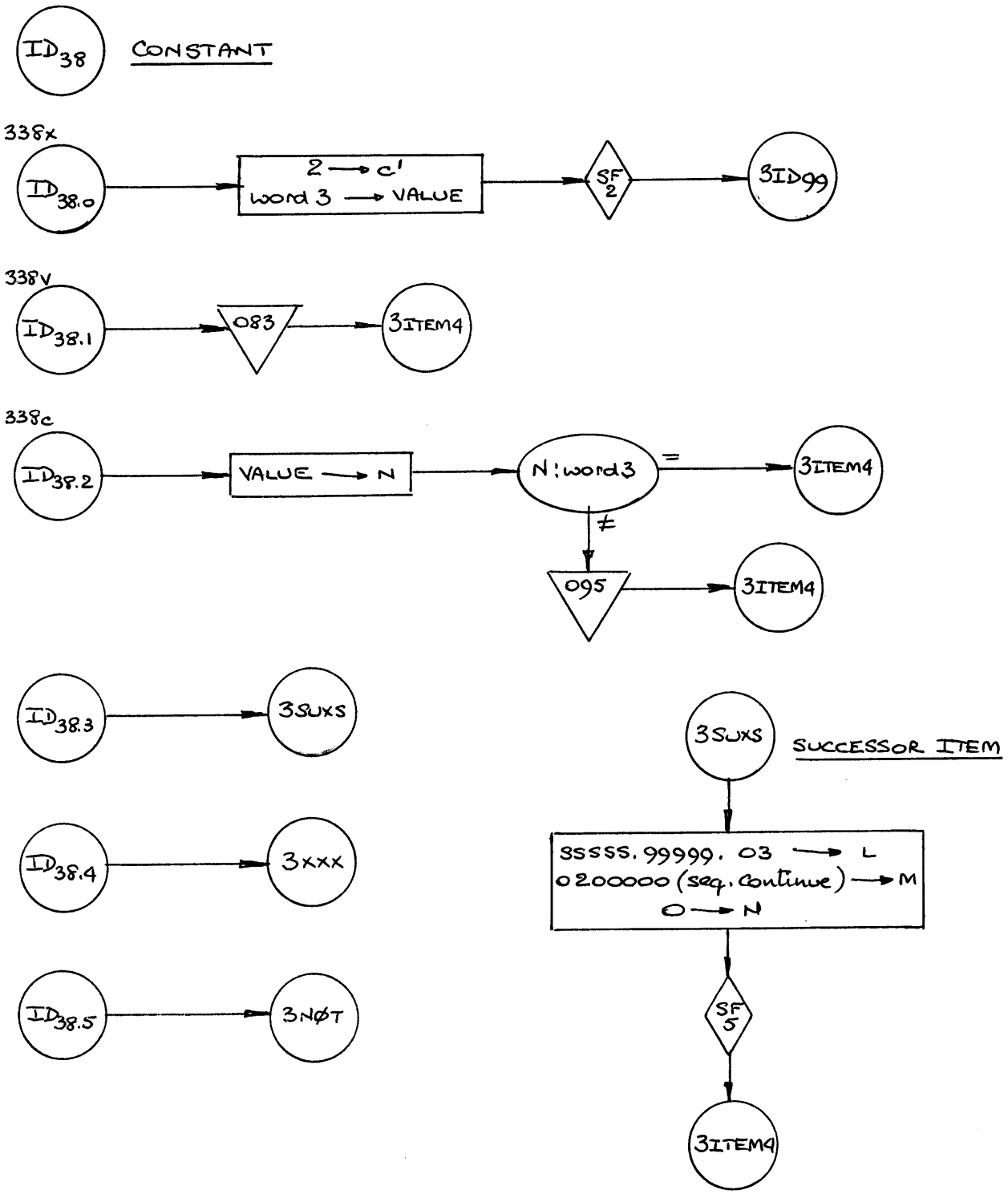


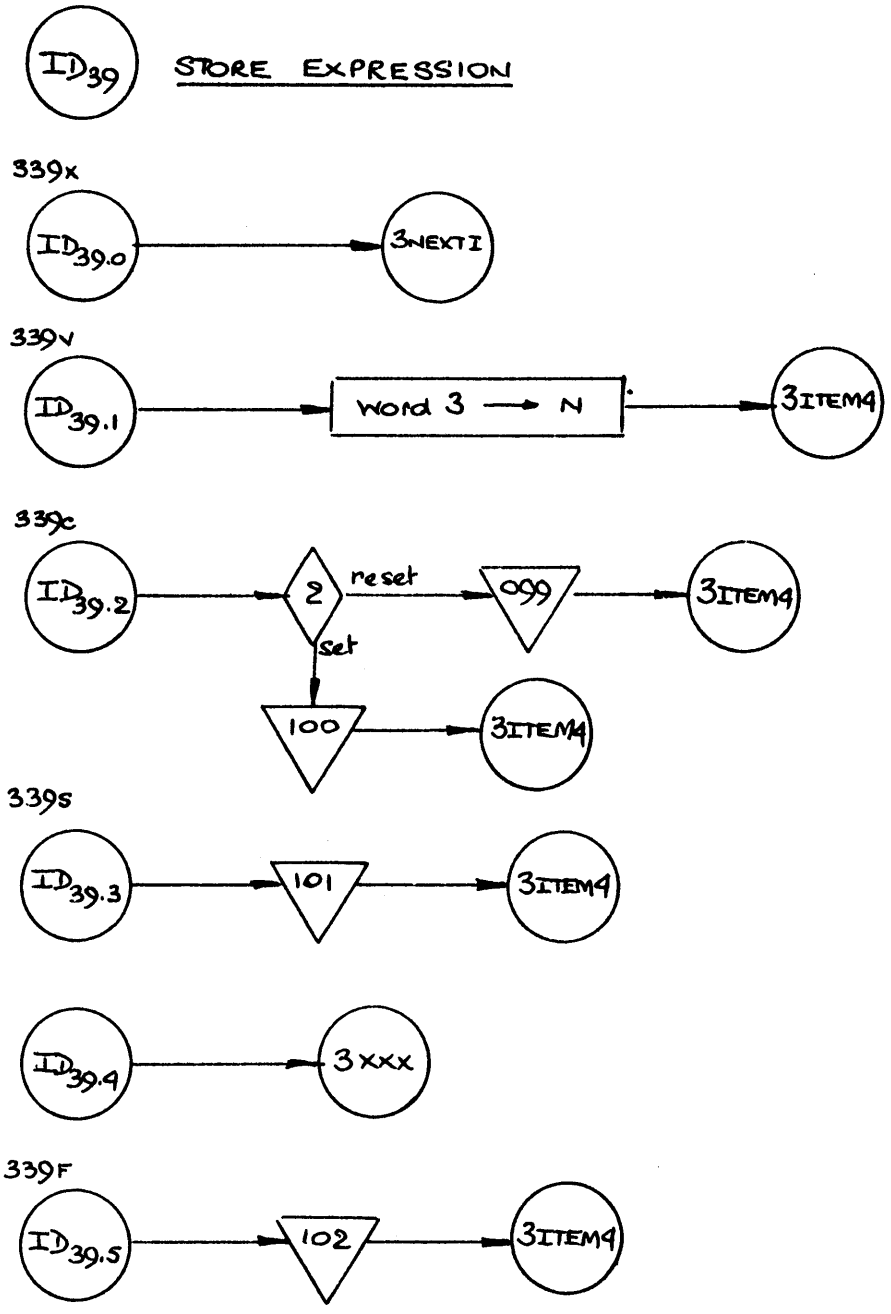

079 Unassigned variable in SAL exits list.

081 Constant in SAL exits list.

080 Parameter in SAL exits list.

082 Format in SAL exits list.


ID38
CONSTANT
338x
ID38.0
2 → c'
word 3 → VALUE
SF 2
3ID99
338v
ID38.1
083
3ITEM4
338c
ID38.2
VALUE → N
N:word3
=
3ITEM4
≠
095
3ITEM4
ID38.3
3SUXS
ID38.4
3XXX
ID38.5
3NØT
3SUXS
SUCCESSOR ITEM
55555.99999.03 → L
0200000 (seq. continue) → M
0 → N
SF 5
3ITEM4
083
Variable and constant have same name.
095
Multiple defining items for constant.




099 Parameter and expression have same name.

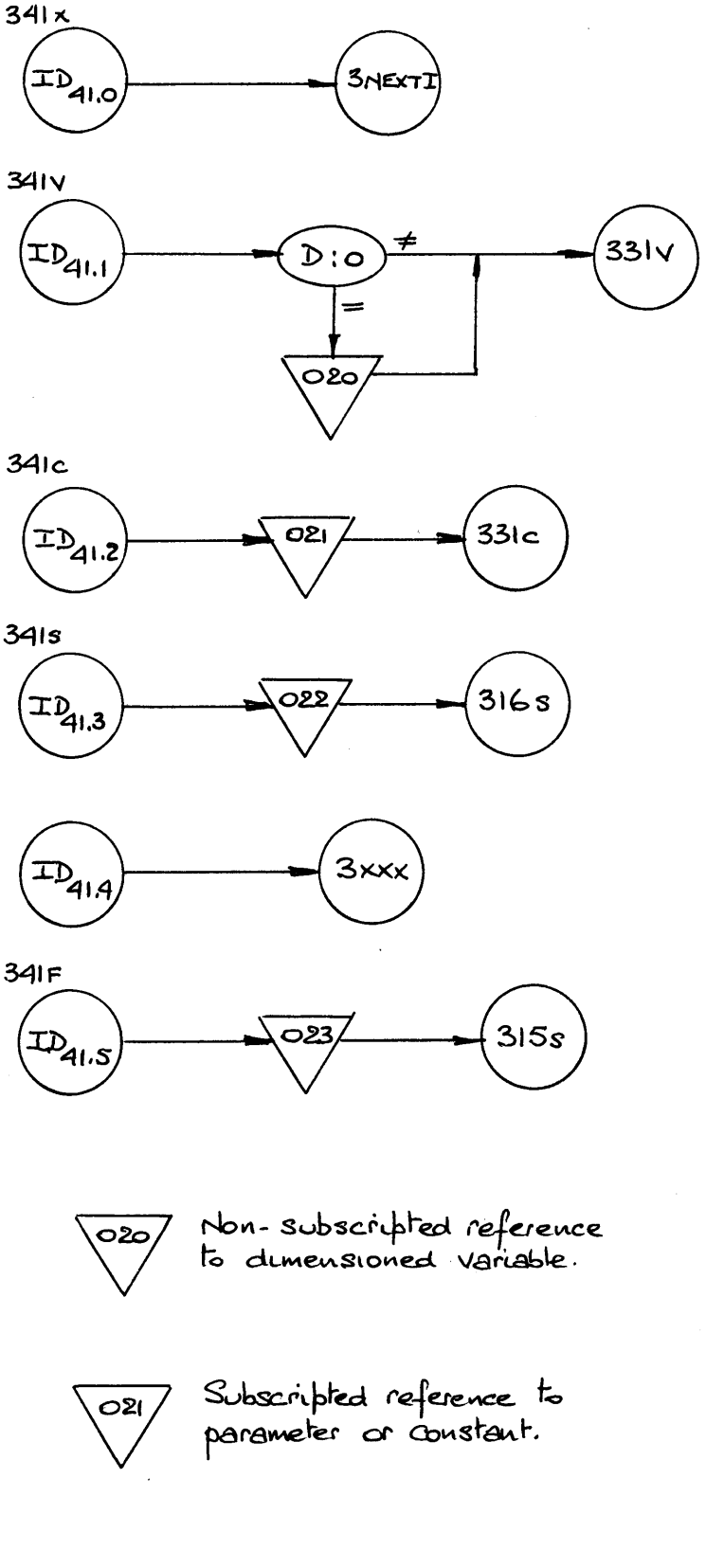
101 St. name and expression have same name

100 Constant and expression have same name

102 Format and expression have same name.




SUBSCRIPTED ARGUMENT OF CALL

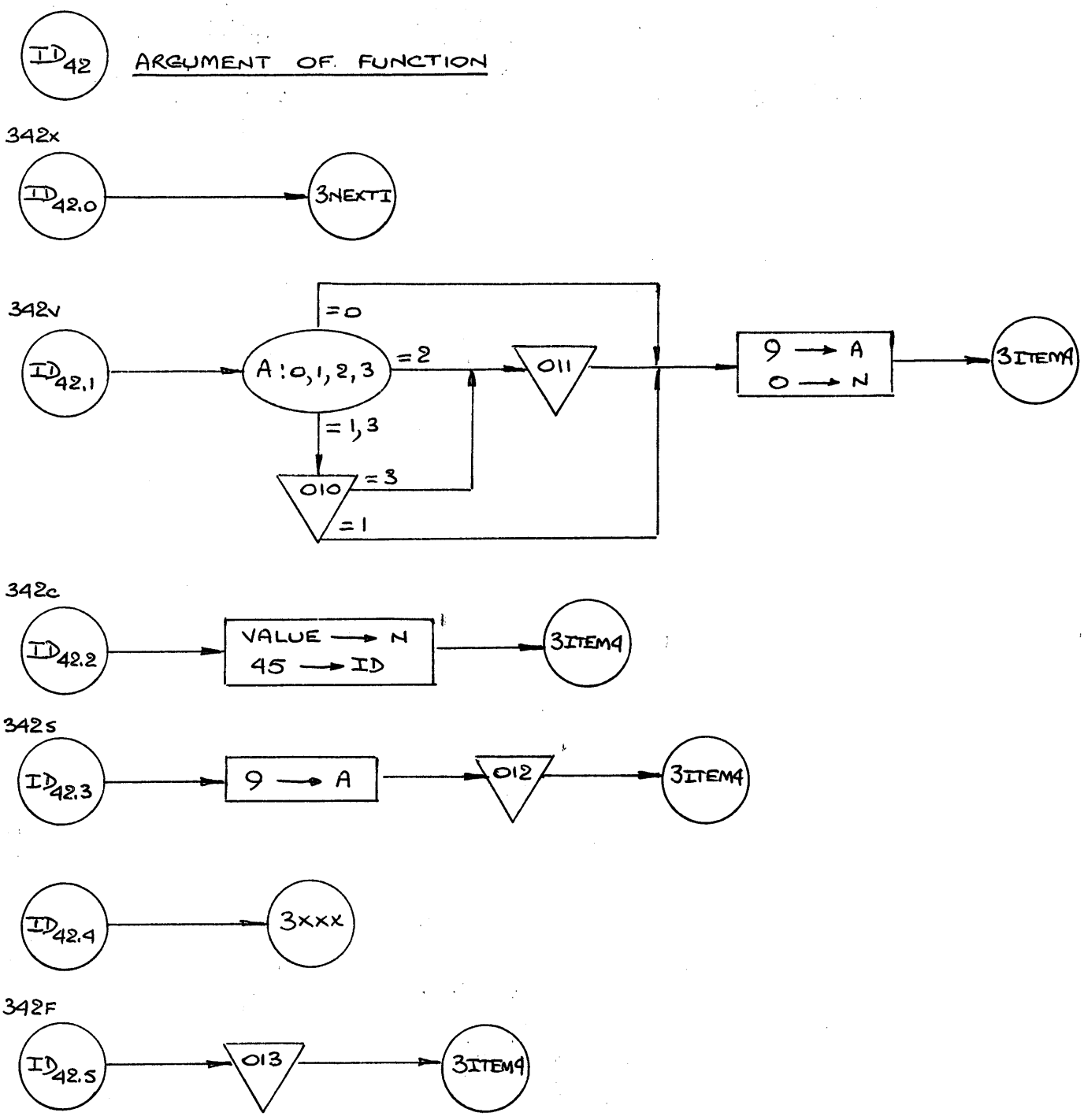



020 Non-subscripted reference to dimensioned variable.

022 Subscripted reference to statement name.

021 Subscripted reference to parameter or constant.

023 Subscripted reference to format name.

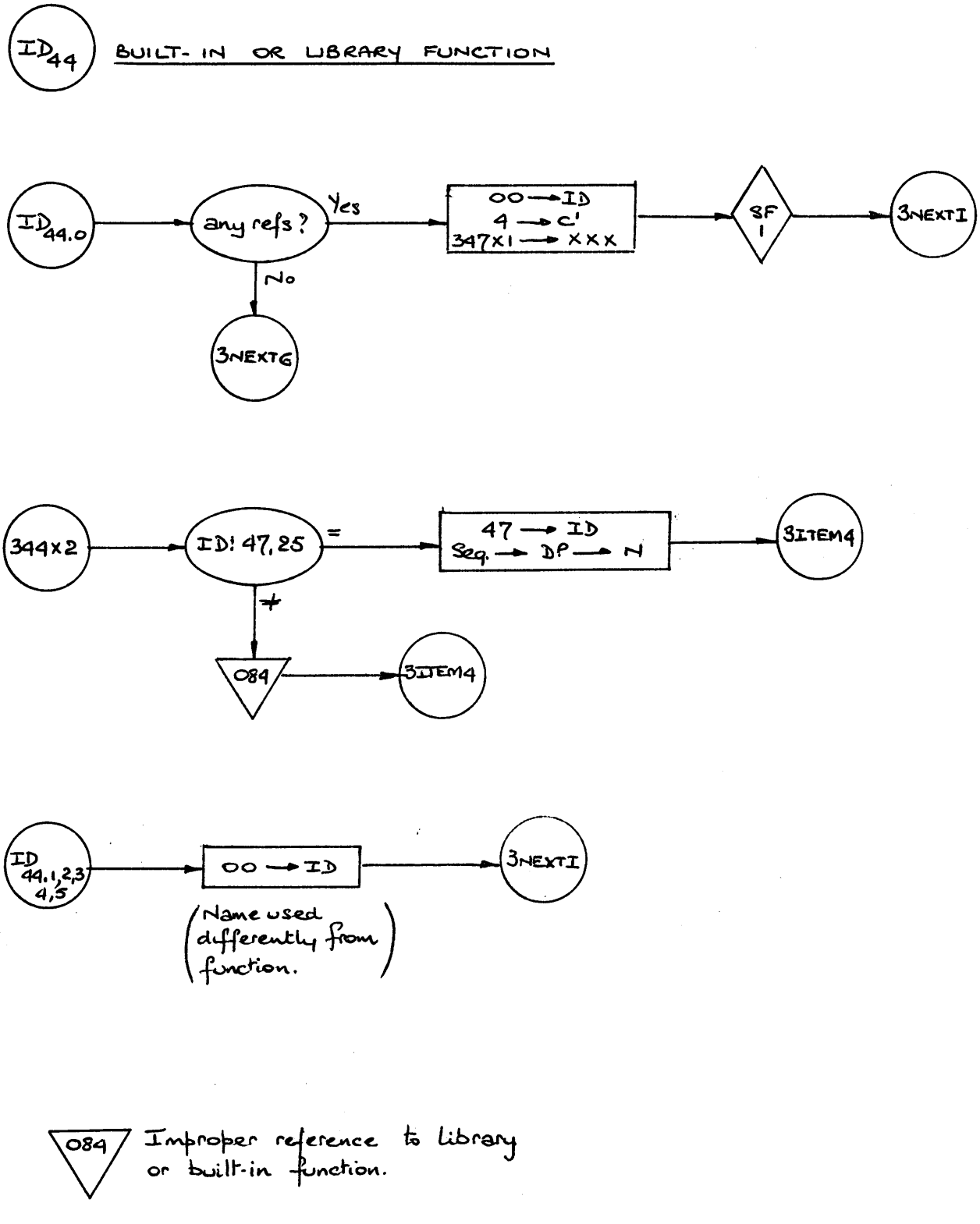



010 Argument of function in common statement.

012 Argument of function is statement name.

011 Argument of function in dimension statement.

013 Argument of function is format name.


ID44
BUILT-IN OR LIBRARY FUNCTION
ID44.0
any refs?
Yes
No
00 → ID
4 → C'
347X1 → XXX
SF 1
3NEXTI
3NEXTG
344X2
ID! 47,25
=
+
47 → ID
Seq. → DP → N
3ITEM4
084
3ITEM4
ID 44.1,2,3 4,5
00 → ID
3NEXTI
(Name used differently from function.)
084
Improper reference to library or built-in function.

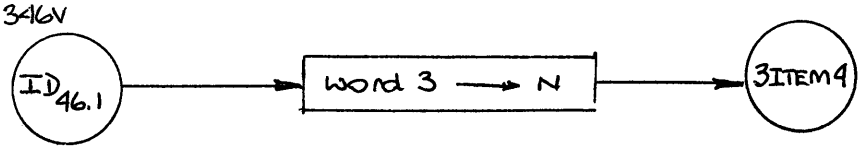



LIMIT OF DØ

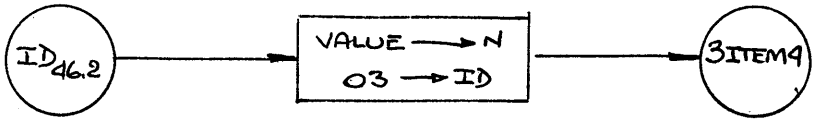
346x



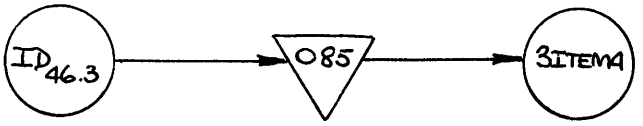

346V

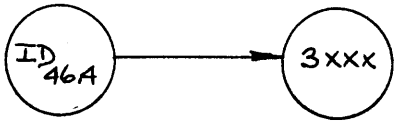



346c

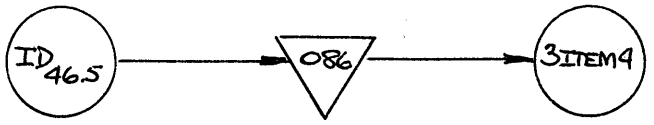



346s




346F



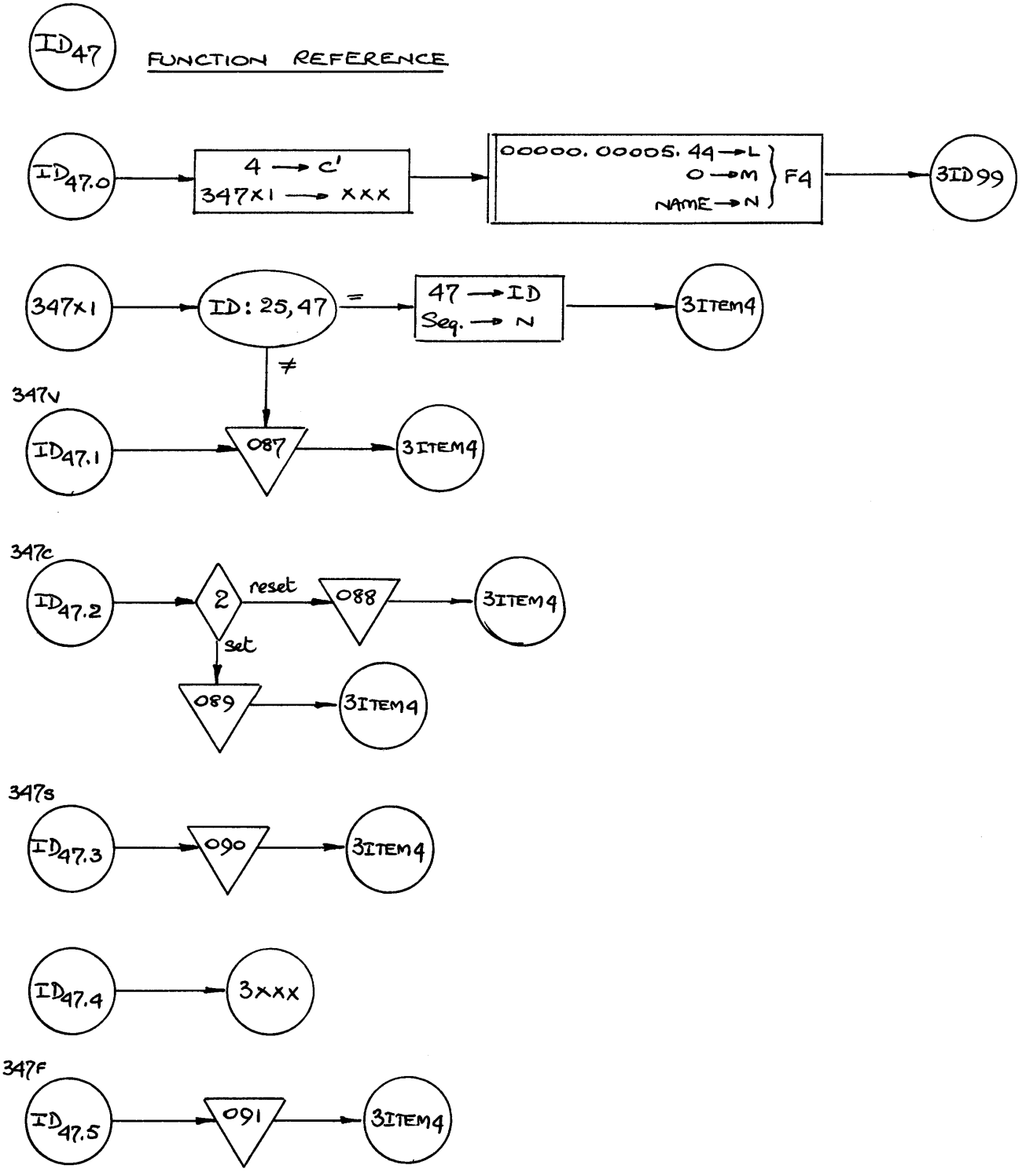




St. name used as Limit of DØ




Format used as Limit of DØ




087 Improper reference to function

089 Constant referenced as function.

091 Format name referenced as function.

088 Parameter referenced as function.

090 St. name referenced as function.

# PHASE IV

Phase IV performs the following major tasks.

1. Determines relative locations of all variables in memory.

2. Solves the system of linear equations provided by EQUIVALENCE statements.

3. Determines the mode of arithmetic expressions and attaches it to operations and functions within the expression.

4. Updates the definition points of common and equivalent variables.

5. Determines and forwards to Phase V a great deal of information about DO-loops and lists.

6. Converts subscripts to single expressions.

7. Performs the numeric arithmetic in reasonably uncomplicated subscripts and splits or puts their terms into three parts.

8. Computes for Phase VII the number of elements in implicit arrays.

9. Creates all of Files 50, 51, 52 and 60, and part of Files 61 and 94. (Output to File 60 and File 61 is in F4 format. Appendices describe the other files).

10. Performs several error checks on the source-program-generated items.

File 4 contains nine classes of source language statements in the order below. (Within each class source language ordering is preserved, unless otherwise noted).

A. FUNCTION or SUBROUTINE statement if one is present.

B. LIBRARY calls

C. PARAMETER statements

D. One item for each variable not appearing in a DIMENSION, EQUIVALENCE, or COMMON statement, or as a dummy argument of a FUNCTION or SUBROUTINE statement; hereafter called "simple variable" items. (Alphabetic order).

E. DIMENSION statements.

F. EQUIVALENCE and COMMON statements.

G. CONSTANT statements.

H. Arithmetic statement functions.

I. Executable statements, including:

1. Arithmetic statements
2. Induction variable items
3. Control statements (IF's, GO TO's, input-output statements, etc.).
4. CALL statements.

The treatment of each class of statement by Phase IV is indicated below.

Phase IV operates on File 4, which is sorted on the first word of each item. The general form of the items in File 4 is:

| | | |
|---|---|---|
| Word 1: | Digits 1 - 5 | Sequence number |
| | Digits 6 - 10 | Subsequence number |
| | Digits 11 -12 | Item identification |
| Word 2: | Mode word for the item | |
| Word 3: | Special information depending on the item type. | |
| | a. | A numeric value (for a constant item). |
| or | b. | A "definition point" (D. P.) (for a variable). |
| or | c. | The level (for an induction variable reference). |
| | etc. | |

File 4 was formed by Phase I and Phase III. Items in File 4 that represent punctuation, arithmetic operators, numeric quantities in the source program, and executable LSC statements are filed by Phase I, the rest by Phase III. Phase III creates several new types of items for File 4, as well as substituting constant items for all parameter references. The new items conform to the above three-word format. In most cases, the item identification is that assigned by Phase I.

Output from Phase IV goes to Phases V, VI, and IX. When F4 is thoroughly processed, Phase IV operation is complete.

Note: Phase IV has its own contingency routine.

A. FUNCTION or SUBROUTINE Statement (if any).

Subprograms differ from main programs in that the memory locations of parameters to the subprogram cannot be known at compile time. Later phases of LSC produce coding to generate these locations from the parameters given in the main program's calling sequence.

Phase IV assists in this generative process by sending to File 94 items defining each parameter as 0. (See Appendix III for formats of items sent to File 94).

| | | |
|---|---|---|
| Example: | Source Statement: | FUNCTION F(X, Y) |
| | Object Statements: | DEF X: 0 |
| | | DEF Y: 0 |

B. LIBRARY CALLS

A library call item is output to F94 for each item.

C. PARAMETER Statements.

Phase IV sends a raw-code item to File 94 for each parameter.

| | | |
|---|---|---|
| Example: | Source Statement: | PARAMETER N=30 |
| | Object Statement: | DEF N: 30 |

Since the value of a parameter is substituted for its symbolic name throughout the program, the definition of parameters is not necessary; it is added as a convenience to users who may want to refer to the parameter in SAL coding embedded in the LSC code.

## D. SIMPLE VARIABLES .

Phase IV makes storage assignments for all variables mentioned in the source program (except CONSTANT's, subprogram parameters, and COMMON variables in subprograms).

The storage is reserved by Phase IX. Phase IV sends out an item giving the total size of COMMON and of non-COMMON data. (The COMMON item is not output for subprograms).

Each variable is then defined by a DEF item to fall in COMMON or non-COMMON storage, whichever is appropriate.

1. DIMENSION statements produce DEF's for non-equivalent non-common arrays.

2. EQUIVALENCE statements produce DEF's for all equivalent variables.

3. COMMON statements produce DEF's for all non-equivalent common variables.

4. The remaining variables are defined by Phase IV on receipt of a group of "simple variable" items from Phase III. The "simple variables" are defined in alphabetic order and are in alphabetic order in memory when the program is run.

The leading statements of the object code thus provide the user with a complete storage map for variables in his program.

## E. DIMENSION STATEMENTS

Phase III places in the mode word of each array item its relative dimension table reference. Phase IV creates the dimension table and files all dimensions in it for use in later Phase IV processing.

DIMENSION statements also have reservative force. DEF's for "ordinary" arrays go to FILE 94. Also, each array name is defined in terms of an LSC-created symbol which represents the actual location of the array in storage, again by F 94 items.

Note: The location of the array contains its first element (1, . . . ,1) while the name of the array refers to the non-existent element (0, . . . ,0); that is, the array X begins in the storage location (1, . . . ,1) beyond the location named X. This substitution of variable is done in Phase IV to speed up later processing and enhance the readability of the LSC output.

## F. EQUIVALENCE and COMMON STATEMENTS

These two types of statements are intermixed. Since EQUIVALENCE statements cannot be processed until all the EQUIVALENCE relations are known, the subroutine which processed COMMON variables is called by the EQUIVALENCE processor.

### 1. Phase IV Treatment of EQUIVALENCE Statements

Each statement contains one or more chains of variables. A set of interrelated chains is here called a class. The processing as as follows:

a. Two internal files are created, one containing an item for each chain; the other, an item for each variable.

b. The terms of the subscript of each variable are multiplied by its dimensions to yield a single subscript. (If any term is zero, 1 is substituted and an error is output). The dimensions of the variable are multiplied to yield its total size. If the variable is COMMON, the chain is marked COMMON.

c. All previously processed variables are searched. If a match is found in a chain of the same class, the current item is either redundant or in error. The two subscripts are compared for error and the current item is dropped. If a match is found in a different class, the difference of the two subscripts is used to "relativize" all chains of the later class to the earlier class.

d. After all variables have been scanned, the subscripts within each class are relativized to a mutual point. The largest subscript and the largest inverse subscript (distance from the mutual point to the end of the array) are saved.

e. During the final scan, Phase IV computes the size of each class (largest subscript + largest inverse subscript) and the start of each class. It files in F94 an item for each variable giving

its category (COMMON or not) and its distance from the beginning of its category

2. Phase IV Treatment of COMMON statements

Variables which appear in EQUIVALENCE statements are ignored (they have been processed already). DEF items are filed in F94 for the others.

G. CONSTANT Statements

Both halves of the constant and its mode word are forwarded to F94 for output.

H. Arithmetic Statement Functions

Phase IV ignores the left sides of arithmetic statement functions. The right side begins with a mode-of-expression item and it's analyzed by the Expression Analysis Subroutine EXAN described in Appendix I.

While subscripts are permissible in ordinary arithmetic statements, they are not in arithmetic statement functions; if one appears, EXAN outputs an error.

I. EXECUTABLE STATEMENTS

When the first executable statement is encountered, the two data-size items (above) are output. To aid Phase VI and VIII, a continue item goes to File 61 which makes the first executable statement the beginning of a block of code.

Executable statements are of many types, and the types are intermixed unpredictably. Phase IV processes input from File 4 in groups, each group containing all items with identical sequence numbers as assigned by the previous phases. The identification of the least item of the group determines the path of processing.

1. Arithmetic Statements: Left Side

The EXAN subroutine processes subscripted variables on the left and in input-output lists, the entire right side of arithmetic statements and arithmetic statement functions, the arguments of CALL statements and the operands of control statements.

Each variable on the left appears in a group by itself. Preceding and following the list of variables on the left are "list" items. Phase IV attached the arithmetic mode of the expression to these items.

a. If the variable is equivalent, it is treated as a definition point for all equivalent variables. (Treatment of definition points in Phase IV is described in Appendix II, TEST 1 subroutine).

b. If the variable is unsubscripted, Phase IV asks whether it represents an undimensioned variable or an implicit array, if the latter, the number of elements in the array, if attached to the item.

c. If the variable is subscripted, EXAN analyzes the subscript.

d. If the variable is an array scanned by an explicit induction variable, it appears in File 4 as a DO loop group (q.v.) where the group contains only the variable within its range.

2. Arithmetic Statements: Ride Side

The leading item in a group containing an arithmetic expression determines the processing as follows:

a. +, -, *, /, or **. Go to EXAN.

b. Colon. Colons also precede CALL's and subscripted variables on the left side and input-output lists. Treatment of these cases is explained elsewhere. Unexceptional cases are handled by EXAN.

c. Mode of Expression Items. Phase III outputs one such item corresponding to each variable on the left. Phase IV examines them; if any is double precision, the mode of the expression is set to double precision; if not, it is set to single precision. EXAN then analyzes the expression, attaching the mode to operations and functions.

3. CALL Statements

Because the called subprogram may redefine any variable in COMMON storage, the CALL statement is considered a definition point for all COMMON variables (see TEST 1, Appendix II). At each CALL, Phase IV arranges for Phase V to compute and store all current common induction variables.

The arguments of the CALL are processed by EXAN. If statement names appear as the arguments of CALL's, Phase III will have generated successor items and attached them to the CALL group. These are sent unchanged to File 60.

4. Control Statements

DO statements are described separately below. The operands of all other control statements are processed by EXAN.

5. DO Statements

Note: Explicit arrays in input-output lists and on the left of arithmetic expressions are treated as DO loops after Phase I.

Phase I and III generate and Phase IV receives the following items:

a. At the beginning of each loop:

1. A 19 item, containing:

A. The mode word of the induction variable.

B. The sequence number of the loop's end.

2. The initial, final, and incrementing values of the induction variable (constant or variable). These values are forwarded immediately to F51, along with the dictionary reference of the induction variable.

3. TO and FROM items for possible transfers of control to or from the current loop, except transfers to the same loop which do not bypass inner loops.

b. At the end of each loop:

A 20 item marking the end of the loop and containing the mode word of the induction variable.

Throughout Phase IV, a table is kept in memory of current loops. An entry is made in the table at the beginning of each loop; the entry is output at the end of the loop.

Each entry contains:

a. The sequence number at which the loop begins.

b. The sequence number at which it ends.

c. Whether or not the induction variable is COMMON.

d. Whether or not the loop has the same range as the next outer loop.

e. Whether or not there are entrances to the loop, i.e., FROM items from outside the loop's range.

f. Whether or not there are exits from the loop, i.e., TO items to outside the loop's range.

g. Whether or not the loop is optimum as an outer loop, i.e., the only transfers to the loop are from nearby in the same loop.

h. Whether or not it is optimum as an inner loop, i.e., has no exits to outside its range.

i. Whether or not coding must be generated to compute and store the induction variable, i.e.,

1. It is used in arithmetic, or in a control statement.

2. It is COMMON and a CALL exists within its range.

3. SAL coding exists within its range.

4. Its loop contains an exit.

j. The new name of the induction variable. To facilitate Phase V processing, each induction variable is "renamed" according to

its nest number and level number, where these are defined as follows:

1. The nest number (0-999) begins at zero and increases by 1 each time one or more loops end. There are as many "nests" in this sense as there are innermost loops.

2. The level number (0-99) begins at zero, increases each time a loop begins and decreases each time one ends; it represents the number of current loops at any point. 100-level is the L part of the new name.

Items a through i are forwarded to F51 only when the loop is complete. At that time, also, an error is output if there exist one or more outside entrances to the loop but no outside exits. Also, the loop is called non-optimum if it is at Level 1 or if the next outer loop is non-optimum as an outer loop.

Note on Optimum Loops

When a subscript contains induction variables from two successive loops, Phase V will attempt to combine the increment of the outer variable with the initializing of the inner variable if the compiler can guarantee that the execution of one of these terms necessitates the execution of the other; if so, the inner loop is called "optimum".

The inner loop is optimum only if it is optimum as an inner loop and the next loop out is optimum as an outer loop.

The flow of control can be quite complex. Phase IV does not do a flow analysis; but where there is doubt it errs on the side of safety.

5. SAL Code

The presence of SAL code is signalled by one item in F4. When such an item is encountered, the SAL block item output by Phase III is changed to a continue item for Phases VI and VIII. All current induction variables are marked compute and store.

Phase IV - - Appendix I - - EXAN (Expression Analysis Subroutine)

A. Treatment of Expressions by EXAN

Expressions treated by EXAN arise in a number of different types of LSC statements, as described by the writeup. All expressions in File 4, however, consist of sequences of operators and operands in "Polish" notation, each operator requiring two operands.

B. Processing in Phase IV consists of:

1. Attaching definition points to variables as described in Appendix II.

2. Attaching the mode of the expression to operators and functions not within subscripts for use by Phases VI and VII.

3. Recording uses of induction variables (except in simple subscipts) for Phase V (see DO loops).

4. Deleting subscripts within subscripts and subscripts within arithmetic function statements (these are errors).

5. Breaking down subscripts into several parts, testing the parts and sending them to Files 50, 52, and 60, as described below.

C. Treatment of Subscripts by EXAN.

Each subscript consists of from 1 to 9 expressions or terms, terms being set off by commas in the source language. Except that terms may not contain subscripts, each term is itself a full arithmetic expression, as simple as a single constant or as complex as a nest of 400 functions.

By multiplying each term by the product of the dimension of the previous terms, Phase IV converts the source language subscript

T1, T2, . . . into the single arithmetic expression
T1 + D1 *T2 + D1 * D2 * T3 + . . .

(where D1, D2, . . . are the dimensions of the subscripted variable).

If the wrong number of terms is present, Phase IV outputs an error item and processes only those for which dimensions have been given.

As it creates a single expression from the subscript, Phase IV breaks the expression into three parts.

1. A constant part, This, and the name and sequence number of the subscripted variable, go to File 50. The two are renamed to become the M-address of the output SAL code.

2. An induction variable part. Each induction variable, its numeric coefficient, and the sequence number go to F52. The induction variables are included in the B-register of the output code.

3. A remainder. The remainder contains ordinary variables, functions, and the part of the subscript, if any, which is too complicated for Phase V processing. The remainder goes to F60, and is included in the B-register of the output code.

EXAMPLES: A has dimension (2, 3, 4)

a.

| | | |
|---|---|---|
| File 1 | A (1,2,3) | 1 + 2 (2) + 3 (6) = 23 |
| File 2 | : A , , 1 2 3 | |
| File 50 | + 23 A | |
| Files 60-61 | : A ' 0 | |
| Files 7 | A' | |

b.

| | | |
|---|---|---|
| File 1 | A (I, J, K) | I,J,K induction variables. |
| File 4 | : A , , I J K | |
| File 52 | 1I, 2J, 6K; n = 0 | |
| Files 60-61 | : AB | |

B is a B-register name.

c.

| | | |
|---|---|---|
| File 1 | A (X, Y, Z) | no induction variables. |
| File 4 | : A , , X Y Z | |
| Files 60-61 | : A : CX + X + *2 Y * 6Z | |

CX = "convert to fixed" function.

d.

| | | |
|---|---|---|
| File 1 | A (1, J, Z) | J and Z as above |
| File 4 | : A , , 1 J Z | |
| File 50 | + 1 A | |
| File 52 | 2 J ; n = 1 | |
| Files 60-61 | : A fx+ * 6 Z B | |

fx+ = fixed point add operation

D. Treatment of a Single-Item Subscript-Term by EXAN

1. Numeric. The term is multiplied by p (Product of dimensions to that point) and added to total constant for the subscript. An error is written if it is larger than the corresponding dimension, or less than 1.

2. Induction variable. The product p is added to the coefficient for the appropriate induction variable. (Coefficients for all current induction variables are set to zero before processing of the subscript begins).

3. Ordinary variable, or function without arguments. The SAVE subroutine saves the item and its coefficient.

E. Treatment of Multiple-Item Subscript Terms by EXAN

The processing is in three passes, as follows:

1. Scan forward, creating an internal file, one item for each item of the term.

   A. Operation items include the type of operation and the addresses of the operands.

   B. Numeric items contain their value.

   C. Variable items contain coefficients of 1 and whether or not induction variable.

   D. Terms containing division, exponentiation, a function with arguments, or more than one non-induction variable are not broken down. Instead, the term goes to F60, preceded by + , * , and p items.

2. Scan backwards, examining operations only.

   A. Addition. Perform addition of numerics.

   B. Subtraction. Perform subtraction of numerics. If either operand is non-numeric, multiply the terms of the second operand by -1.

   C. Multiplication. Perform multiplication of numerics. If only one is numeric, multiply the terms of the other by it. If both are non-numeric, send the complete term to F 60 preceded by +, *, and p items.

3. Scan forward, examining operands only.

   A. Numeric. Multiply by p and add to term's constant. If zero, ignore.

   B. Induction variable. Multiply coefficient by p and add to induction variable coefficient table. If the coefficient is negative, note special situation.

   C. Other variable or function without arguments. The SAVE subroutine saves the item.

Source Statements: A = B(. . . , (4*3)-(2+I),. . . )

Treatment of the term: (4*3) - (2+1)

F4 input: - * 4 3 + 2 I

1st Pass Table:

| | | | | |
|---|---|---|---|---|
| Entry: | 0: | - | 1, 4 | (operand addresses) |
| | 1: | * | 2, 3 | " " |
| | 2: | const. | 4, 0 | (coefficient) |
| | 3: | const. | 3, 0 | " |
| | 4: | + | 5, 6 | (operand addresses) |
| | 5: | c | 2, 0 | (coefficient) |
| | 6: | I | 1 | " |

2nd Pass:

Scanning backward the first operation is + in (4). Its operands (5) and (6) are both single but not both numeric. No change. Next operation is in (1). Both operands are numeric; so the constant designator c goes to (1); 4*3 goes to coefficient (1); "ignore" mark goes to (2), (3). - in (0) has one non-numeric operand. Its terms (2 and I) go to a new table, the coefficients of which are multiplied by -1; so that -2 goes to coefficient (5); -1 to coefficient (6).

3rd Pass:

The table looks like this (ignoring operations):

| | |
|---|---|
| W0 | ignore |
| W1 | constant 12 |
| W2 | ignore |
| W3 | ignore |
| W4 | ignore |

| | | |
|---|---|---|
| W5 | constant | -2 |
| W6 | I | 01 |

The constants 12 and -2 are added to the total constant for the subscript.

If I is an induction variable, -1 is added to its coefficient so far. If not, the SAVE subroutine files it.

F. Post-Processing of Subscript Terms

After a decomposable term is processed, a note is made if it is potentially negative. This is necessary because the LARC represents numbers in signed magnitude form but does not consider the sign of the index register in indexing.

2. Induction Variable Part.

An item goes to F52 for each current induction variable with non-zero coefficient.

3. Non-Induction Variable Part.

The table set up by the SAVE routine is scanned and its entries output. If it is not empty, items causing fixing of its components are output.

4. Potentially Negative Subscripts.

If part of the subscript goes to F52, the note of negativity goes with it. If not, "+100000" goes to F60. Thus, the M-field of the generated B-register is biased so that it cannot go negative.

The command

F #0 A+20 #1

works identically for the two cases

(#1) = 1 and (#1) = -1.

LSC adds the constant portion of a subscript to the M-address of the object code: the rest of the subscript is contained in the referenced B-register. LSC must, therefore, insure that no B-register is negative when it is referenced.

LSC assumed that each term of the subscript is positive and, therefore, only those terms which Phase IV decomposes can possibly create negative B-registers. In fact, a negative B-register can result only if the term contains a constant greater than zero. It must also contain a negative induction variable or another factor or factors non-analyzable by Phase IV.

Phase IV also checks whether a decomposed term contains only induction variables and a positive constant greater than the corresponding dimension. This is an error.

G. Post-processing of Subscripts

1. Constant Part

If the total constant is zero, the item for the subscripted variable goes to F 61. If not, the variable and its constant go to F50.

If the subscript is all constant, a zero goes to F60 as the subscript. (Phase VI eliminates zero subscripts.)

Phase IV--Appendix II--Subroutines

A. TEST1

TEST1 processes an item in an arithmetic expression and returns to the calling sequence at one of 5 exits as follows:

1. Constant
2. Induction Variable
3. Unsubscripted Variable

The parameter number of the variable, if any, is moved to the mode word. The d-digit of the mode word is set to zero.

4. Subscripted Variable

   After the above, d is set to 1.

5. Operation

   Operation and functions are marked with the mode of computation of the expression. Functions are then treated as variables.

To eliminate unnecessary recomputation of arithmetic quantities by the object program, Phases III and IV attach to each non-induction variable a definition point, representing the last possible change in the variable's value.

TEST 1 modified the Phase III assigned definition points of common and equivalent variables only. The phase keeps a running record of the definition points for those special classes of variables.

The EQUIVALENCE point is initialized when the first executable statement is processed, and replaced by the current sequence number whenever an equivalent variable is encountered:

1. On the left of an arithmetic expression.
2. In an input list.
3. As an argument of a CALL.
4. As the induction variable of a loop.

The COMMON point is updated at each CALL statement.

B. TEST2

TEST2 tests all limits of loops. Its primary function is to update the definition points of common and equivalent variable limits.

C. SETZ

This subroutine, which is called by TEST2 (see below) and by some of the calling sequences to TEST1, simply records the necessity to compute and store an induction variable by setting a digit to 1 in the table for that variable.

D. SAVE

This routine sets up a table of non-induction variables in subscripts and their coefficients. (A maximum of 9 entries).

It insures that the source statements

```
DIMENSION      A(4, 5, 6)
    B = (A (X, X, X)
```

produce this code

```
F    #0    (.25 (02) )
M    #0    X
CX   #0    61
F    #0    A #0
S    #0    B
```

not this code

```
F    #0    (.20 (02) )
M    #0    X
F    #1    (.4 (01) )
M    #1    X
A    #0    #1
A    #0    X
CX   #0    61
     etc.
```

E. ITEM

Given a coefficient, R, and item, K, and a sequence number, S, this routine outputs to File 60 on S the items

```
* R  K
```

If this is not the first call to ITEM within a given subscript, a + item is put out on the sequence left from the last call.

Phase IV--Appendix III--Output to F50, F52, F52

F50 4-word item

W1 c c c c c c 0 f f f f f

W2.4 F40 item for variable

c : fixed point constant
leading digit = 0 or 1 for + or -

: dictionary reference of variable.

F51 4-word item, 4 items per loop

Item 1 W1 I I I I I 0 0 D D D D D

W2 t u v w x y z E E E E E

W3-4 ignored

Items 2-4 W2 I I I I I j0 F F F F

W2-4 F4 item for DO limit

I : new induction variable name

D : sequence of start of loop

E : sequence of end of loop

t to z : indicators

j : subsequence 1,2,3, of limits

F : dictionary reference of I

F52 2-word item

W1 I I I I I M 0 M a a a a a a

W2 S SS 0n

I : new induction variable name

a : fixed point constant, leading digit 0 or 1 for + on -

n : 1 if potentially negative.

Phase IV--Appendix VI--Output to F94

F94 4-word item

W1 (Sequence)

W2 (I.D.)

W3 (In general, left of =)

W4 (In general, right of =)

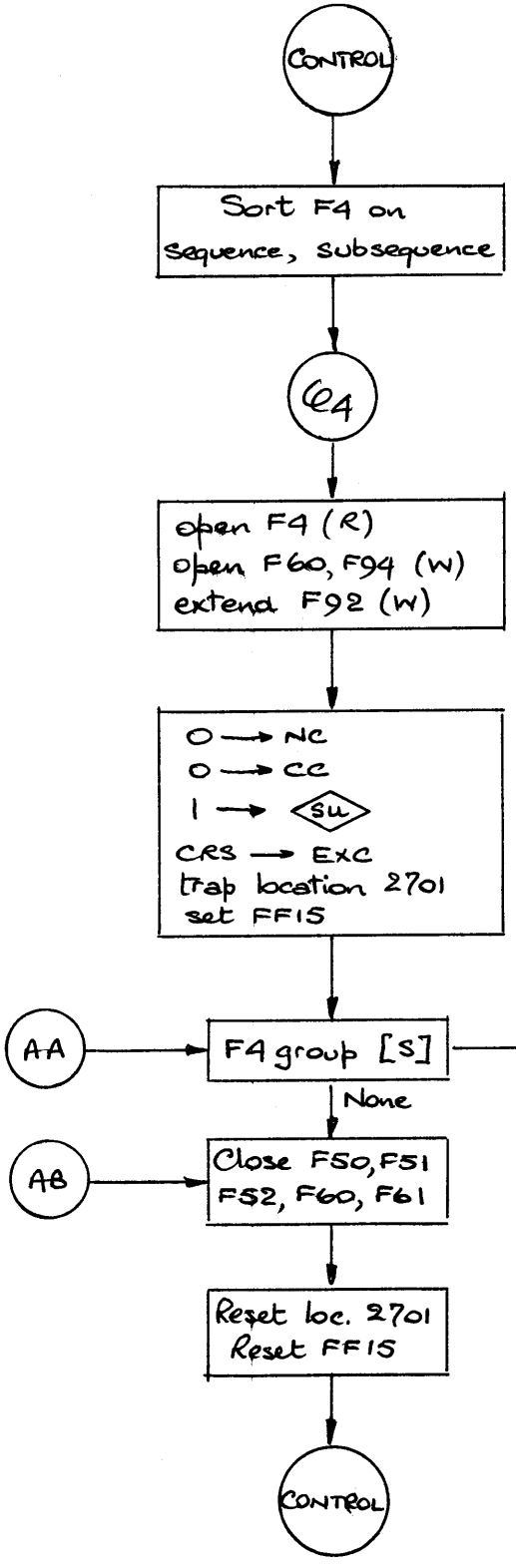
Note: For ID = 11, second word contains abcdeuffffID

| ID | Implied by I.D. | Word 3 | Implied by I.D. | Word 4 | Generating Source Statement |
|---|---|---|---|---|---|
| 2 | DEF | F | : G - | K | DIMENSION |
| 3 | DEF | G | : 9 DAT+ | K | DIM, EQU I |
| 4 | DEF | G | : 9 COM+ | K | EQUIVALENCE |
| 5 | DEF | G | : 9 COM+ | K | COMMON |
| 6 | DEF | F | : 9 DAT + | K | EQUI |
| 7 | DEF | F | : 9 COM+ | K | COMM, EQUI |
| 8 | $STOR | | : 9 DAT | K | - |
| 9 | DEF | F | : | K | PARAMETER |
| 10 | $STOR | | : 9 COM | K | - |
| 11 | Reserve k-storage for F | 1st half | | 2nd half | CONSTANT |
| 12 | LIB | F | | ignored | LIBRARY |

F = contents of dictionary reference f.

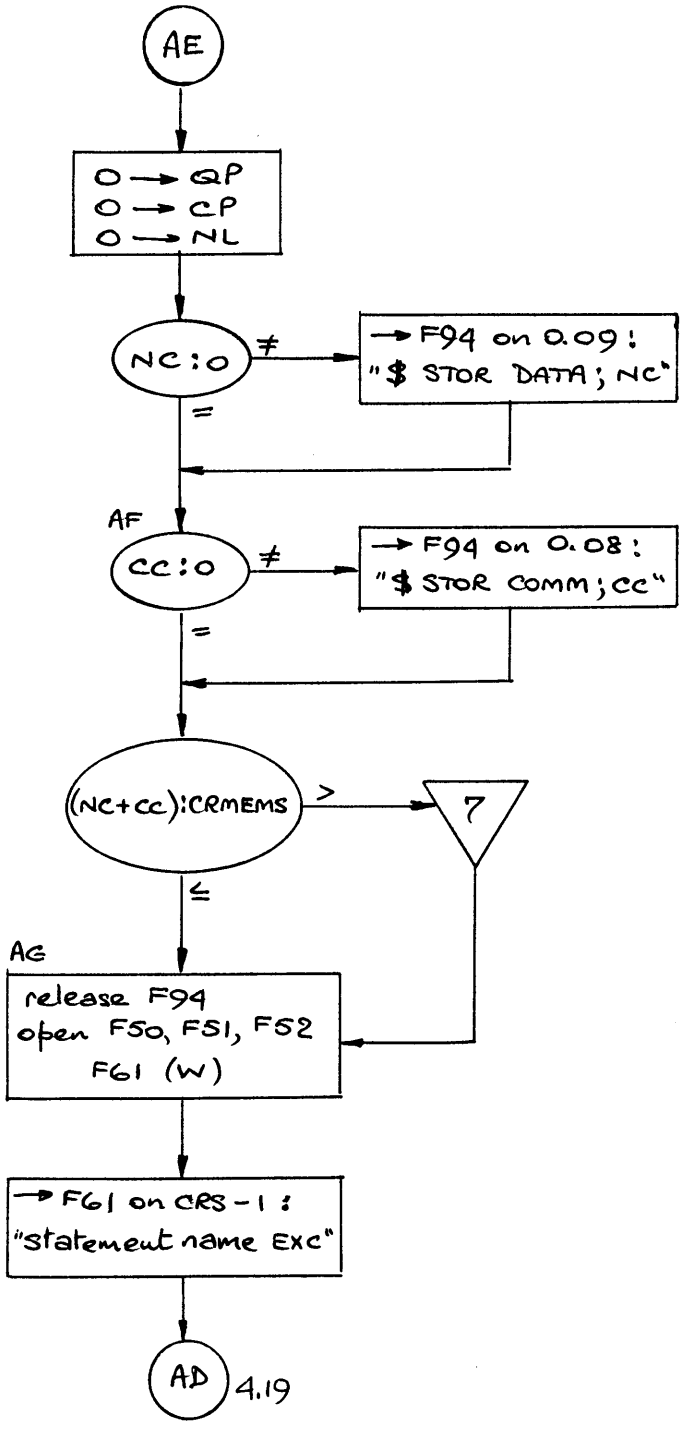
G = symbol generated by Phase IX from f.

K = fixed-point constant




(R) = file is to be read
(W) = file is to be written
CRS = sequence of first executable instruction.
EXC = statement name of first executable instruction.
NC = count of storage for not-common variables
CC = count of storage for common variables

location 2701 is trapped and FF15 is set so that Ø4 can do its own analysis of contingencies.




QP = definition point of EQUIVALENT variables.

CP = definition point of COMMON variables.

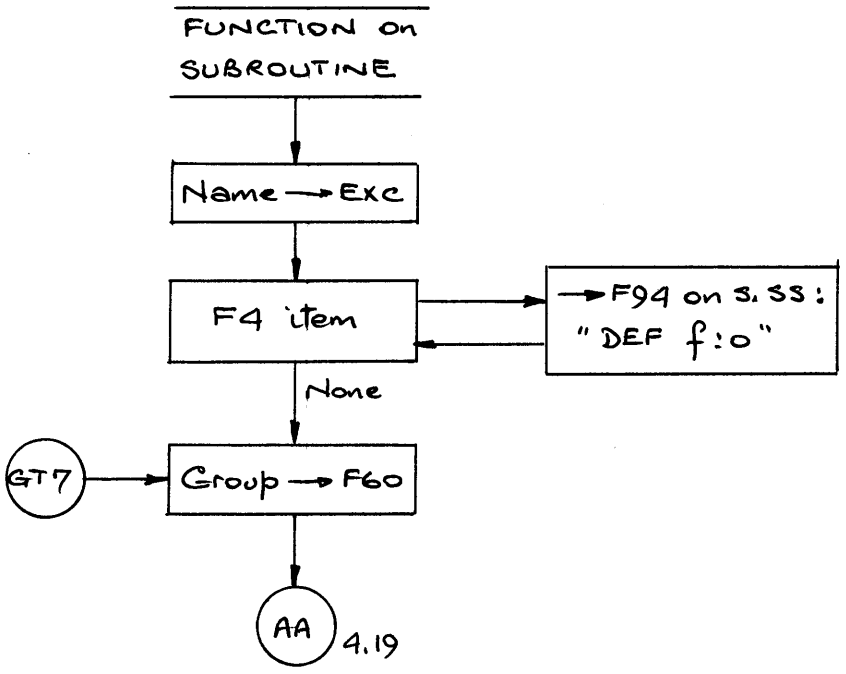
NL = nest and level numbers of current induction variable.

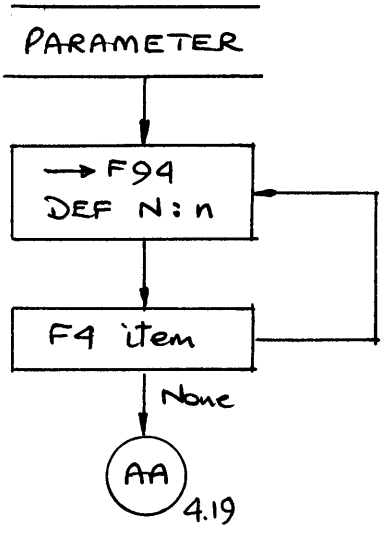
N = 000 - 999

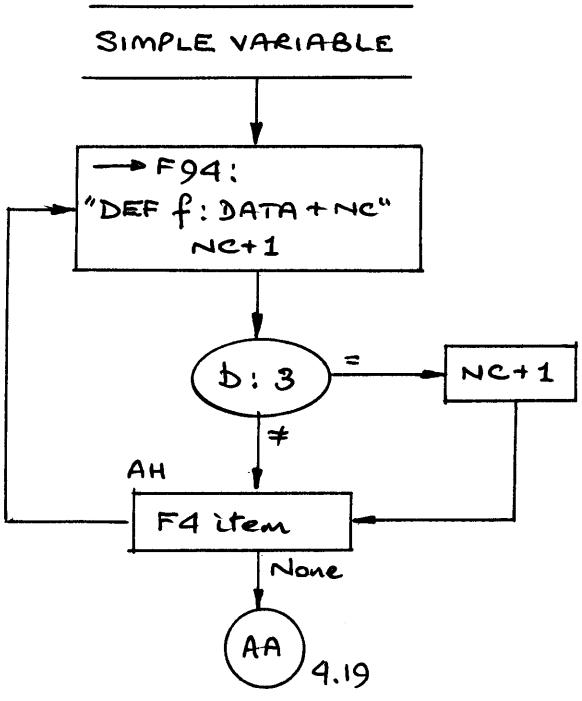
L = 00 - 99

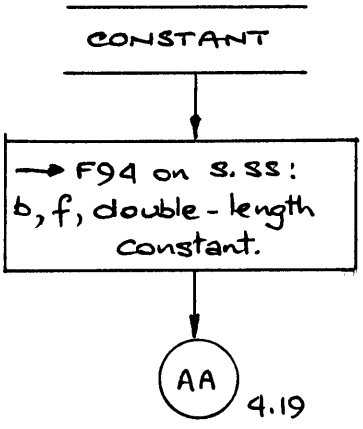
CRMEMS = memory available for data storage.

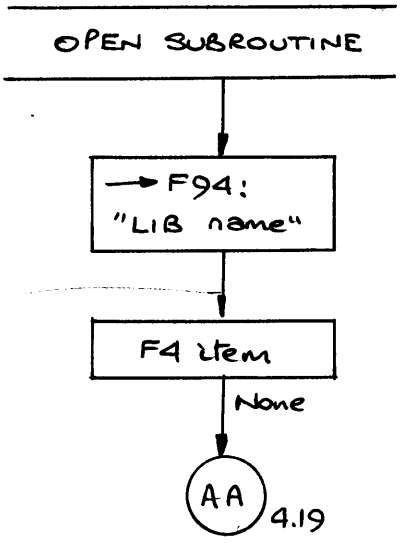
7 Too much storage required.



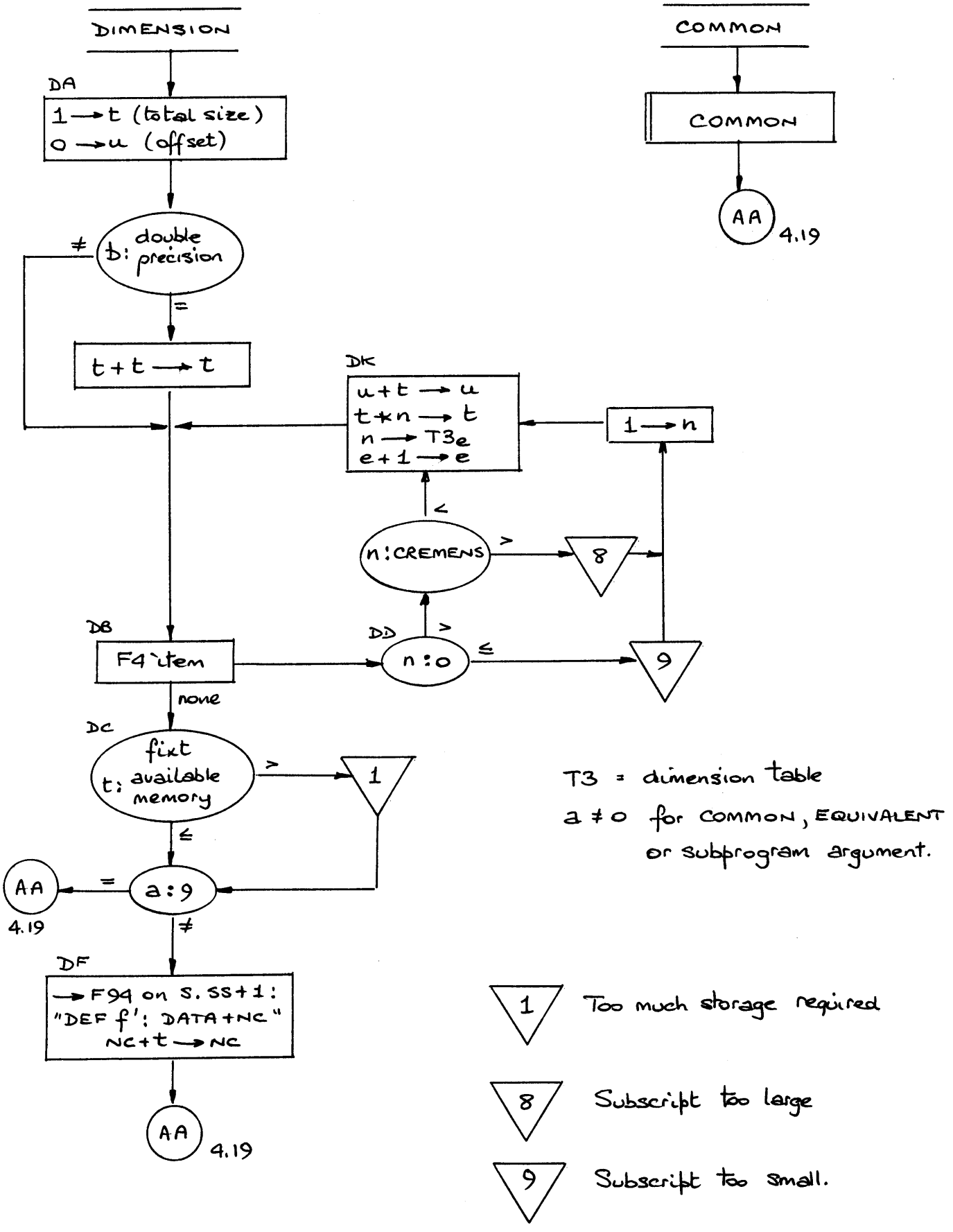



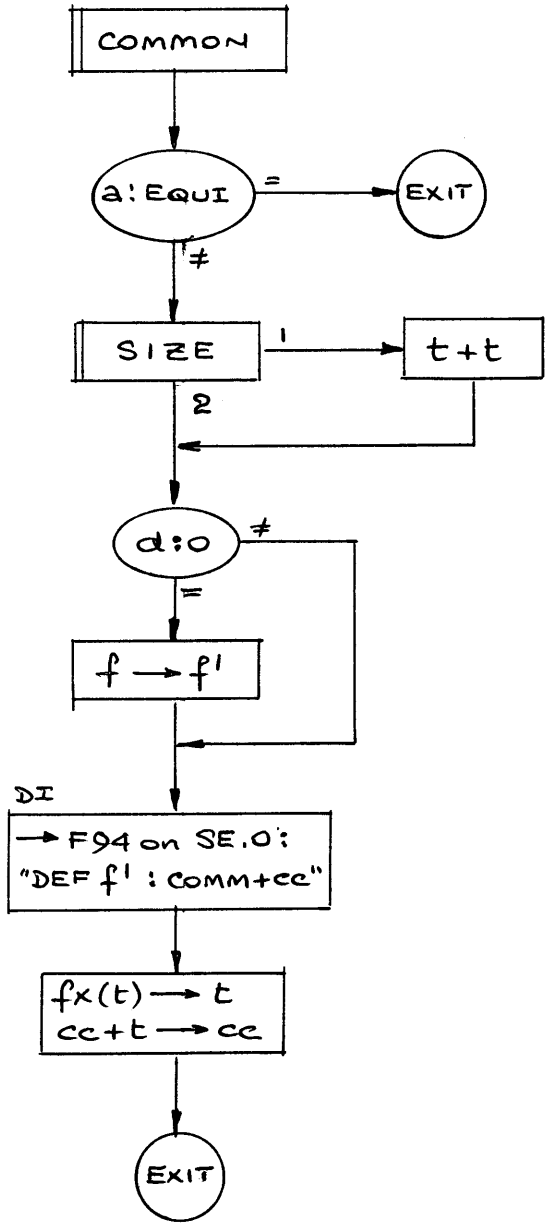



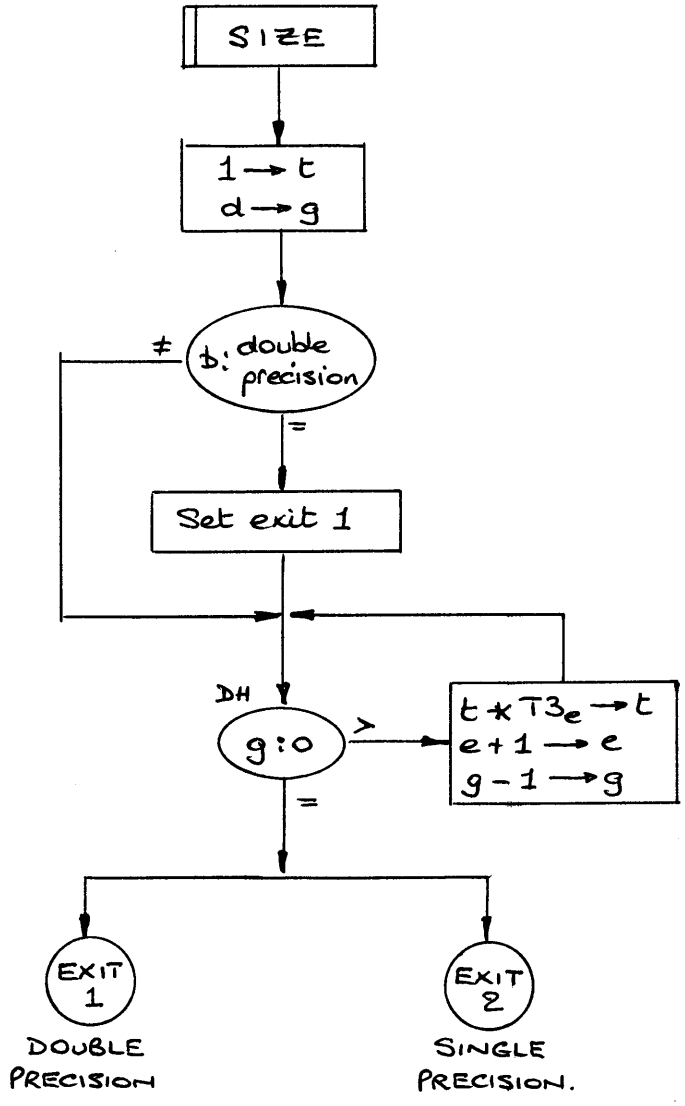




b and f are mode word fields. n is the numeric value of the parameter.

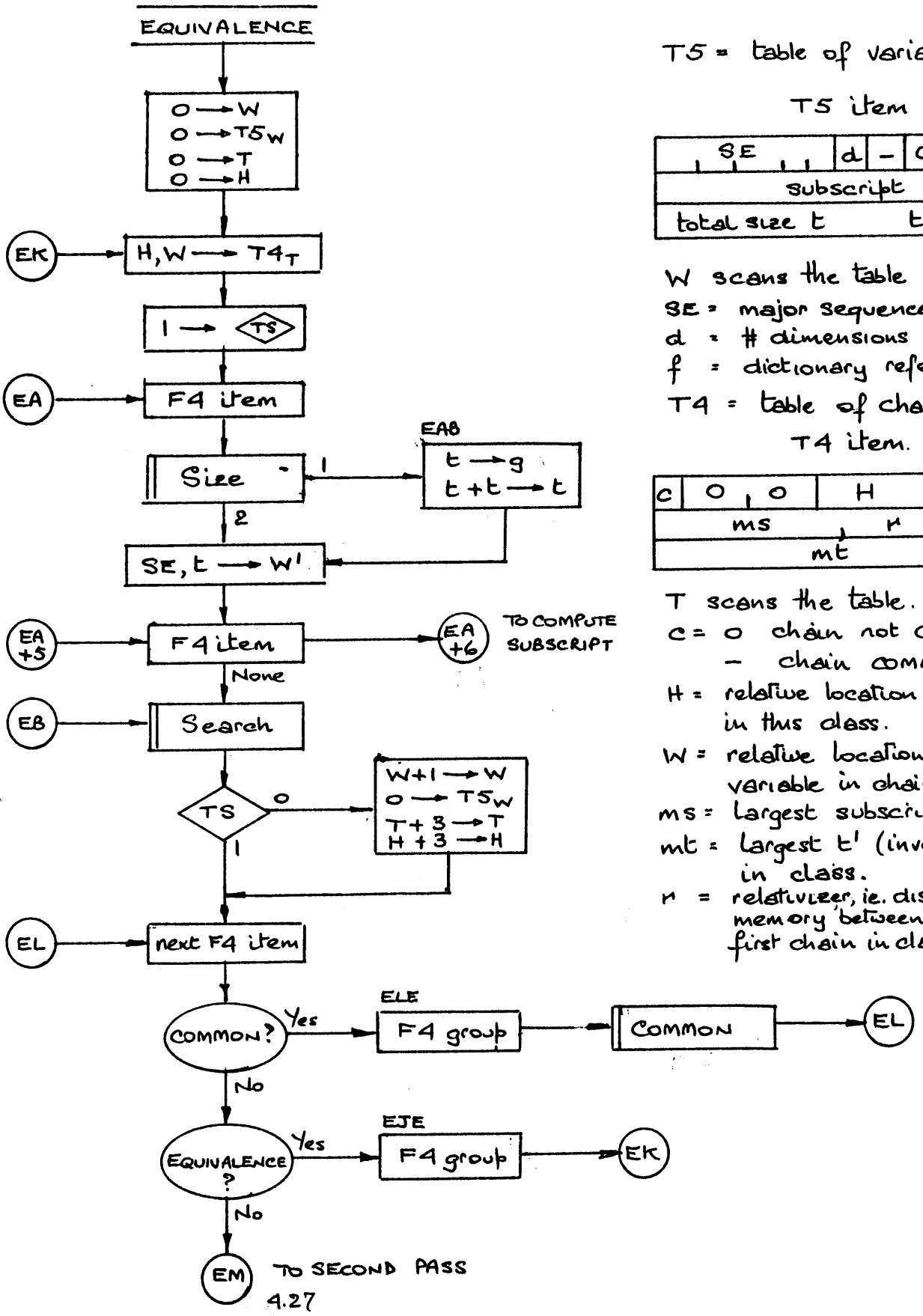

DIMENSION
DA
1 → t (total size)
0 → u (offset)
≠
b: double precision
=
t + t → t
COMMON
COMMON
AA
4.19
DK
u + t → u
t * n → t
n → T3e
e + 1 → e
1 → n
<
n : CREMENS
>
8
DB
F4 item
DD
>
n : 0
≤
9
none
DC
fixt
t: available memory
>
1
≤
AA
4.19
=
a : 9
≠
DF
→ F94 on S.SS+1:
"DEF f': DATA+NC"
NC+t → NC
AA
4.19
T3 = dimension table
a ≠ 0 for COMMON, EQUIVALENT or subprogram argument.
1
Too much storage required
8
Subscript too large
9
Subscript too small.




Subroutine to assign storage to common variable.




Subroutine to multiply the dimensions of a variable.




T5 = table of variables.

T5 item

| SE | d | - | 0 | f |
|---|---|---|---|---|
| subscript s | | | | |
| total size t | | | t-s = t' | |

W scans the table
SE = major sequence
d = # dimensions
f = dictionary reference

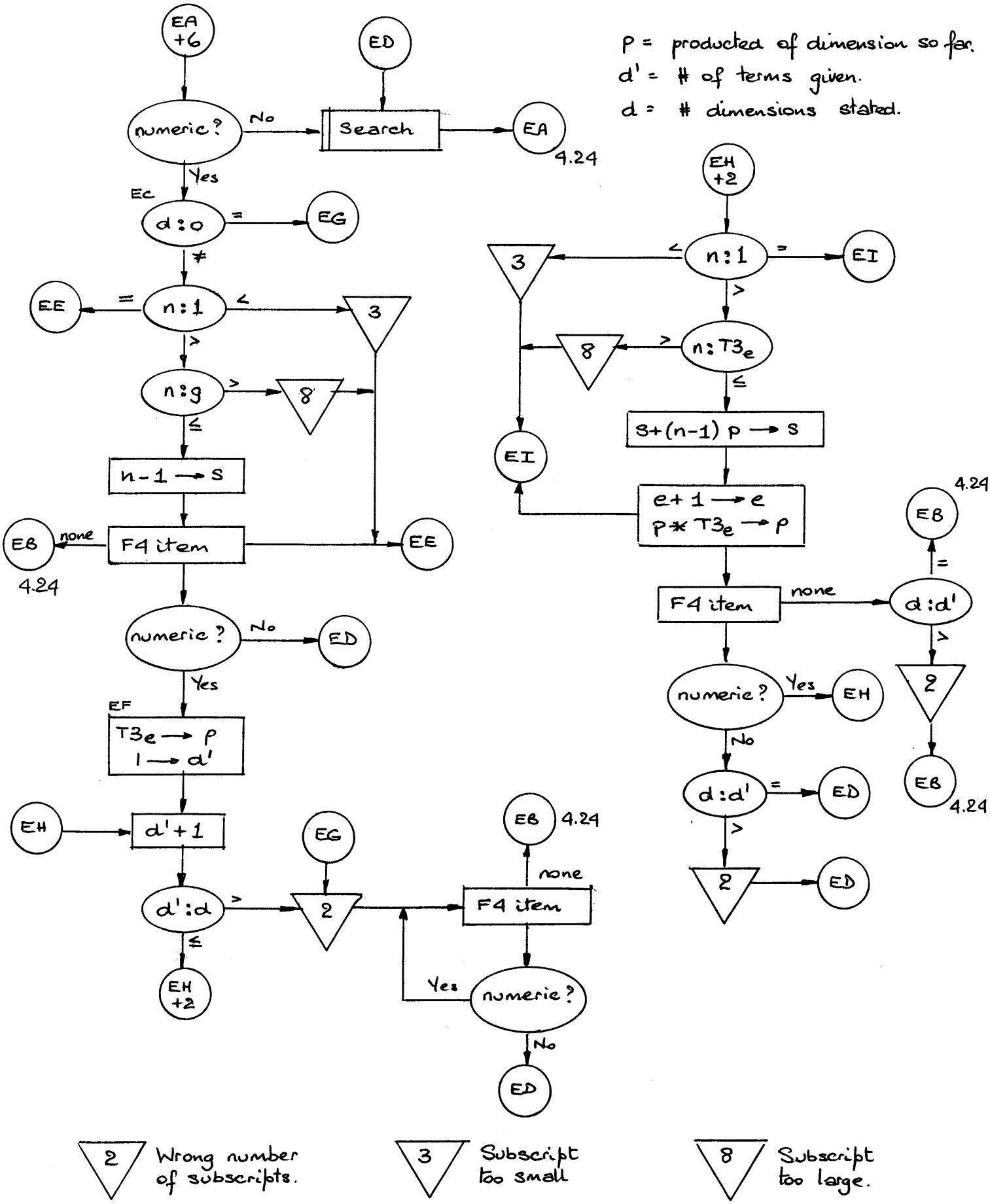
T4 = table of chains.

T4 item.

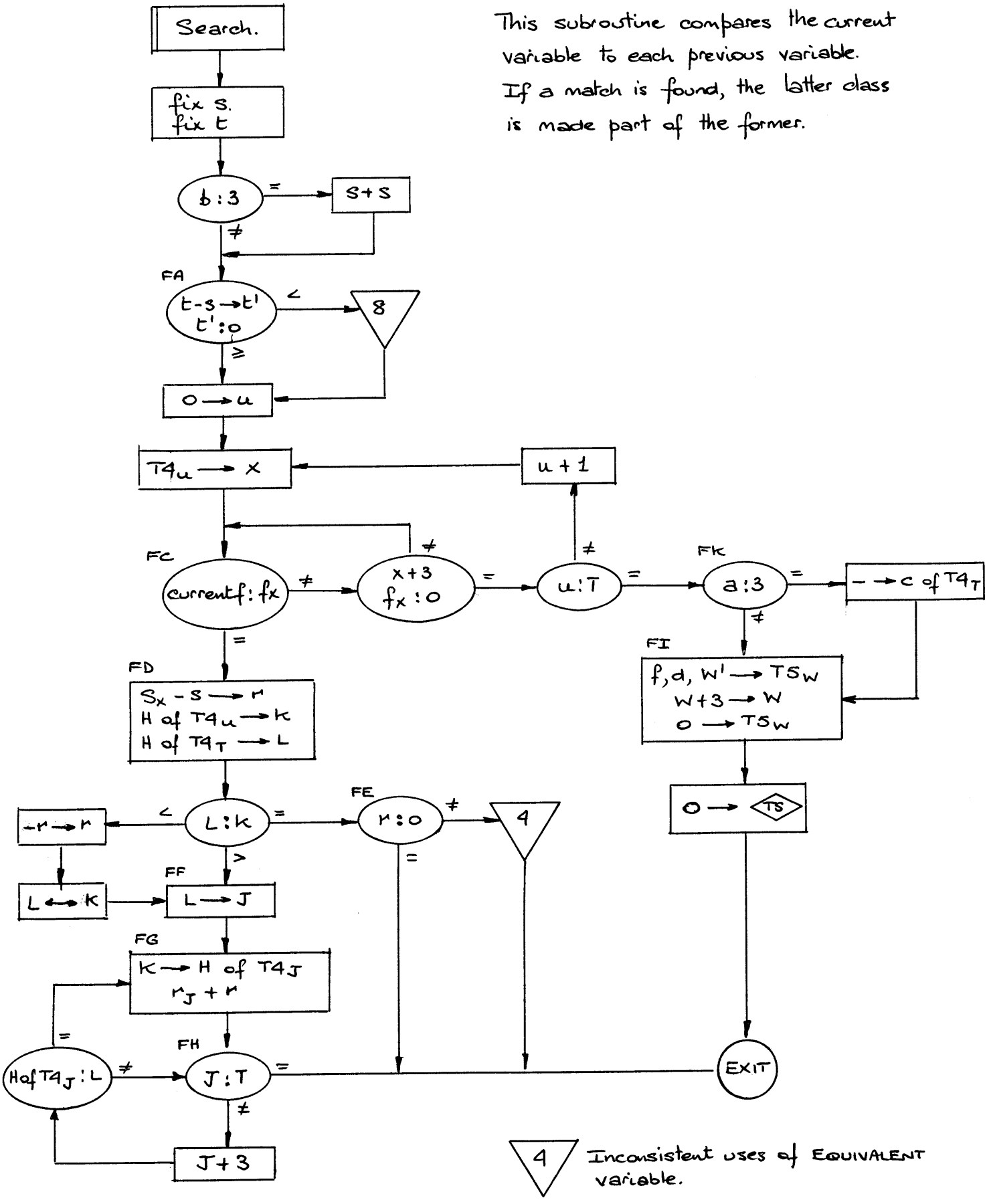
| c | 0 | 0 | H | W |
|---|---|---|---|---|
| ms | | | r | |
| mt | | | | |

T scans the table.
c = 0 chain not COMMON
\- chain COMMON
H = relative location of first chain in this class.
W = relative location of first variable in chain.
ms = largest subscript in class
mt = largest t' (inverse subscript) in class.
r = relativizer, ie. distance in memory between chain and first chain in class.

# EQUIVALENCE (continued): COMPUTE SUBSCRIPT

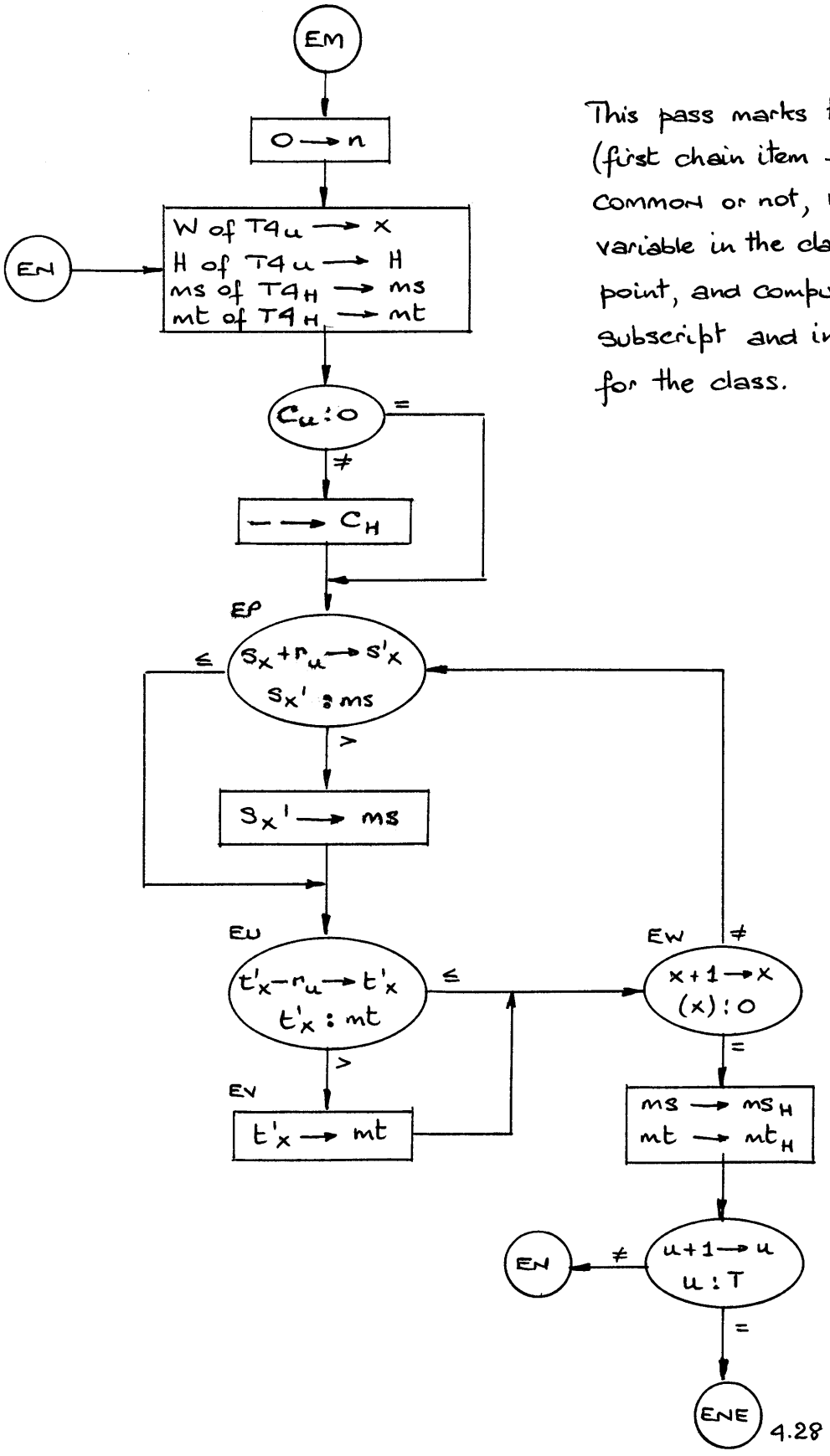


EQUIVALENCE (continued): search chain subroutine.




This subroutine compares the current variable to each previous variable. If a match is found, the latter class is made part of the former.

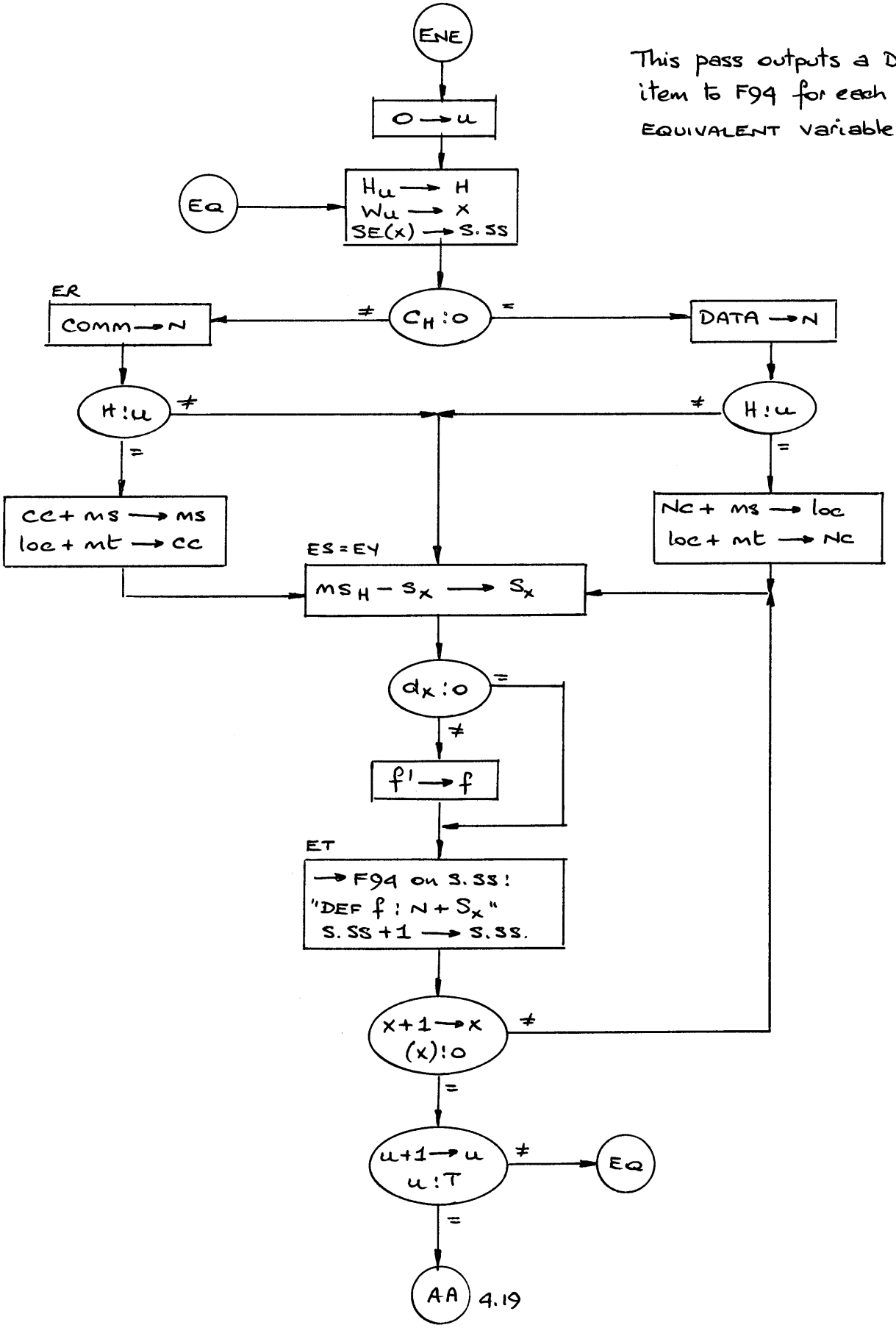
EQUIVALENCE (continued): Second pass.



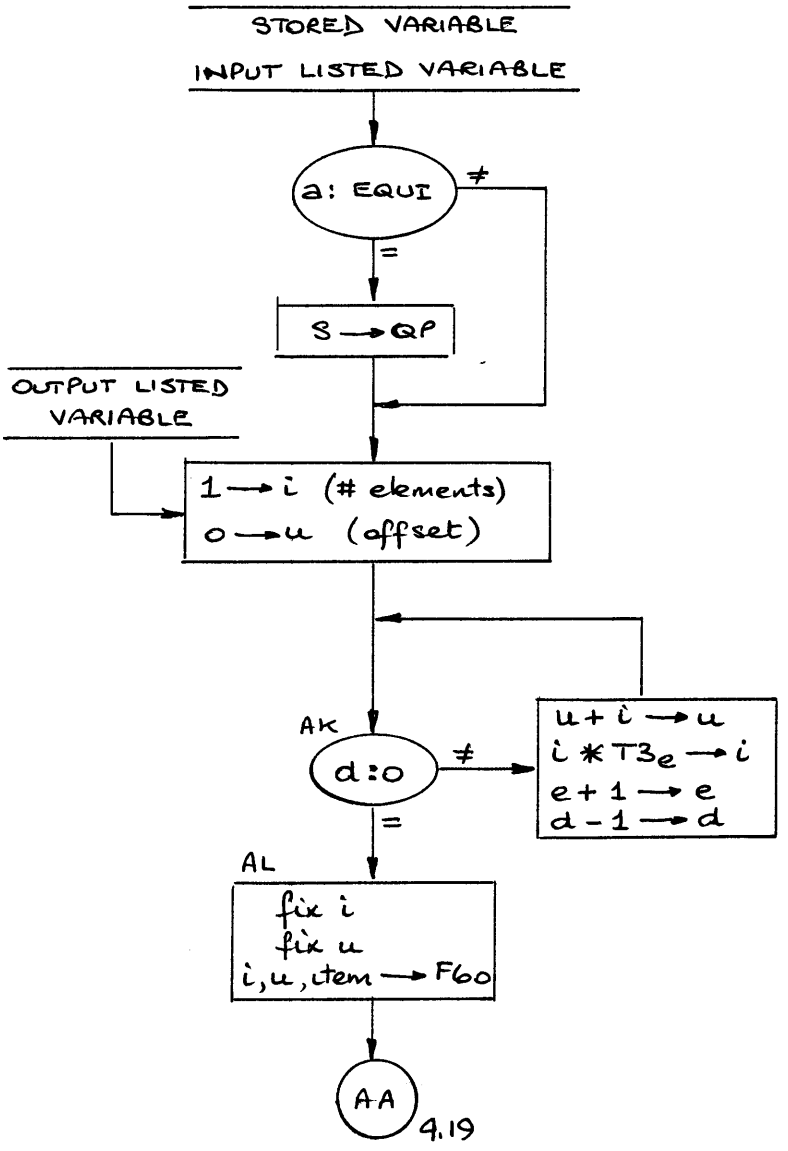

This pass marks the class items (first chain item for each class) common or not, relativizes every variable in the class to a single point, and computes the maximum subscript and inverse subscript for the class.

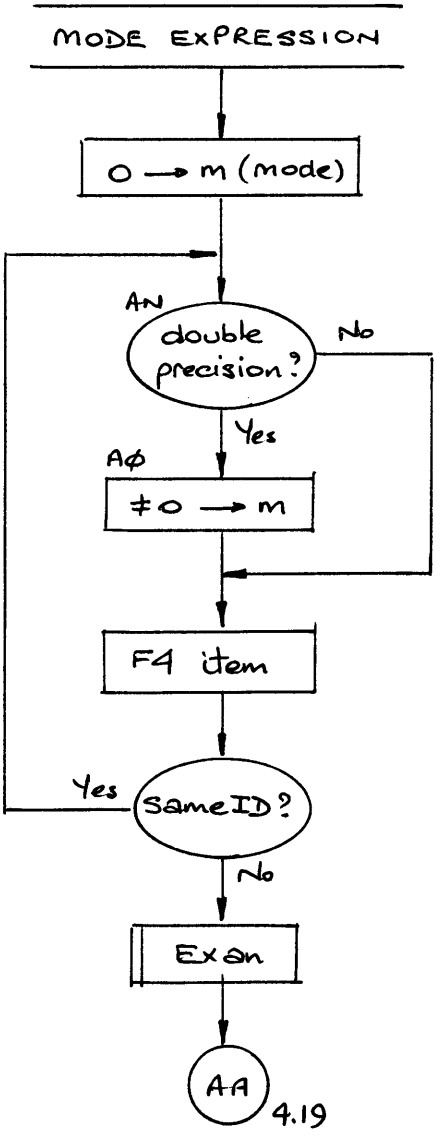
4.28

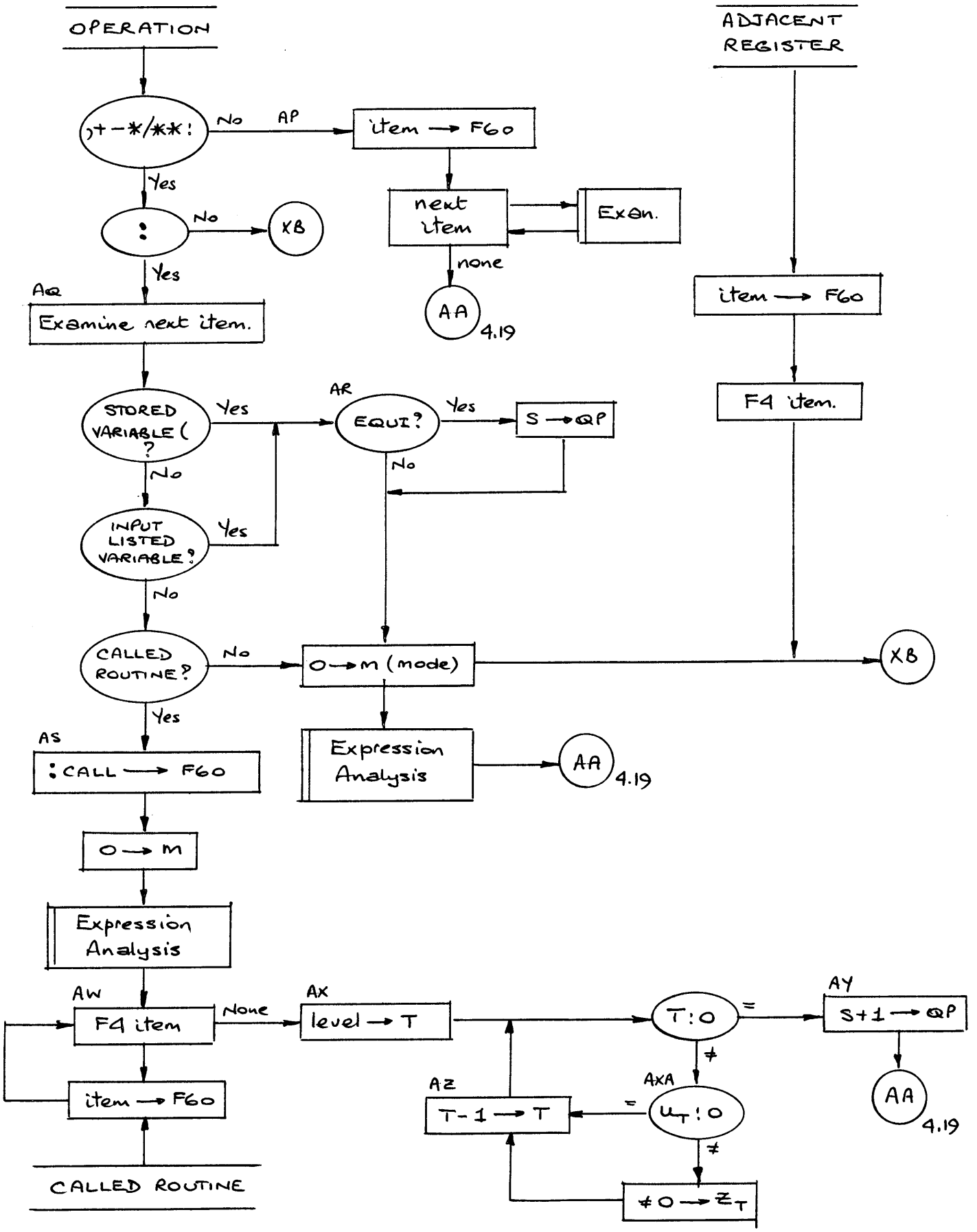
EQUIVALENCE (continued): Third pass.

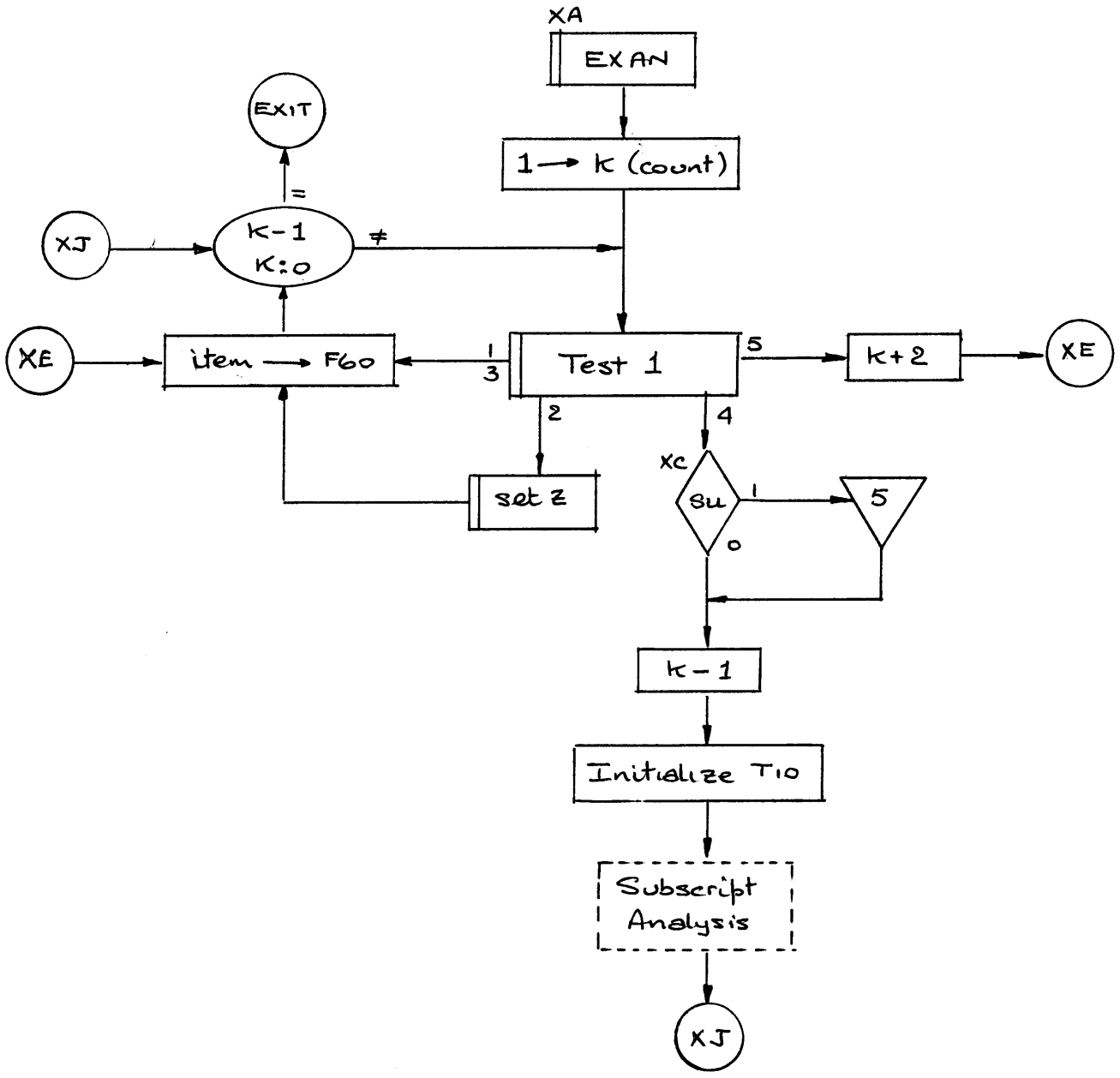



This pass outputs a DEF item to F94 for each EQUIVALENT variable.


STORED VARIABLE
INPUT LISTED VARIABLE
a: EQUI
≠
=
S → QP
OUTPUT LISTED VARIABLE
1 → i (# elements)
0 → u (offset)
AK
d:0
≠
u + i → u
i * T3e → i
e + 1 → e
d - 1 → d
=
AL
fix i
fix u
i,u,item → F60
AA
4.19

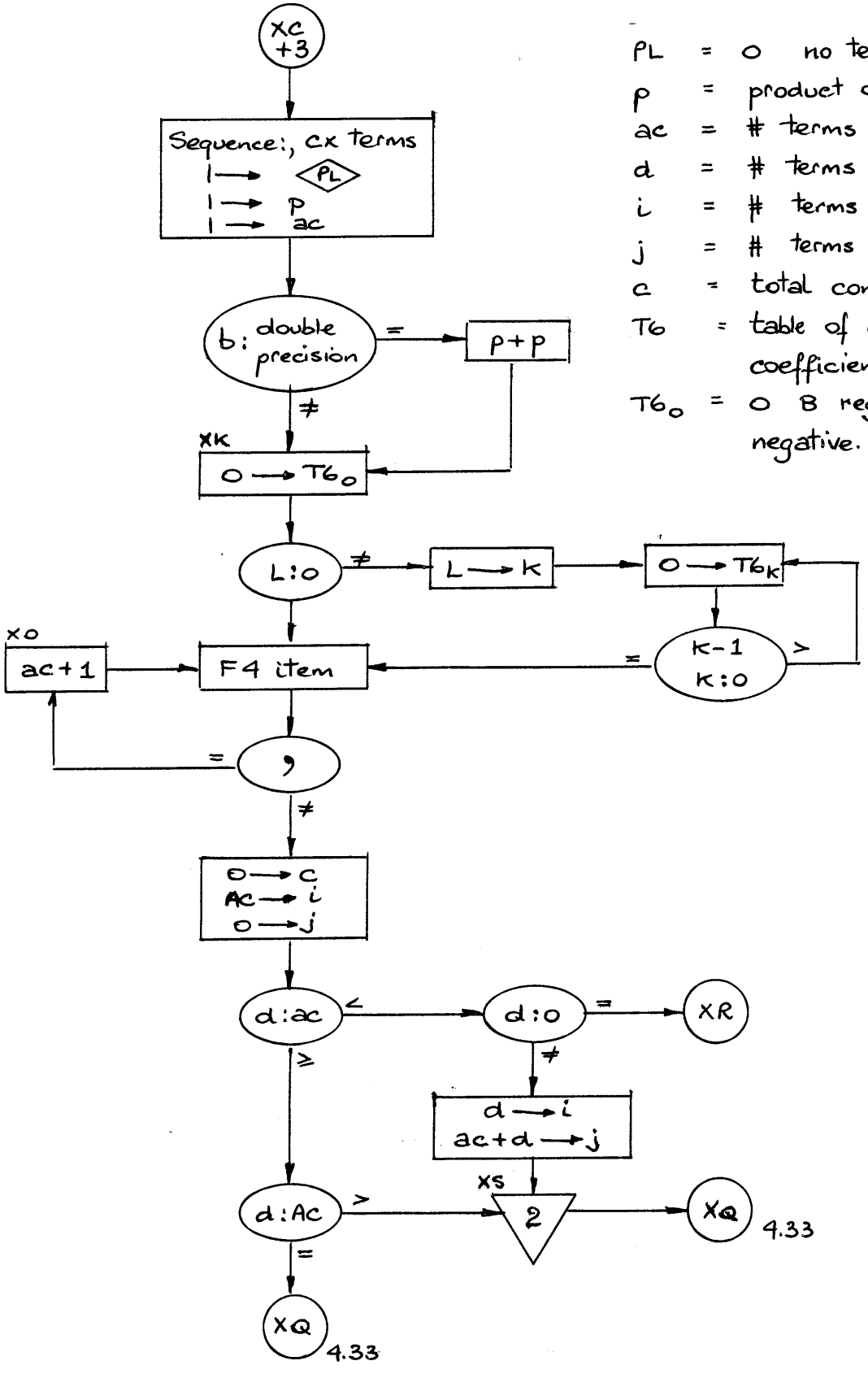

MODE EXPRESSION
0 → m (mode)
AN
double precision?
No
Yes
AΦ
≠0 → m
F4 item
Yes
SameID?
No
Exam
AA
4.19


OPERATION
ADJACENT REGISTER
)+-*/**:
No
AP
item → F60
Yes
:
No
XB
Yes
next item
Exam.
none
AA
4.19
AQ
Examine next item.
item → F60
STORED VARIABLE ( ?
Yes
AR
EQUI?
Yes
S → QP
F4 item.
No
No
INPUT LISTED VARIABLE?
Yes
No
CALLED ROUTINE?
No
0 → m (mode)
XB
Yes
AS
: CALL → F60
Expression Analysis
AA
4.19
0 → m
Expression Analysis
AW
F4 item
None
AX
level → T
T:0
=
AY
S+1 → QP
AA
4.19
≠
AZ
AXA
item → F60
T-1 → T
=
uT:0
≠
≠0 → zT
CALLED ROUTINE




5 Subscript within arithmetic function statement.

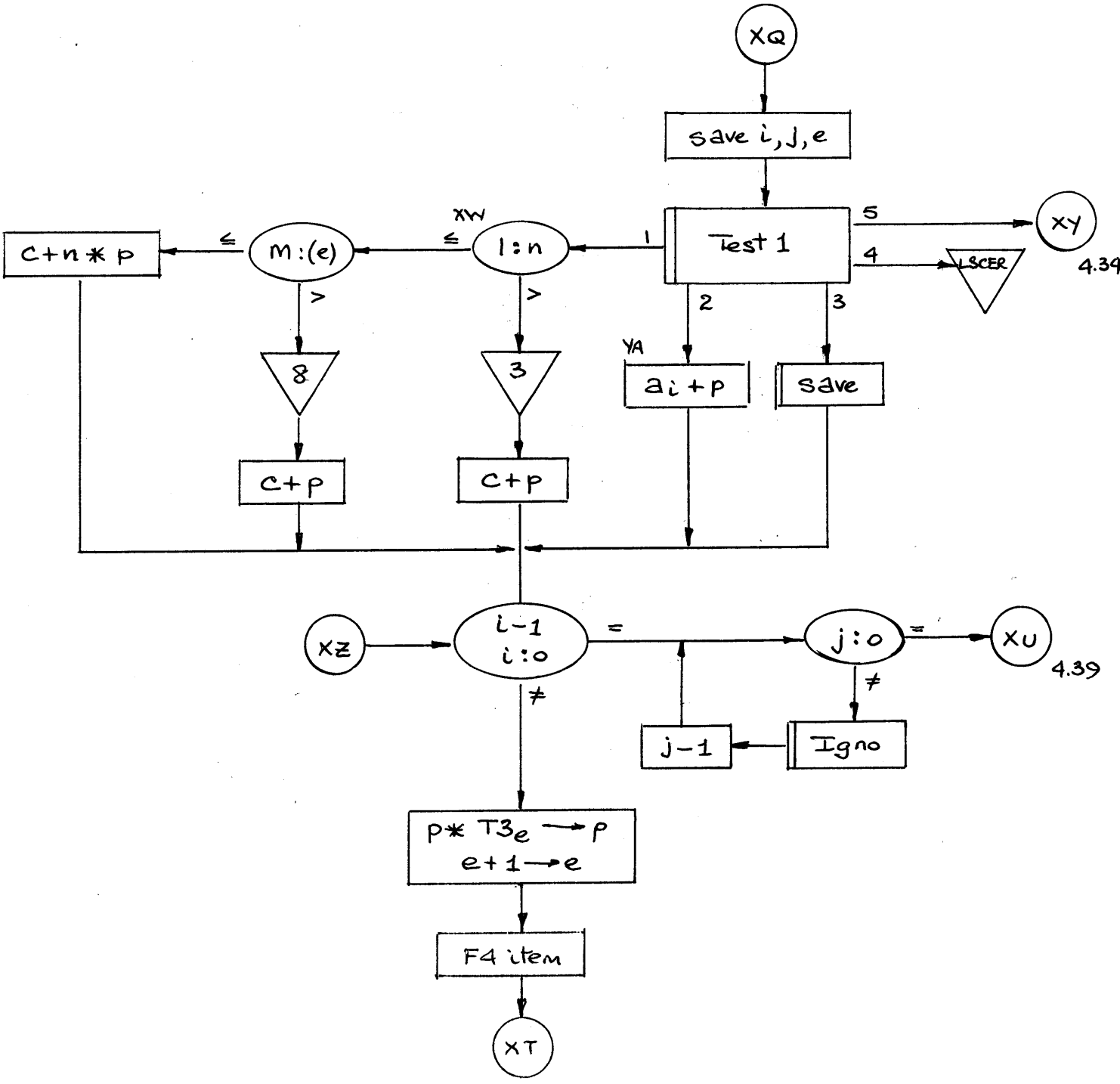
EXAN (continued): Initialize Term Processing.




PL = 0 no terms to F60
p = product of dimensions.
ac = # terms written
d = # terms required
i = # terms to process
j = # terms to ignore
c = total constant for subscript
T6 = table of induction variable coefficients.
$T6_0$ = 0 B register cannot be negative.

2 Wrong number of subscript terms.

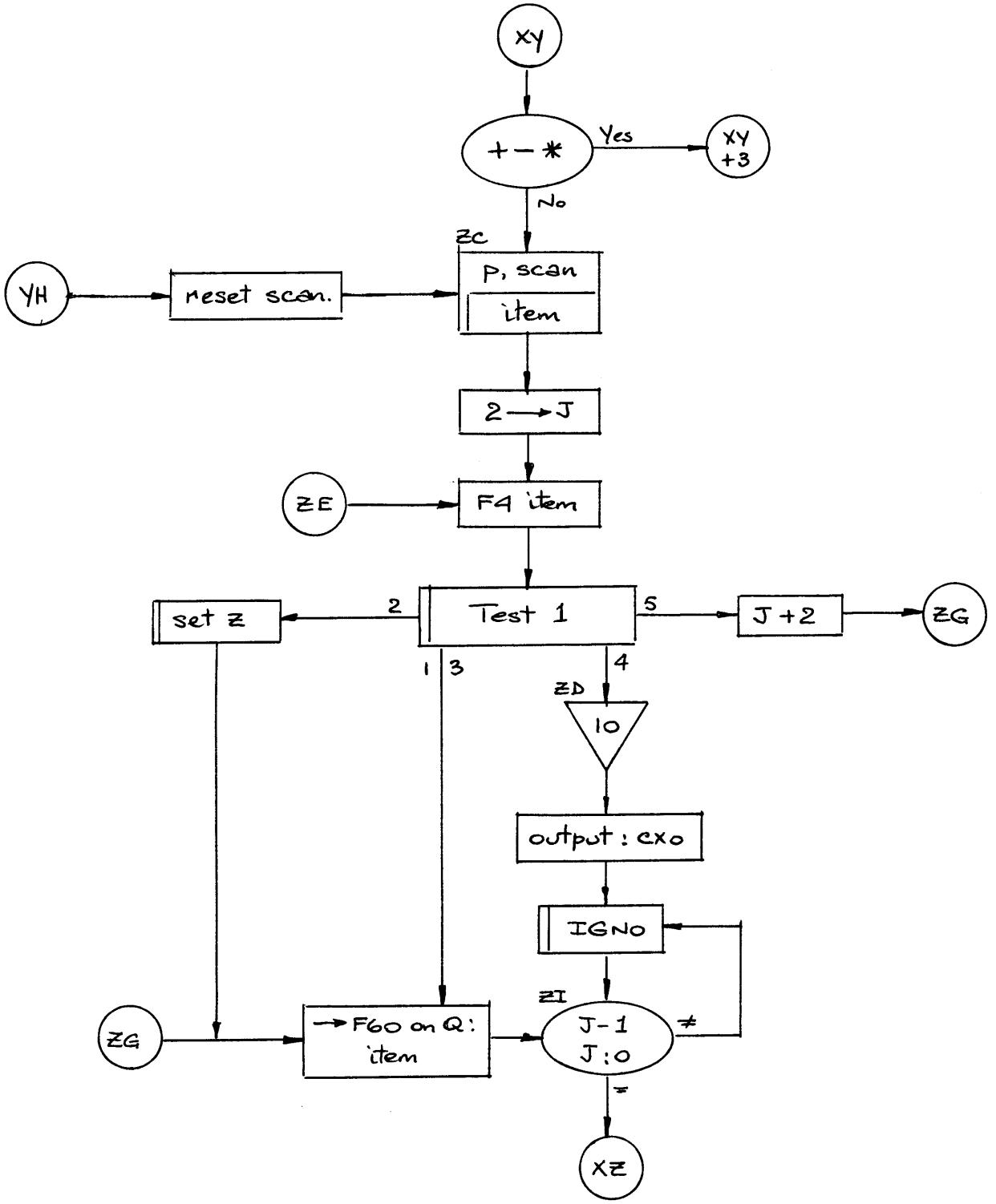
EXAN (continued): Process Simple Term.




3 Term too small

8 Term too large.

EXAN (continued): Process Complex Term.




10 Subscript within subscript.

EXAN (continued): Explanation of Subscript Term Processing.

1. First Pass.

The first pass creates the W table, which contains one item for each item in the subscript term, as follows:

operation:

| | | | |
|---|---|---|---|
| − | T | | $T_W$ |
| | | B | $B_W$ |
| | | A | $A_W$ |

T = 1, 2, 3 for +, −, *
B = W of second operand.
A = W of first operand.

numeric:

| | |
|---|---|
| 1.0 | $T_N$ = floating pt. 1 |
| numeric | $C_W$ |
| X ——— X | |

variable:

| | |
|---|---|
| location of item | $T_W$ |
| numeric | $C_W$ |
| D | |

D = 0 ⇒ induction variable.

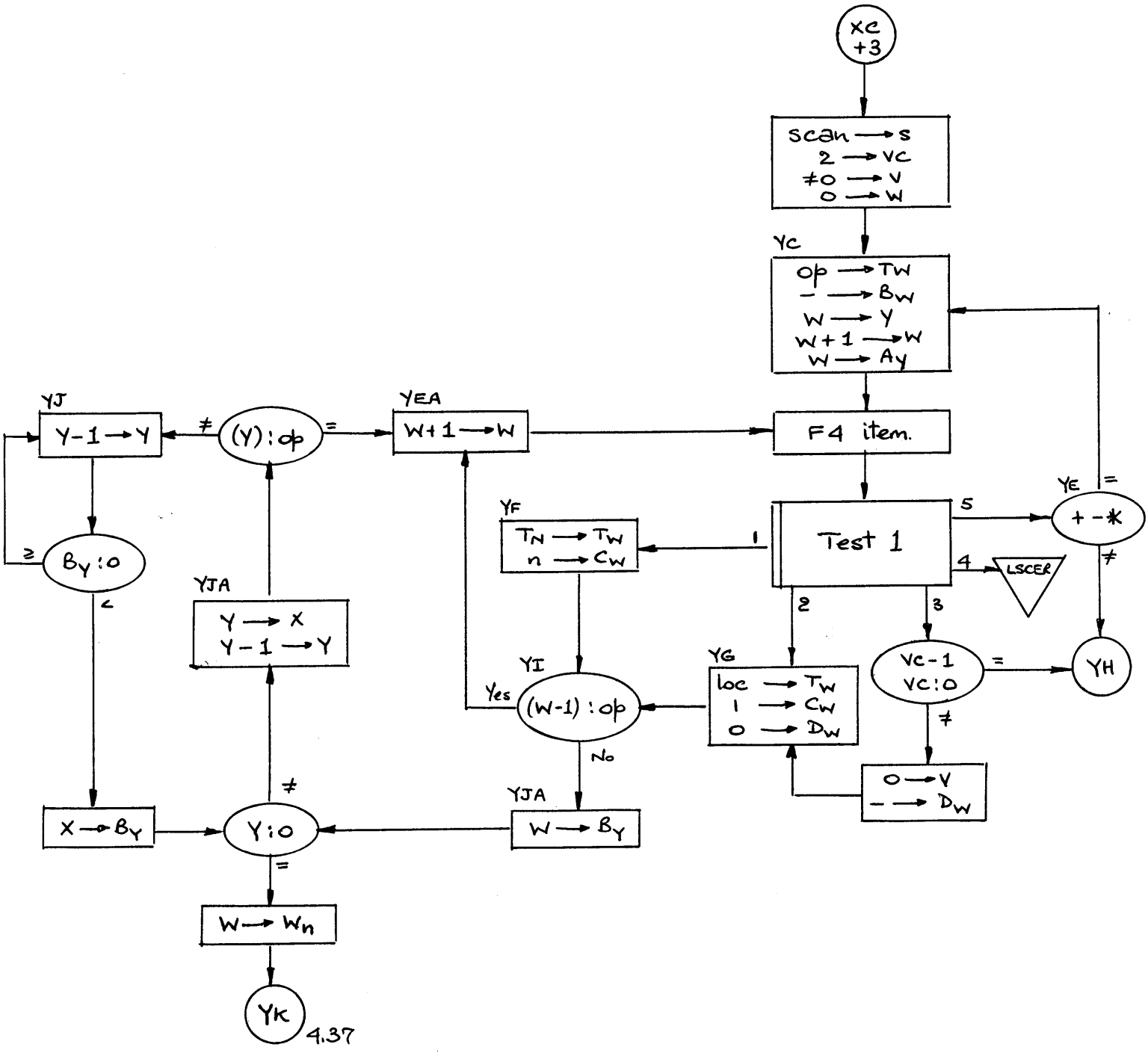
2. Second Pass.

The second pass consists of an inspection from back to front of the operations in the table. Constant arithmetic is done; the coefficients of variables are computed.

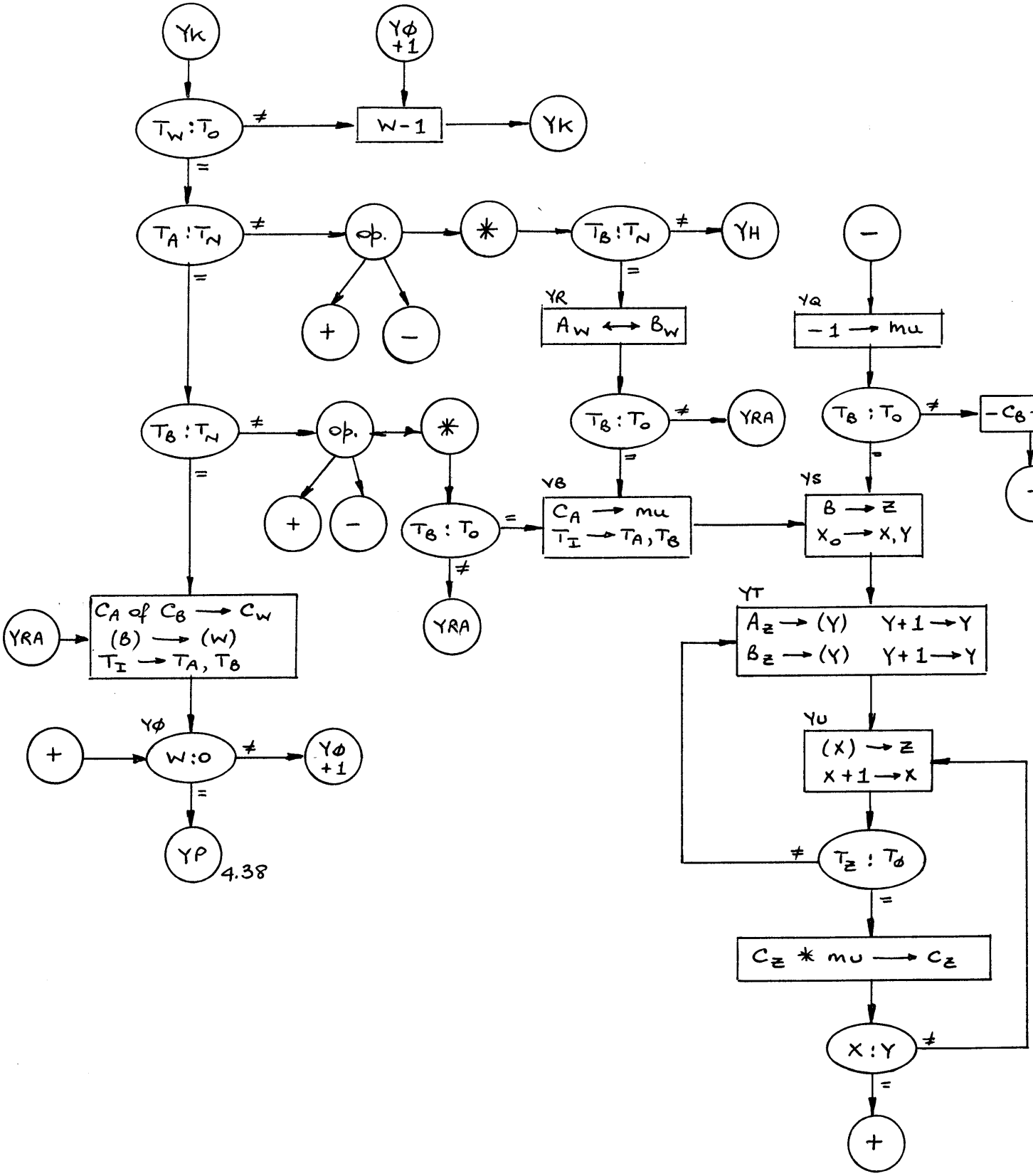
3. Third pass.

Nothing remains in the Table but additive constants and additive variable terms with coefficients. These are output.

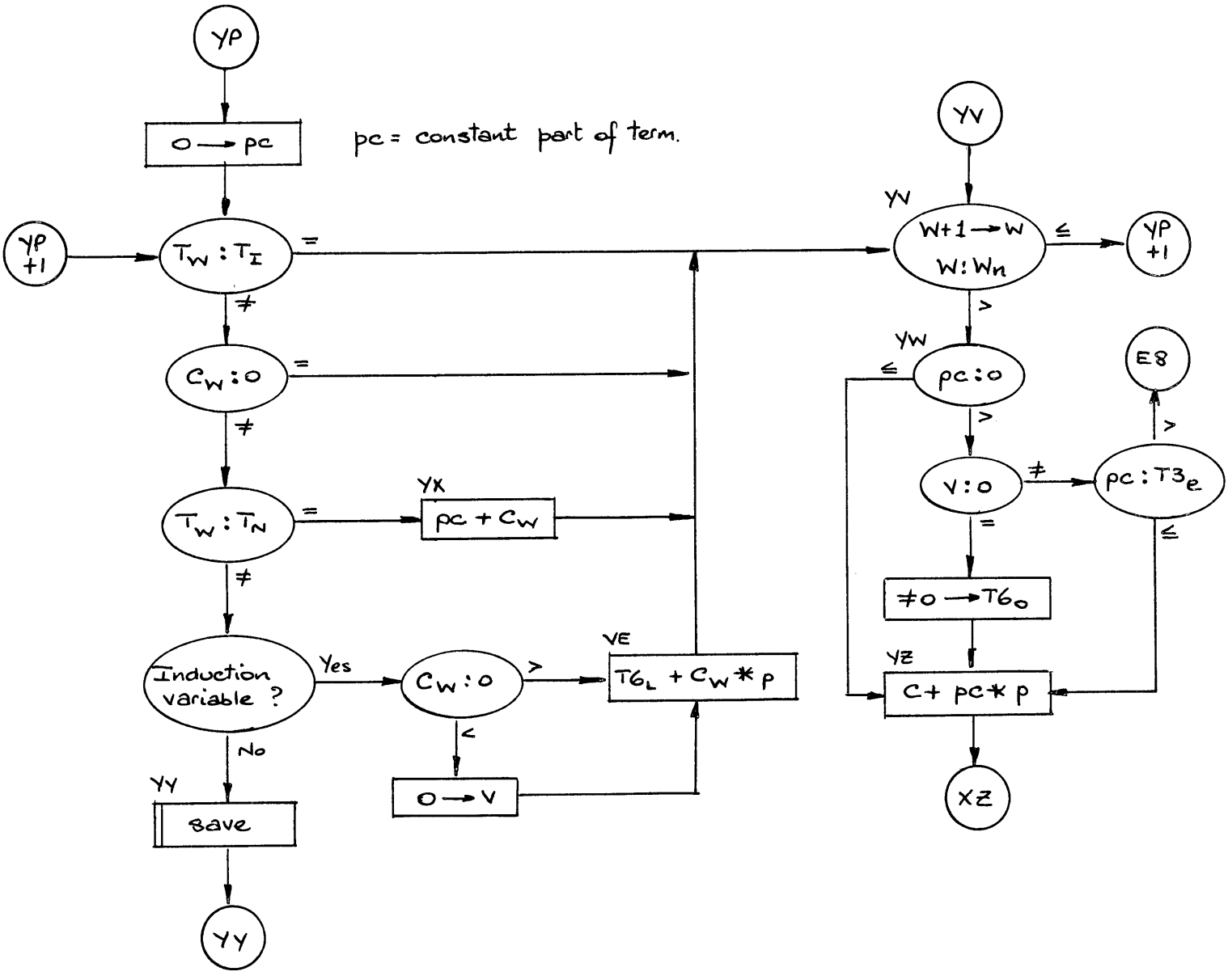
EXAN (continued): Term First Pass.



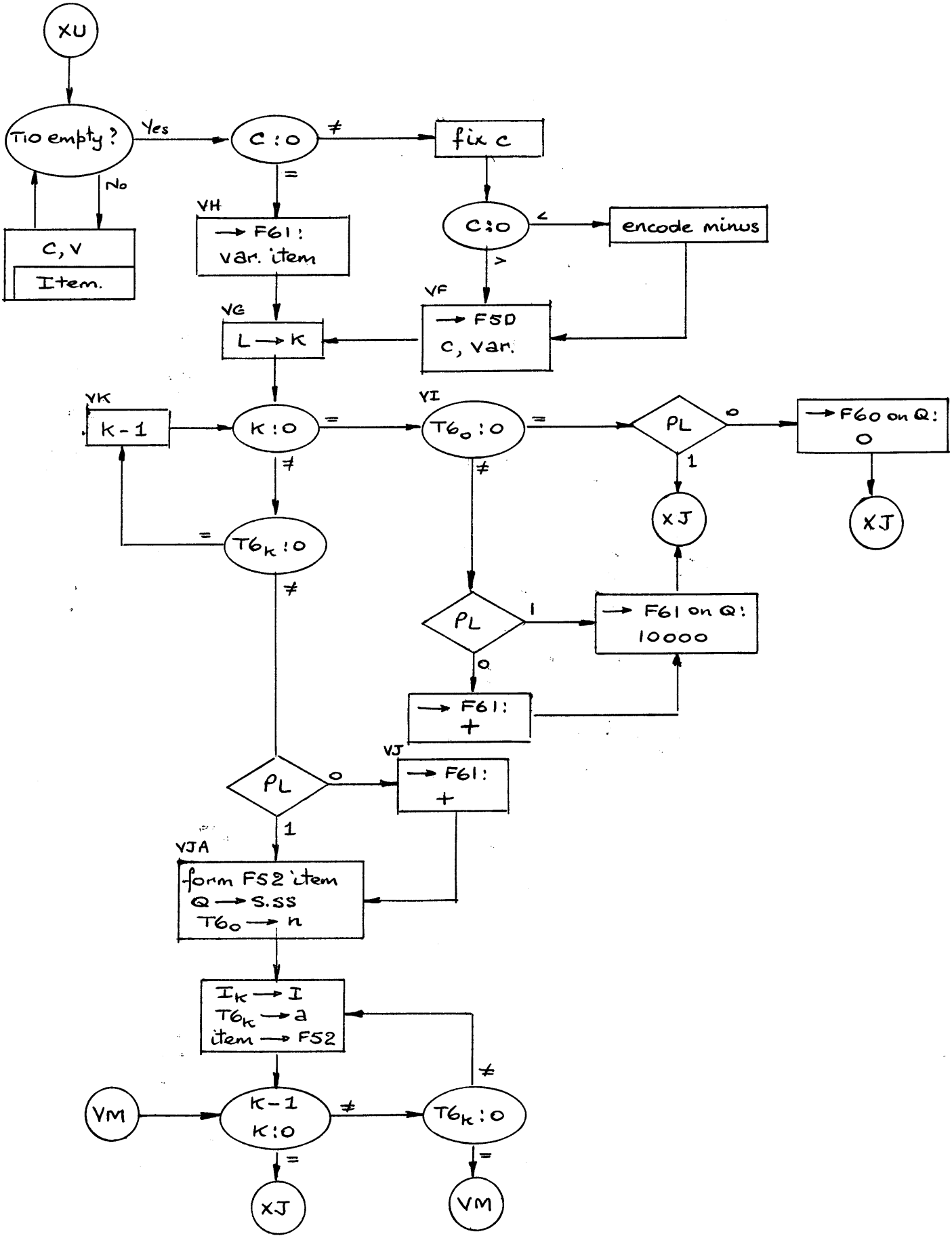
EXAM (continued): Term Second Pass.

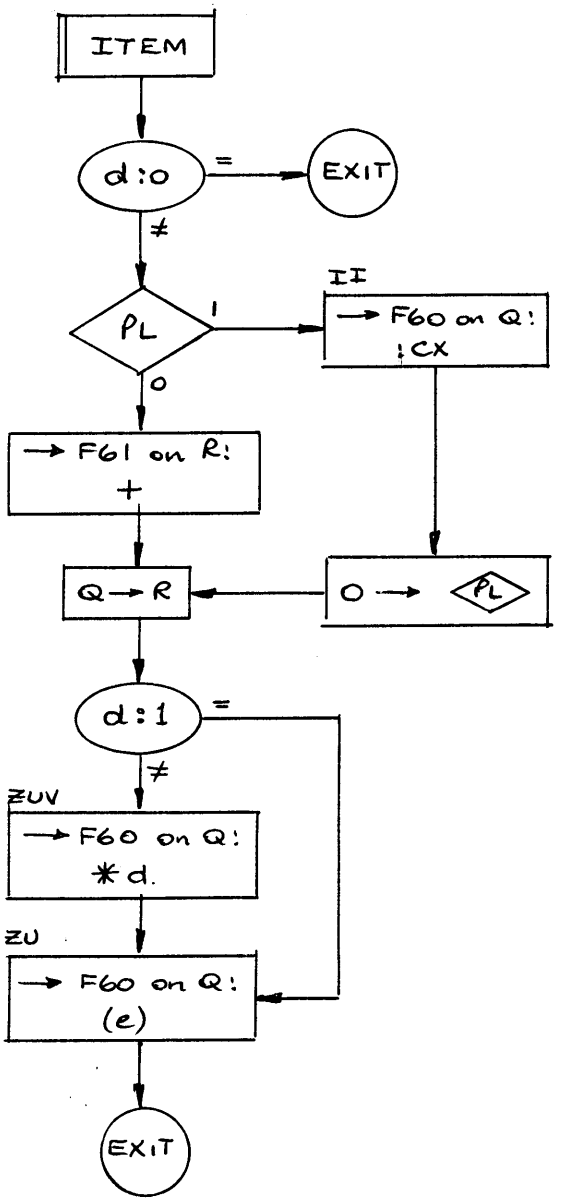


EXAN (continued): Term Third Pass.

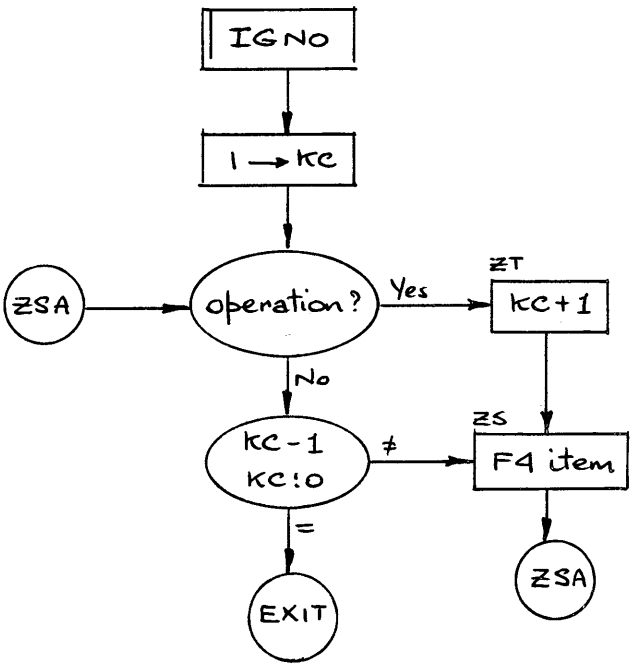


EXAN (continued): Post-Analysis of Subscript.

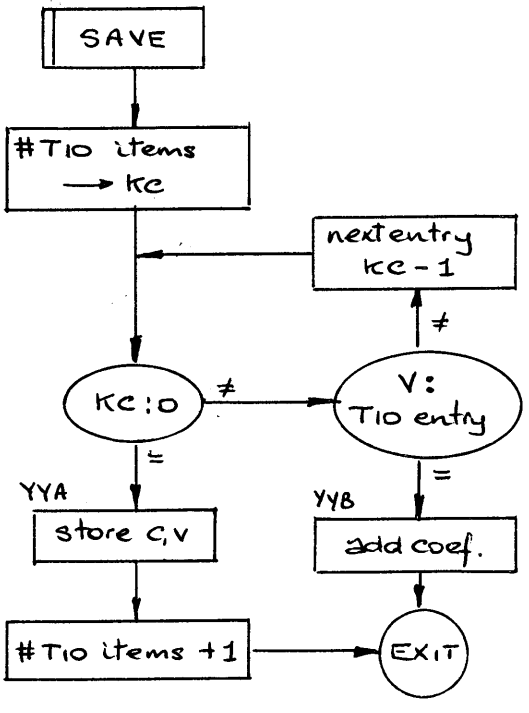


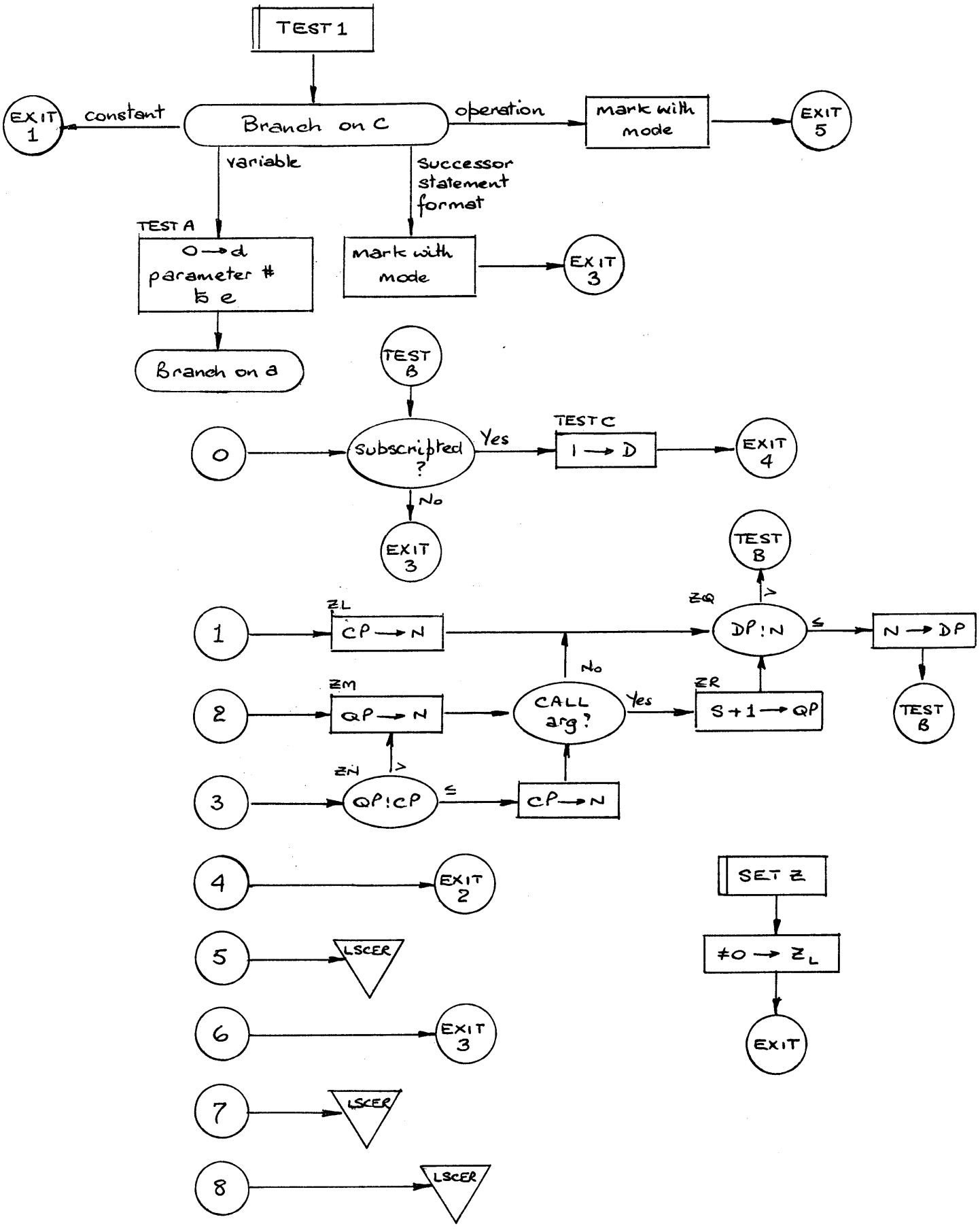


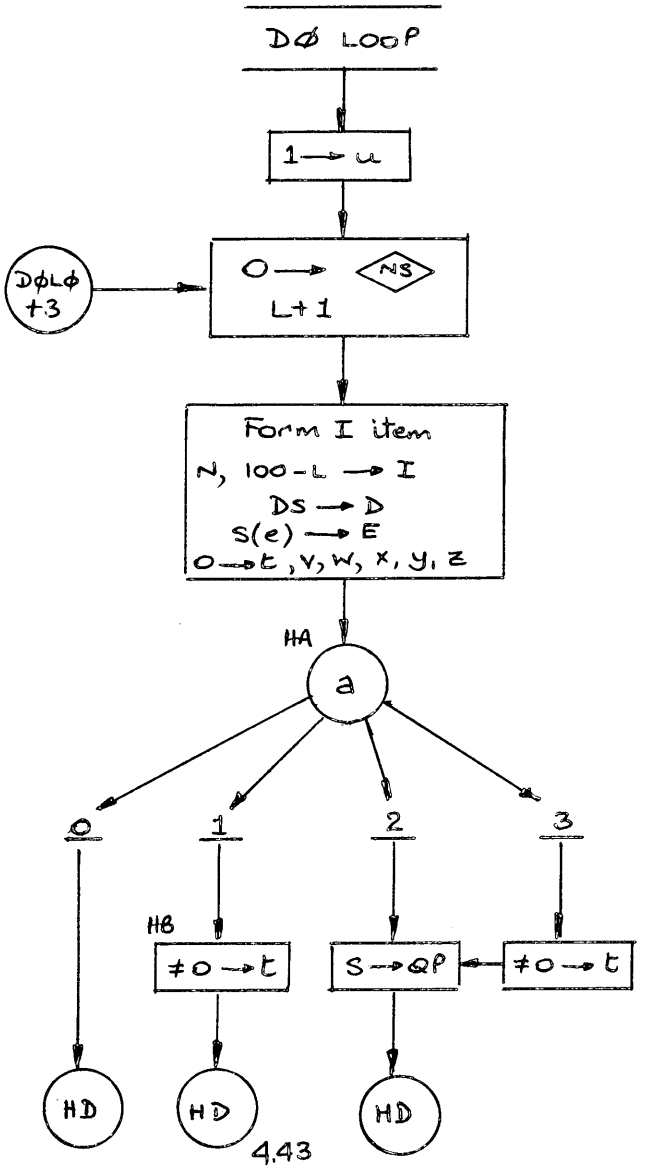

This subroutine sends a variable or punctuation item to F60.
The form of A(X+2Y+I) in Q6 is:
: A $\overset{FX}{+}$ : CX + X * 2YI'
where $\overset{FX}{+}$ represents fx. pt. addition.
and I' is the A register containing I.




This subroutine scans a Polish string without operating on it.




TEST 1
EXIT 1
constant
Branch on C
operation
mark with mode
EXIT 5
variable
Successor statement format
TEST A
0 → d
parameter # ≯ e
mark with mode
EXIT 3
Branch on a
TEST B
0
subscripted ?
Yes
TEST C
1 → D
EXIT 4
No
EXIT 3
TEST B
ZL
ZEQ
1
CP → N
DP : N
≤
N → DP
TEST B
ZM
ZR
No
2
QP → N
CALL arg?
Yes
S+1 → QP
ZN
3
QP : CP
≤
CP → N
4
EXIT 2
5
LSCER
6
EXIT 3
7
LSCER
8
LSCER
SET Z
≠0 → Z_L
EXIT




I = new name of induction variable
D = start of loop
E = end of loop
F = dictionary reference
j = 0, 1, 2, 3 sequence F51 group
t = 0 not common
u = 0 not common range
v = 0 no entrance
w = 0 no exit
x = 0 opt. outer loop.
y = 0 opt. inner loop
z = 0 no compute store.

T9 item

| I | 0 | D |
|---|---|---|
| t, u, v, w, x, y, z | | E |

F51 item

| I | 1 | F |
|---|---|---|
| | | |
| | | |
| | | |

b item

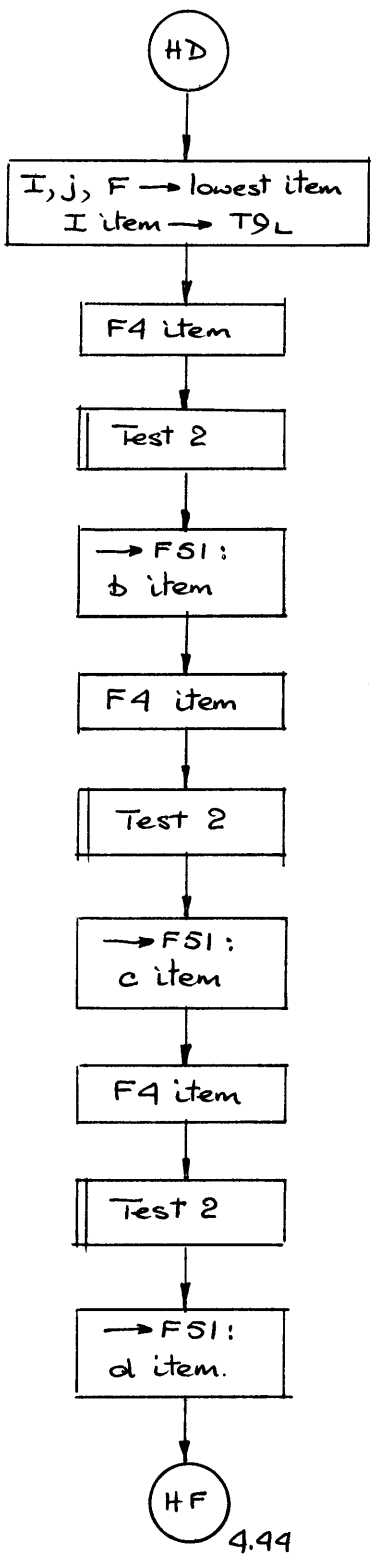
| I | 2 | F |
|---|---|---|
| | | |
| | | |
| | | |

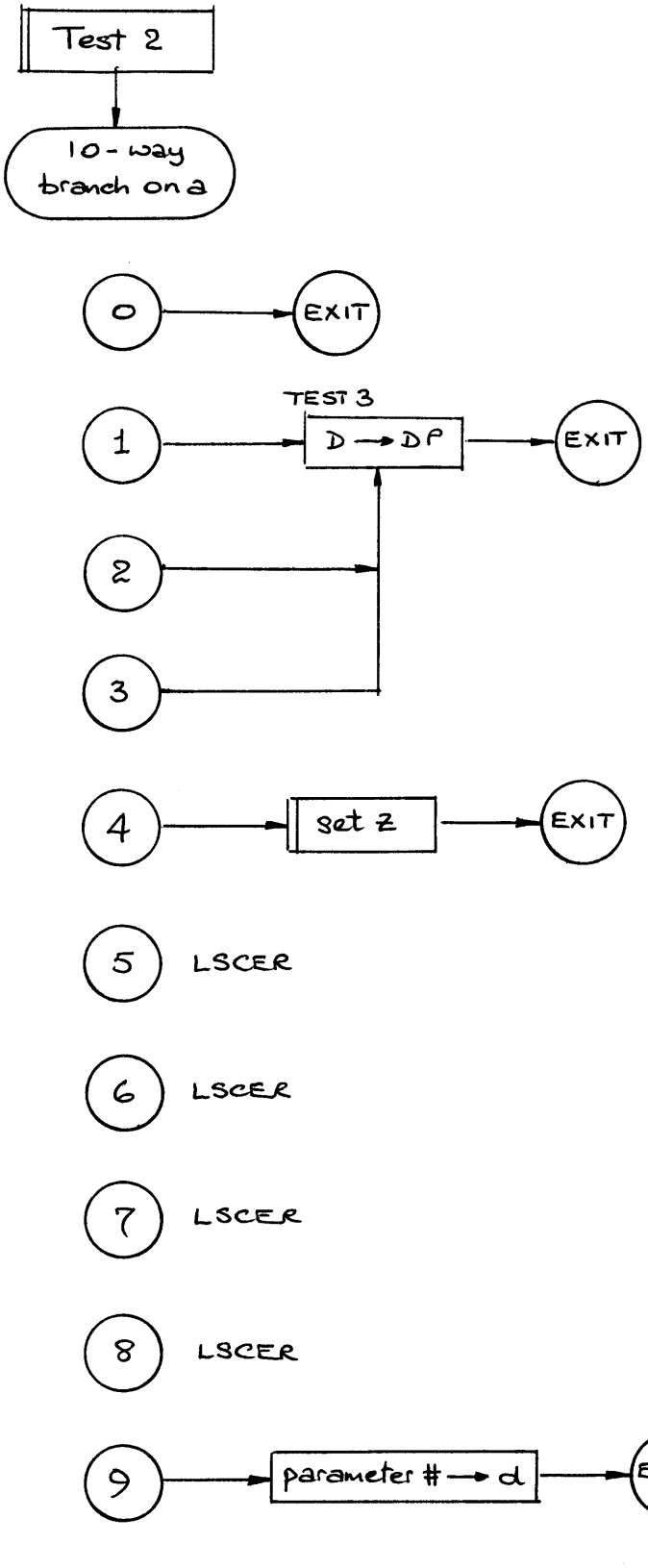
c item

| I | 3 | F |
|---|---|---|
| | | |
| | | |
| | | |

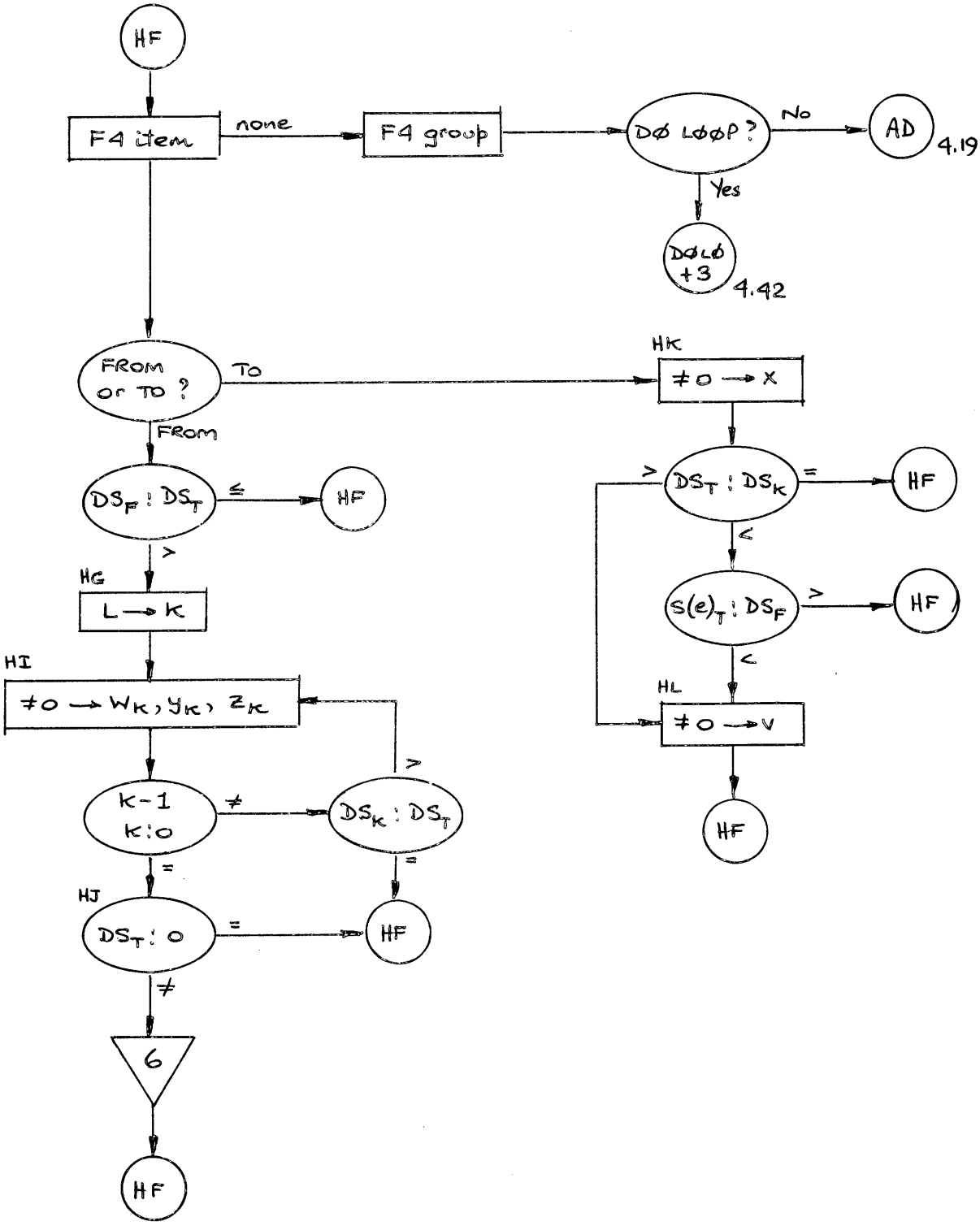
d item

DØ LOOP (continued): Limits.

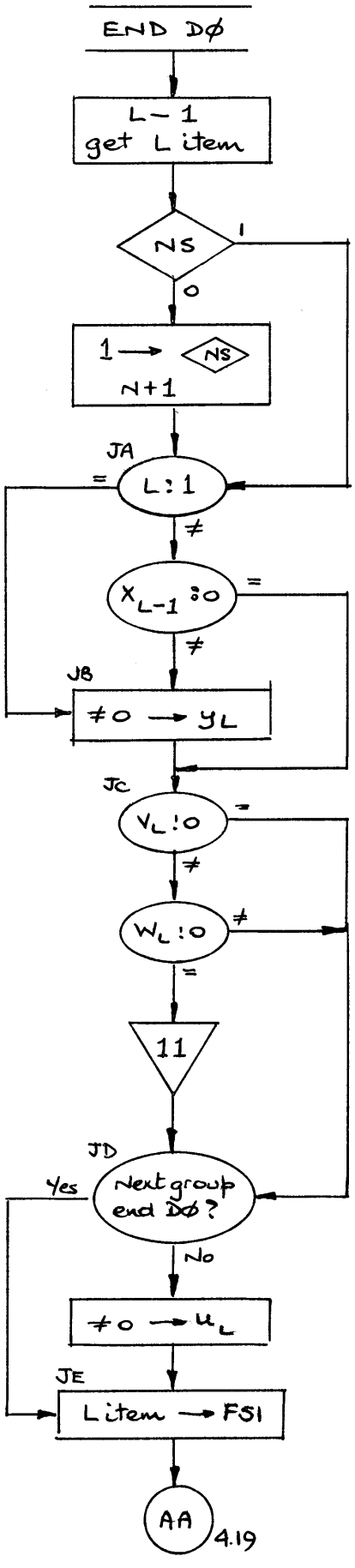


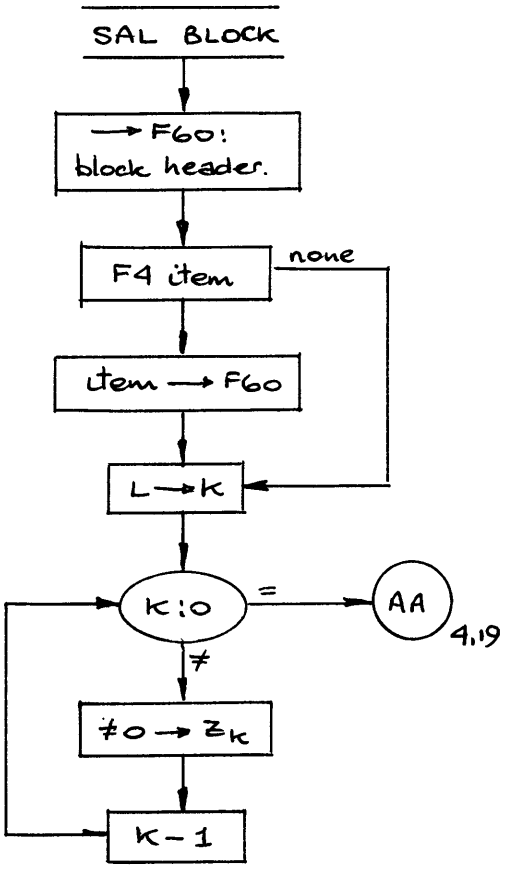



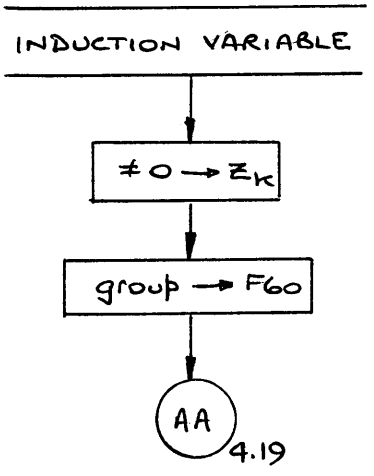
DØ LØØP (continued): TO & FROM items.




6 Illegal transfer from this loop.




11 illegal transfer to this loop.

# PHASE V

PURPOSE:

The main function of Phase V is to optimize an object program with respect to manipulation of indexing quantities. This includes:

1. Renaming subscripted variables containing constants in their subscripts.

2. Creating code-generating items to form induction variables which must be made available to the program.

3. Creating code-generating items to form subscripts containing induction variables.

4. Creating code-generating items to count the number of times through each loop.

INPUT:

Prior to Phase V, Phase III has labeled references to induction variables in the range of the controlling loop as such, and Phase IV has collected a great deal of information about each induction variable, passing this information on in the form of Files 50, 51, and 52.

File 50 contains one item for each variable containing a constant in its subscript. F50 has nothing to do with induction variables; a program without any may still generate a large F50.

File 51 contains a group of items for each loop in the source program, including the old and new names of the induction variable, its range, its limits, exit-entrance information, etc.

File 52 contains an item for each reference to an induction variable in a subscript (unusually complicated subscripts excepted). The item contains the variable's new name, its constant coefficient, the sequence of the reference, and indication whether the subscript is potentially negative.

DATA FLOW:

F50 is read in and its item split to be sent to F61 and F91.

F52 is then input and used to form F53 and F54.

F53 is sorted and processed to form F55 and F56.

F55 is sorted. Then it and F56 are processed in alternating groups by F51 and F54.

When F51 is exhausted, Phase V is finished.

The forms of F50, F51, and F52 are described in an Appendix to the description of Phase IV. The internal files are described in Appendix A. The processing is described in detail below.

Note: Phase V has its own contingency routine.

PHASE Va: RENAME VARIABLES AND CONSTANT SUBSCRIPTS.

Example

Source statement: A (5) = A (I+5) + B (2*J-1)

Object statements:

```
Fdd  #0   A+5   #1   (I)
A    #0   B-1   #2   (2J)
S    #0   A+5
```

F50 contains the pairs A, +5; B, -1; A, +5; these become the M-fields of the object code.

Phase Va's function is to file each pair once in the dictionary (F91) and replace each F50 item for the variable (A, B) with an item for the variable-subscript combination (A+5, B-1).

A F50 group containing identical constants and variables is read in. The constant and variable are sent to F91. The dictionary reference is attached to the mode word of each F50 item, and the item is passed to F61. (The size of F50 and the number of dictionary words required is greatly reduced for most programs by the Phase IV device of starting subscripts from 0 instead of 1.) At the same time subprogram arguments containing negative constants are marked for Phase VII.

PHASE Vb: NUMBER SUBSCRIPT TERMS

Example

Source statements:

```
DIMENSION A(3, 4, 5), B (3, 4, 5,), C (3, 4, 5)
          :
A (I, J, K) = B (I+1, J+1, K+1) + C (I+2, J+2, K+2)
where I, J, K are induction variables.
```

Object statements:

```
F  #0  B+16  #1
A  #0  C+32  #1
S  #0  A     #1
```

Phase V insures that multiply-referenced subscripts (I, J, K in the example) are computed only once by naming "identical" subscripts alike. The names are sent to F61 to be merged into their proper places in the Polish string.

Each term of a subscript contains an induction variable and a coefficient (11 digits of information). The first step in eliminating duplicate subscripts is to condense the information given into four-digit numbers.

Input comes from F52 in groups by term. The term of the group is filed in F54; its number goes to F53 once for each reference to it.

F53 is sorted so that within each subscript, the terms appear "inside out."

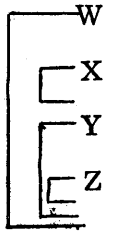
Subscripts containing 1-3 terms go to F55; larger subscripts to F56.

F55 is sorted and now the major work of Phase V begins. All its files are formed; they must now be processed in interrelationship with each other. Output will be to F61 only in the form of both Polish strings and pseudo-code.

PHASE Vc: FILE LOOPS BY LEVEL

The loops shown have been numbered by Phase IV as follows:

| | |
|---|---|
| W | 000 99 |
| X | 000 98 |
| Y | 001 98 |
| Z | 001 97 |




Phase V will process loops W and X and the subscripts within them completely before beginning work on loops Y and Z. When loops Y and Z are processed, all information on loop W will still be available for use in forming subscripts containing terms from Y and W, Z and W, or all three.

Phase Vc reads F51 and files the 16 words for each loop with minor modifications in a table (where each remains until replaced by another loop of the same level or until Phase V is over.) At this point also 3 sequence counters are initialized for each loop:

C (initialization outside the loop); D (incrementation inside the loop); and E (looping logic at the bottom of the loop).

PHASE Vd: UNPACK TERMS

The F54 items for the loop, if any, are input for use by the subscripts and unpacked into more convenient form by a scan.

PHASE Ve: FORM NON-CONSTANT COUNTS; MATERIALIZE INDUCTION VARIABLES

Each loop is now examined in turn. If its initial, final, and incremental values (b, c, and d, respectively in DØ S, I=b, c, d) are all constant, its count is constant. The fastest object code for counting and looping would load one or more A-registers with counts outside the loop, then loop with a BIT command. This is what Phase V does, as described later.

Unfortunately, this method is impractical for variable counts. Phase Ve produces instead code to initialize an A-register to (b-d) outside the loop and to add d and test against zero at the end of the loop.

If the induction variable must be made available to the program (the conditions are described in Phase IV), it is "materialized" by creating the instructions F #n b, S #n I, A #n d, at C, D, and E respectively. (See bottom of Vc.)

PHASE Vf: FORM NON-CONSTANT INITIAL AND INCREMENTAL MULTIPLES; CHECK ERRORS

The information of initial (a*(b-d)) and incremental (a*d) multiples of subscript terms (where a is the coefficient of an induction variable in a subscript) is a space-and-time-consuming process (especially when, as is often the case, there is more than one a to be multiplied). Phase Vf effects the production of code to form these multiples as many levels before use as possible, and saves the "names" of the A-registers in which they are formed for later reference.

Phase Vf also evaluates the count if it is constant, checks that the limits of the loop are reasonable and saves optimization information.

PHASE Vg: FORM SUBSCRIPTS

The subscripts are now in F55 and F56 in slightly different forms. Associated with each subscript are its sequence and subsequence and the Phase Vb-assigned numbers of its terms. The indication whether or not it is potentially negative is still attached.

All F55 subscripts whose innermost terms contain the current induction variable are processed, then all F56 subscripts of this type. The processing for the files is similar:

1. The subscript is checked to see if it is contained in a current loop.
2. The terms are filed in a table.
3. The negative indicator is saved.
4. Item(s) are sent to F61 naming the subscript at each reference to it.
5. The terms are then processed by the subscript subroutine as described in Appendix I.

PHASE Vh: CONSTANT COUNTS

(A reading of Appendix I will clarify this section.)

Constant counts may not be formed before Phase Vh because:

1. In some cases two or more adjacent constant counts may be combined, but these cases can't be distinguished until after Phase Vg.

2. In some cases the BIT instruction which counts may also be incrementing a subscript provided by Phase Vg.

Phase Vh consists of a final scan of the loops input in Phase Vc, from the inside out, with attention to constant counts only.

Tests are made as described below to determine whether the loop can be collapsed with the next outside loop. If not, items to fetch and count down an A-register containing the count modulo 1000 and ad and ab fields, if any have been saved, are output. As many other A-registers as necessary are loaded and "BIT"-ed to complete the count. (The program takes account of the fact that setting up multiple BIT's is not straightforward when any of the partial counts is zero.)

If two loops can be collapsed, no items are output; the product of their counts becomes the count of the outer loop, and if the inner loop's BIT command was to have incremented a subscript, the outer's now will.

Two loops can be collapsed only if:

1. The inner loop is optimum.
2. Their counts are constant.
3. They include exactly the same executable statements.
4. All subscripts containing either induction variable contain both in such a proportion that one complete execution of the inner loop increments the subscript by the amount of one step in the outer loop.

After the last loop's count is inspected, the program returns to Phase Vc to process the next nest. When F51 is exhausted, Phase V is through.

PHASE V - Appendix A - SUBSCRIPT SUBROUTINE

Subscripts are formed in A-registers. Through most of Phase V a five-digit counter serves as a source of A-register names. The counter is initialized at the start of each level-one nest of loops. A-registers are named in such a way as to minimize use of index registers for arithmetic by Phase VIII.

Each subscript is compounded of terms of the form ad and a(b-d) (where b and d may be variable) and constants from four other sources:

1. 100000 - added at the first initialization of a potentially negative subscript.

2. The ad, ab parts of a counting A-register - added as that term's loop is executed.

3. The count of a counting A-register - fetched originally or added later.

4. The total increments of optimum loops with constant limits - added and subtracted as described below.

Like everything else in Phase V, subscripts are examined from the inside out, for only in that way can the optimacy of each loop in turn be checked.

The following examples show the operations produced for several subscripts, where the successive P numbers are successive A-register "names."

Example 1: Single Term

at C: P1 = a1 (b1 - d1)

at D: P1 = P1 + a1 d1

Example 2: Two Terms

at C1: P2 = a1 (b1 - d1)

at D1: P2 = P2 + a1 d1

at C2: P1 = P2 + a2 (b2 - d2)

at D2: P1 = P1 + a2 d2

In the above examples, ad or a(b-d) may be variable; if so it is referred to by the name assigned its register in Phase Vf.

Example 3: Two Terms, Optimizable

"Optimizable" means:

1. Inner loop optimum.
2. Terms from 2 successive loops
3. d of outer loop constant and total increment t of inner loop calculable. t = dk where count k is constant.

| | | |
|---|---|---|
| at C1: | P1 = a1 (b1 - d1) + a2 (b2 - d2) + a2 t2 | 1 - 2 commands |
| at D1: | nothing | |
| at C2: | P1 = P1 + a1 d1 - a2 t2 | 1 command |
| at D2: | P1 = P1 + a2 d2 | 1 command |

Formidable though the analysis seems, the commands generated are minimal, because all operands are A-registers or constants.

Example 4: Completely General Subscript

| | |
|---|---|
| at D, inner loop: | P = P + ad |
| at C, optimizable loops: | P = P + ad(outer) - at(inner) |
| at C, non-optimizable loops: | P = P' + (all waiting a (b - d) s) + (BB, if any) |
| at C, outer loop: | P' = (all waiting a (b - d) s) + (all mt's added later) + (100000 if applicable) |

The Form Subscript Subroutine forms code-generating items to do the work described in the examples. In doing so, it sends several new types of items to F61. (New items are described in Appendix B.)

PHASE V - Appendix B - NEW ITEMS OUTPUT

A. Items in Phase IV form:

1. Fixed-point multiply operation:

W1 - SSSSS sssss 49 S = Sequence
W2 - 00620 07000 00 s = subsequence
W3 - 0————————0

Used to form ad, a (b-d) multiples.

2. Fixed-point add operation:

W1 - SSSSS sssss 49
W2 - 00620 08000 00
W3 - 0————————0

3. Convert-to-fixed-point function:

W1 - SSSSS sssss 47 V = scale factor
W2 - 70700 VV00013
W3 - 0————————0

Used to fix d, b-d before forming multiples.

4. Integer in integer form:

W1 - SSSSS sssss 45
W2 - 0420 000 00000
W3 - integer

Used for adding subscript terms to A-registers.

5. BB literal

W1 - SSSSS sssss 45

W2 - 05200 00000 00

W3 - BB

Used for initializing counts of loops with constant counts.

B. Pseudo-code items:

1. Register name item

W1 - SSSSS sssss 58 A = A-register "name"

W2 - 0000001AAAAA

W3 - 0————————0

Used to name register in which following expression is to be calculated.

2. Register Definition Item

W1 - SSSSS sssss 57

W2 - 0000001AAAAA

W3 - 000LL00TTTTT

Used as a signal to Phase VIII that an A-register has been defined.

LL = inverse level modulo 100.

T is zero except after BIT command; then it is the sequence to which the transfer is made.

3. Add -- same with ID = 52

Used to increment induction variable.

4. Add fixed-point -- same with ID = 54

Used to increment subscript.

5. Negate fixed-point -- same with ID = 59.

6. Store -- same with ID = 53

Used to store induction variable in memory.

7. Transfer Less than Zero

W1 - SSSSS sssss 51 C = numeric part of statement
W2 - 0000001 AAAAA name created by Phase IX
W3 - CCCCC0000000

8. BIT -- same with ID = 56.

9. Store to fast register

W1 - SSSSS sssss 21
W2 - 0
W3 - 0002001AAAAA

PHASE V - Appendix C

FORMAT OF INTERNAL FILES

| | | |
|---|---|---|
| F53: | SSSSS sssss LL | L = level |
| | 0000WWWW000n | n = negative indicator |
| | | W = name of term |
| F54: | IIIII 0 aaaaaa | I = new name of variable |
| | | a = coefficient |

F54 (unpacked):

± aaaaa 0 KKKKK  K = F51 table ref

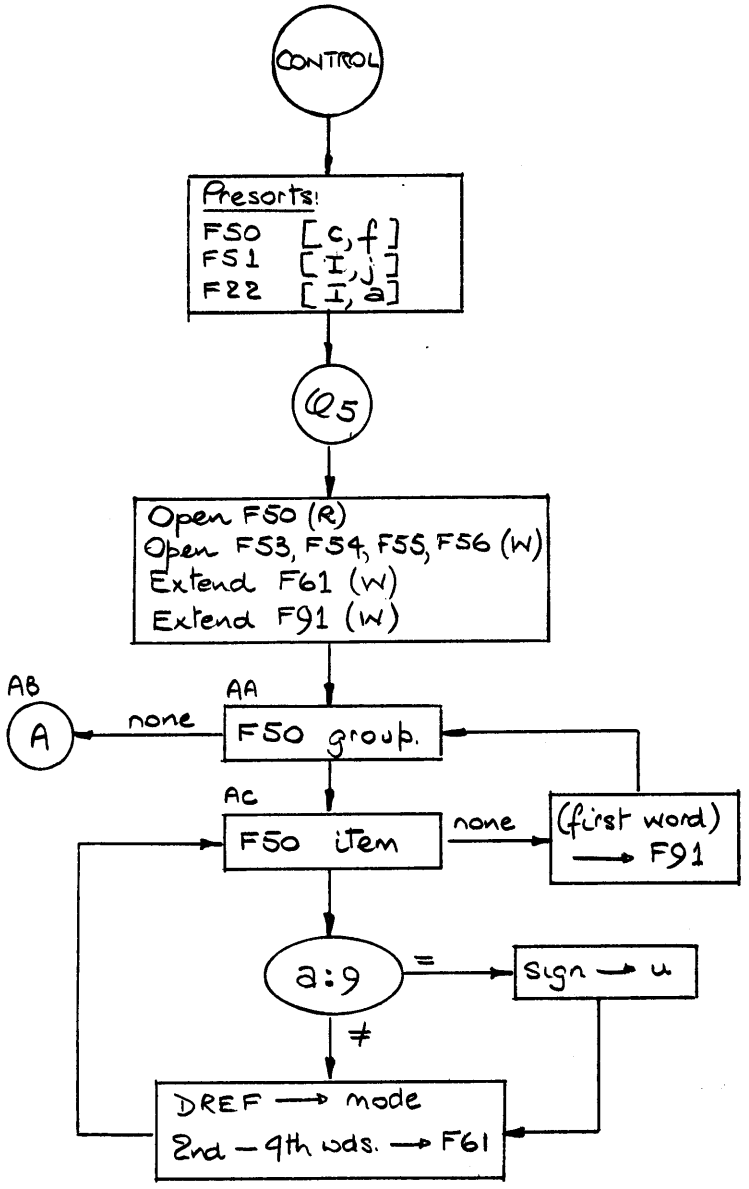
F55: W1 W2 W3

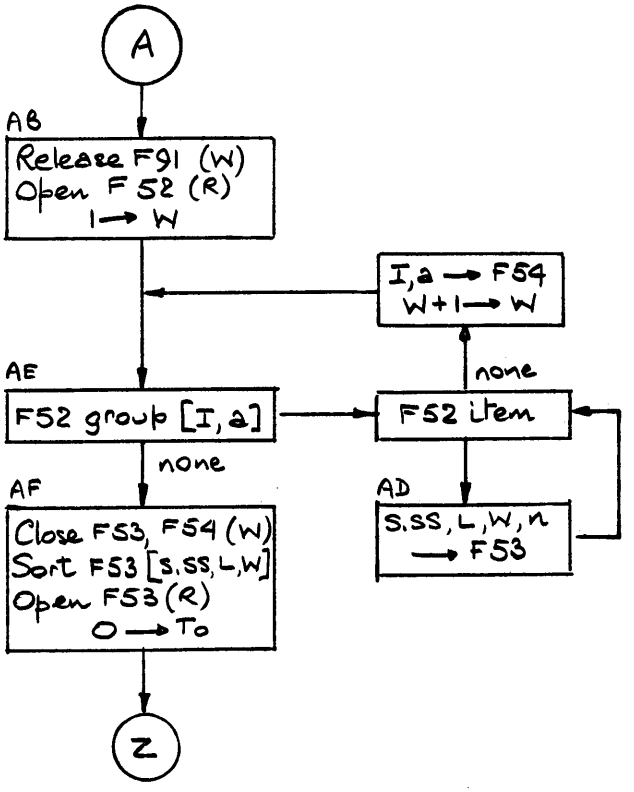
SSSSS sssss 0n

F56: W1 W2 W3

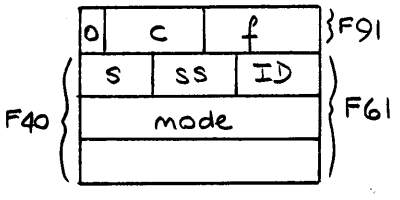
-n SSSSS sssss

One or more a) items followed by one or more b) items, followed by a zero word.

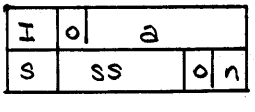



F50 item

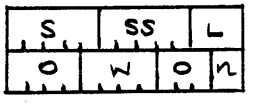



F52 item

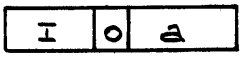


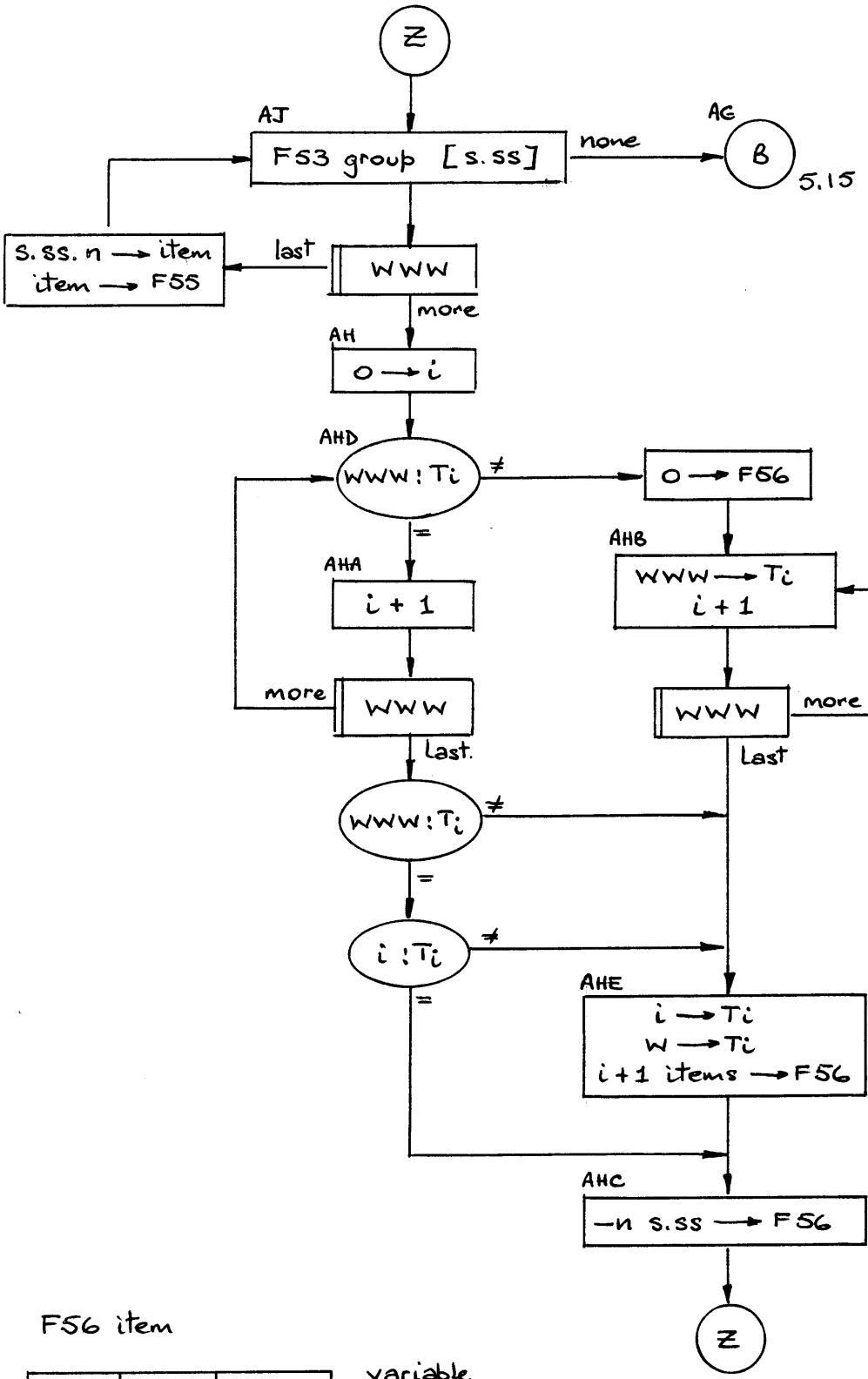

I = n 000-999
L 99-0

F53 item

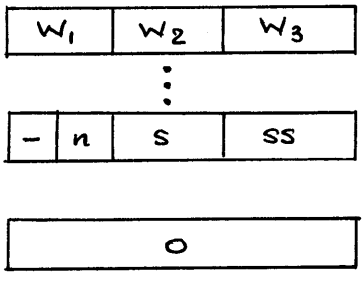



F54 item.

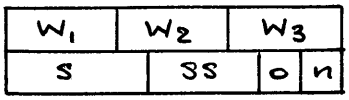


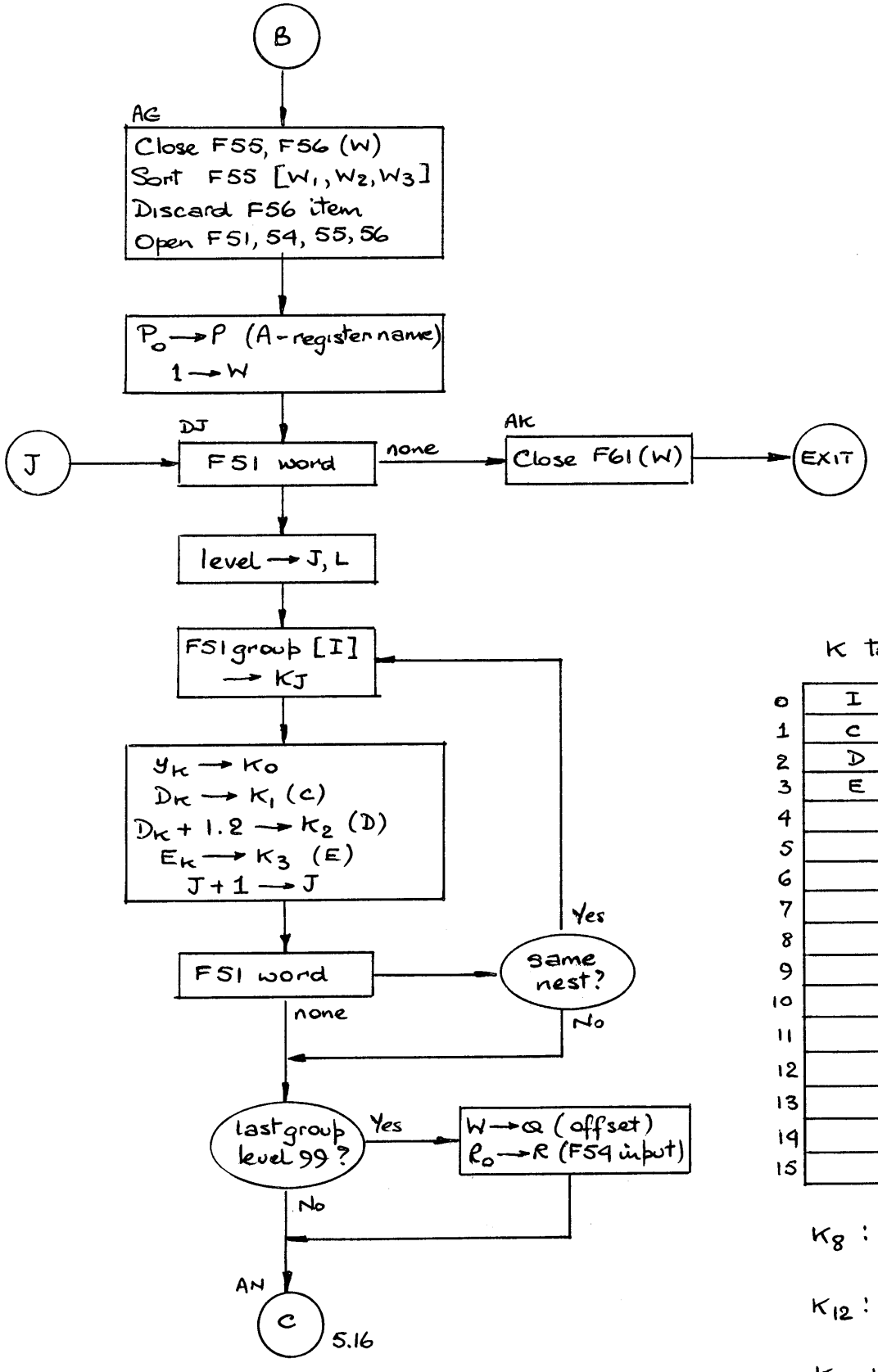



F56 item




F55 item





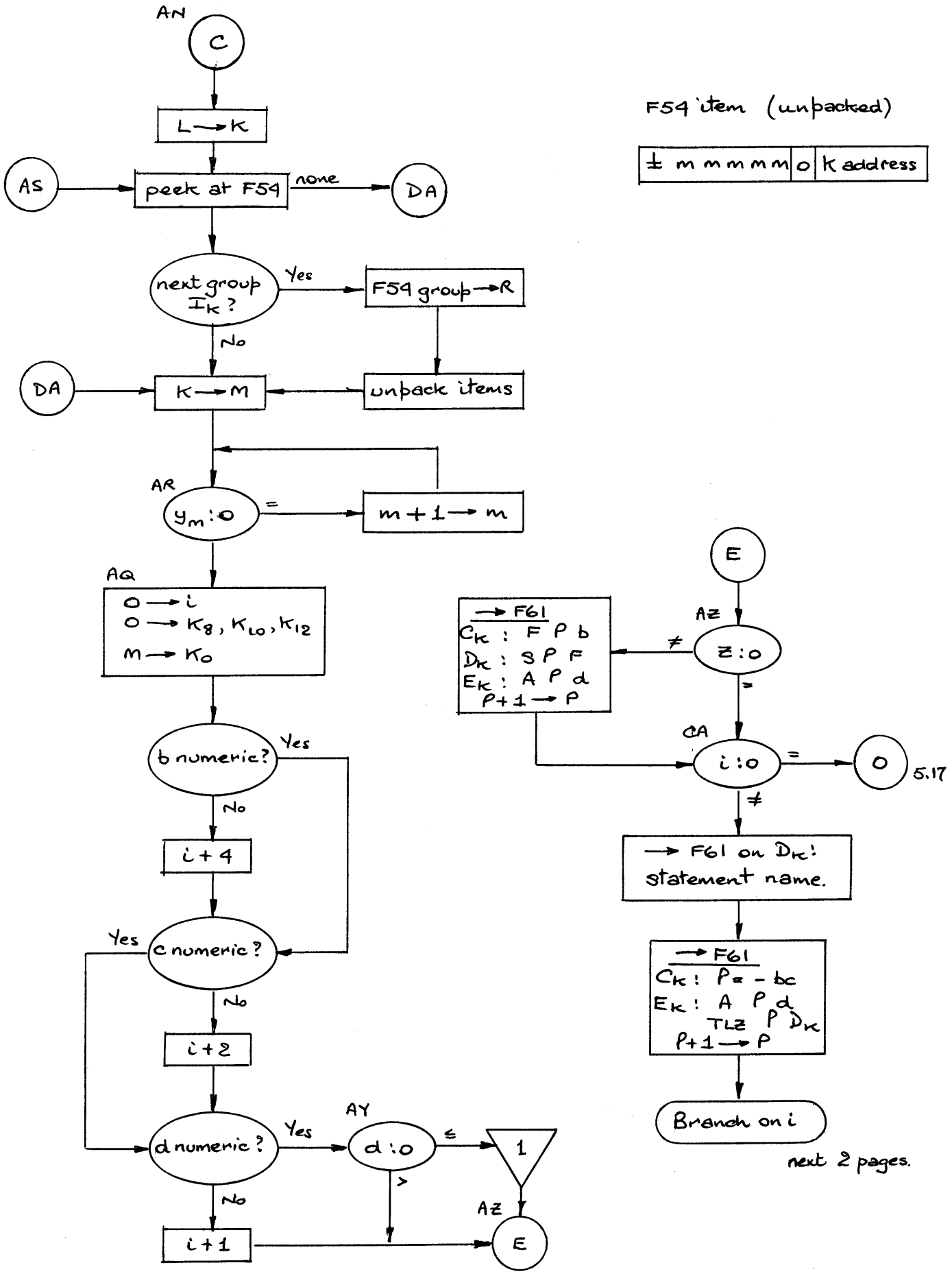

K table item

| | | | | |
|---|---|---|---|---|
| 0 | I | y, u | m | |
| 1 | C | | | |
| 2 | D | | | |
| 3 | E | | | |
| 4 | | z | F | |
| 5 | | | ID | b |
| 6 | mode | | | |
| 7 | value | | | |
| 8 | | | | |
| 9 | | | ID | c |
| 10 | mode | | | |
| 11 | value | | | |
| 12 | | | | |
| 13 | | | ID | d |
| 14 | mode | | | |
| 15 | value | | | |

$K_8$ : dk for i = 0, y = 0
otherwise 0

$K_{12}$ : k for i = 0
otherwise 0

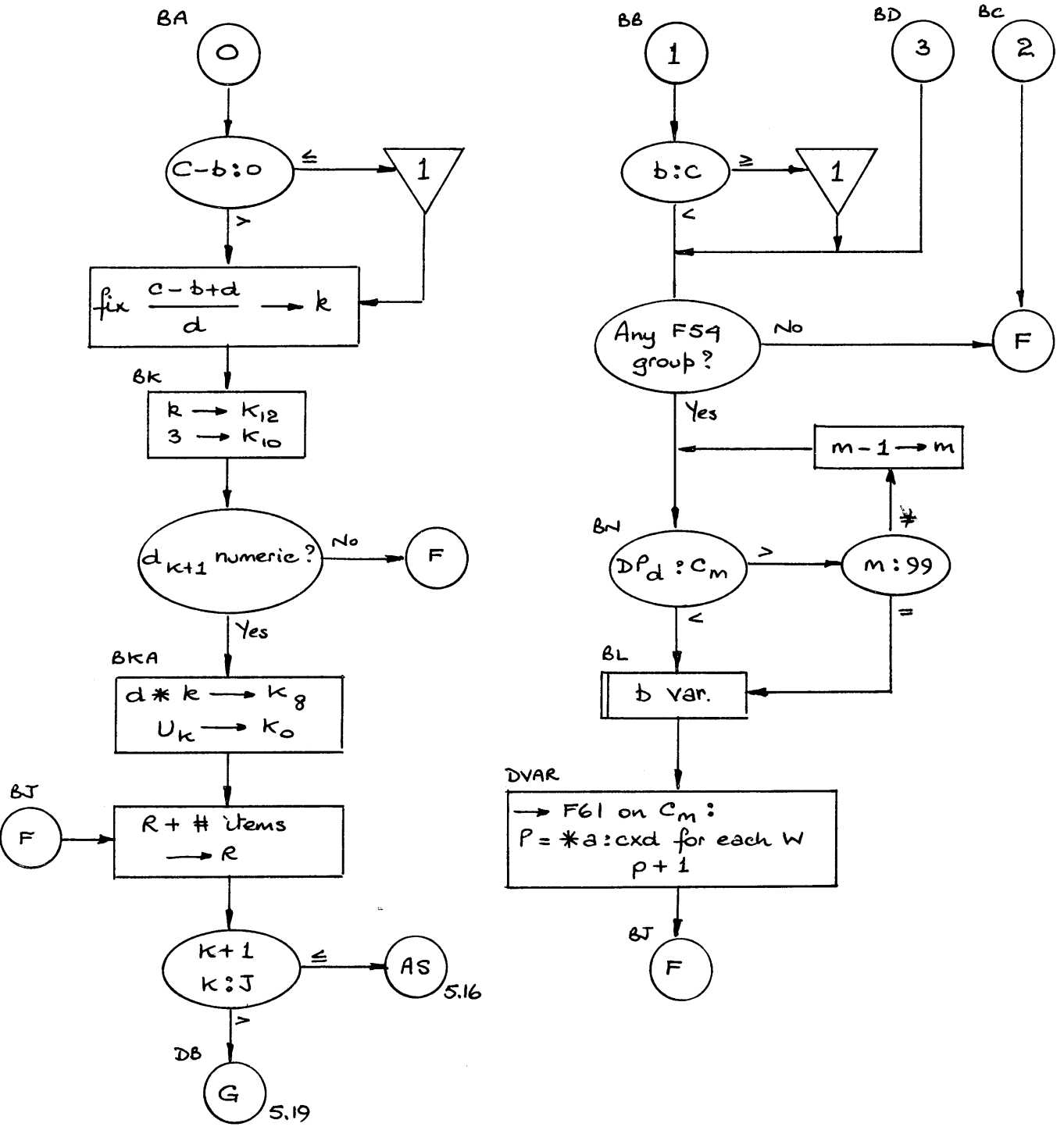
$K_4, K_9, K_{10}, K_{11}$ used for BIT information.



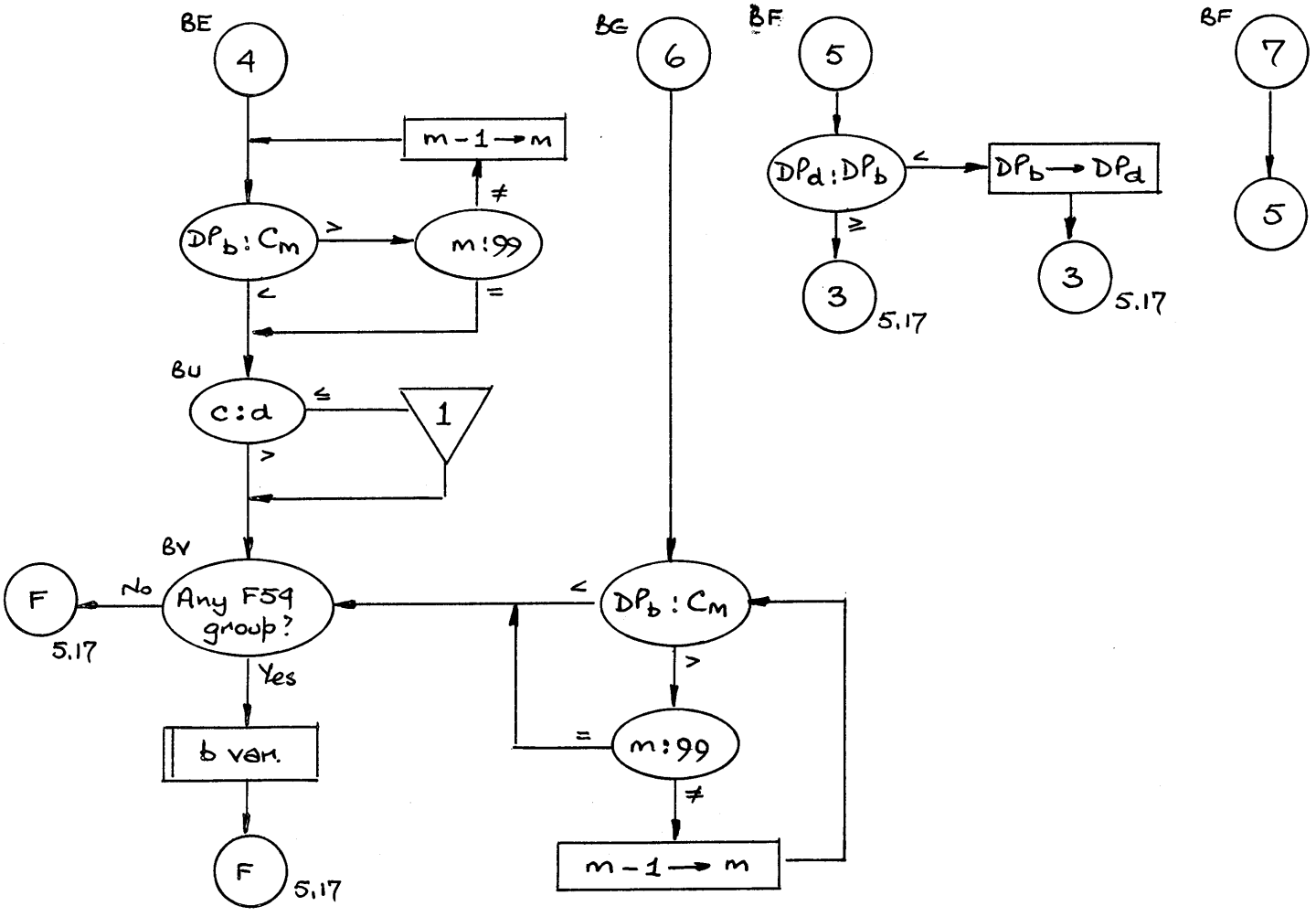

F54 item (unpacked)

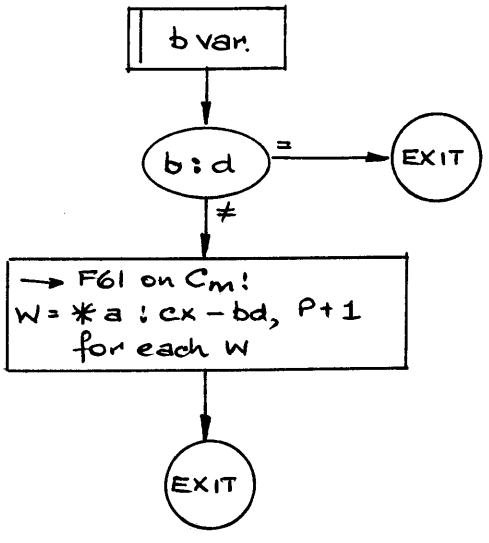
| ± | m m m m m | 0 | k address |
|---|---|---|---|

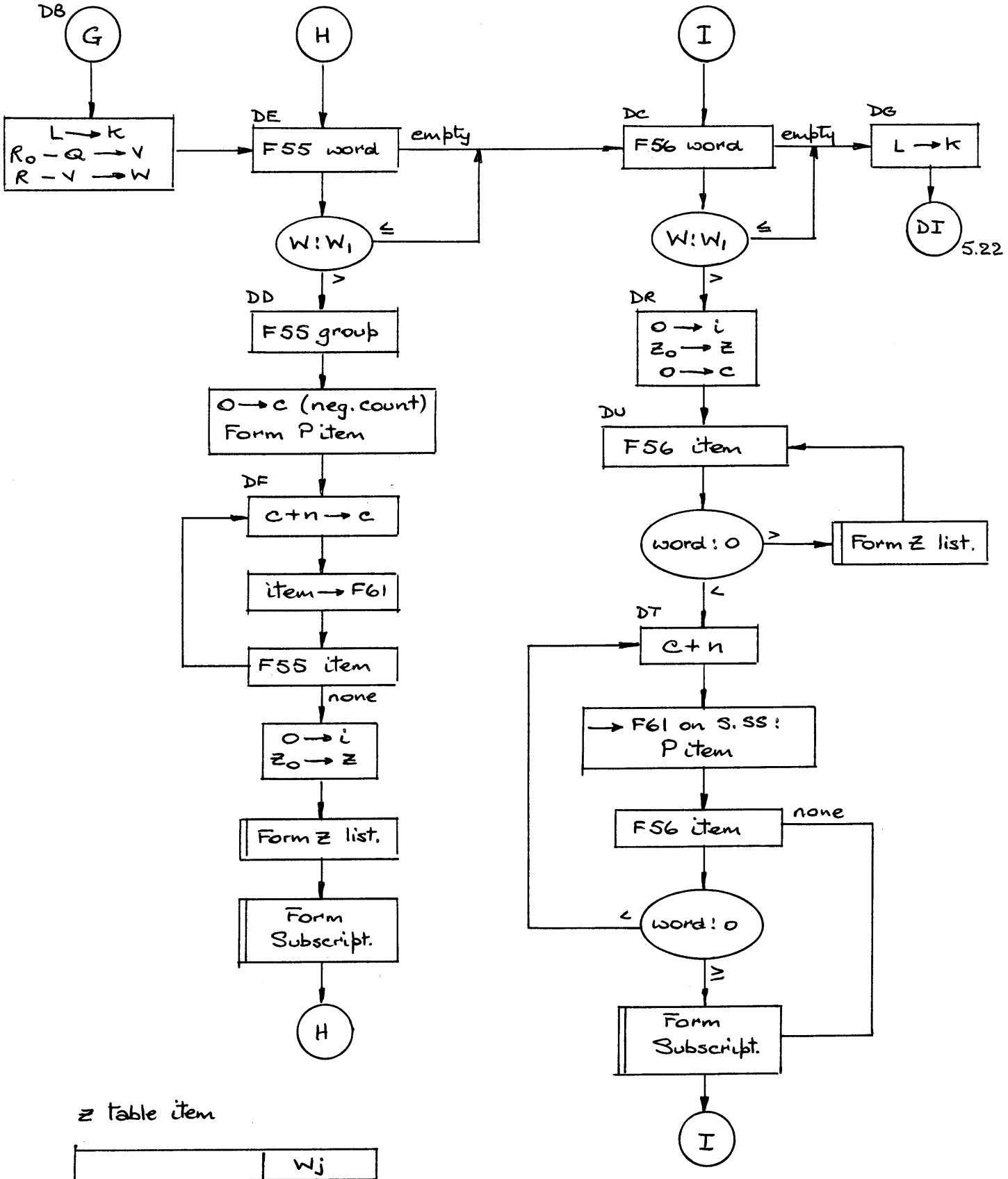
Branch on i


BA
0
C-b:0
≤
1
>
fix (c-b+d)/d → k
BK
k → K12
3 → K10
d K+1 numeric?
No
F
Yes
BKA
d * k → K8
UK → K0
BJ
F
R + # items → R
K+1
K:J
≤
AS
5.16
>
DB
G
5.19
BB
1
b:c
≥
1
<
BD
3
BC
2
Any F54 group?
No
F
Yes
m-1 → m
BN
DPd : Cm
>
m:99
<
=
BL
b var.
DVAR
→ F61 on Cm :
P = *a:cxd for each W
p+1
BJ
F

Branch on i (continued)

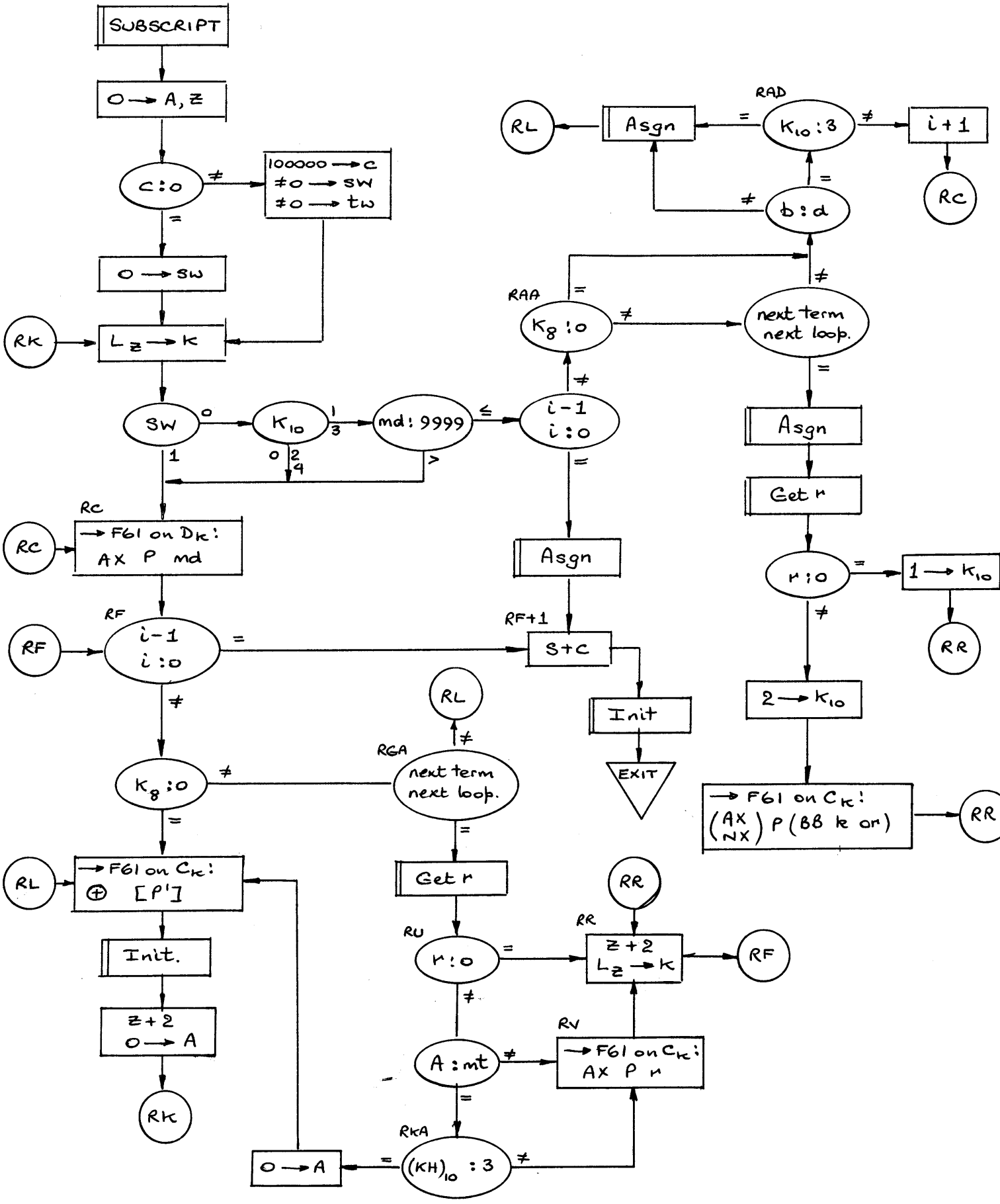


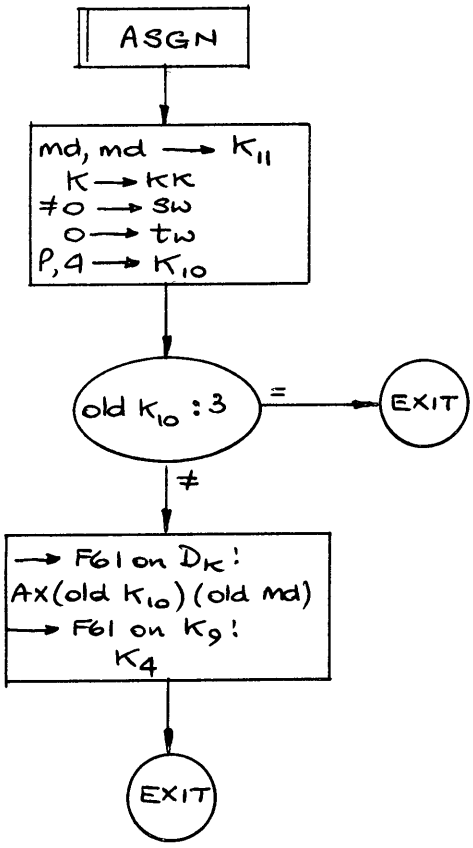



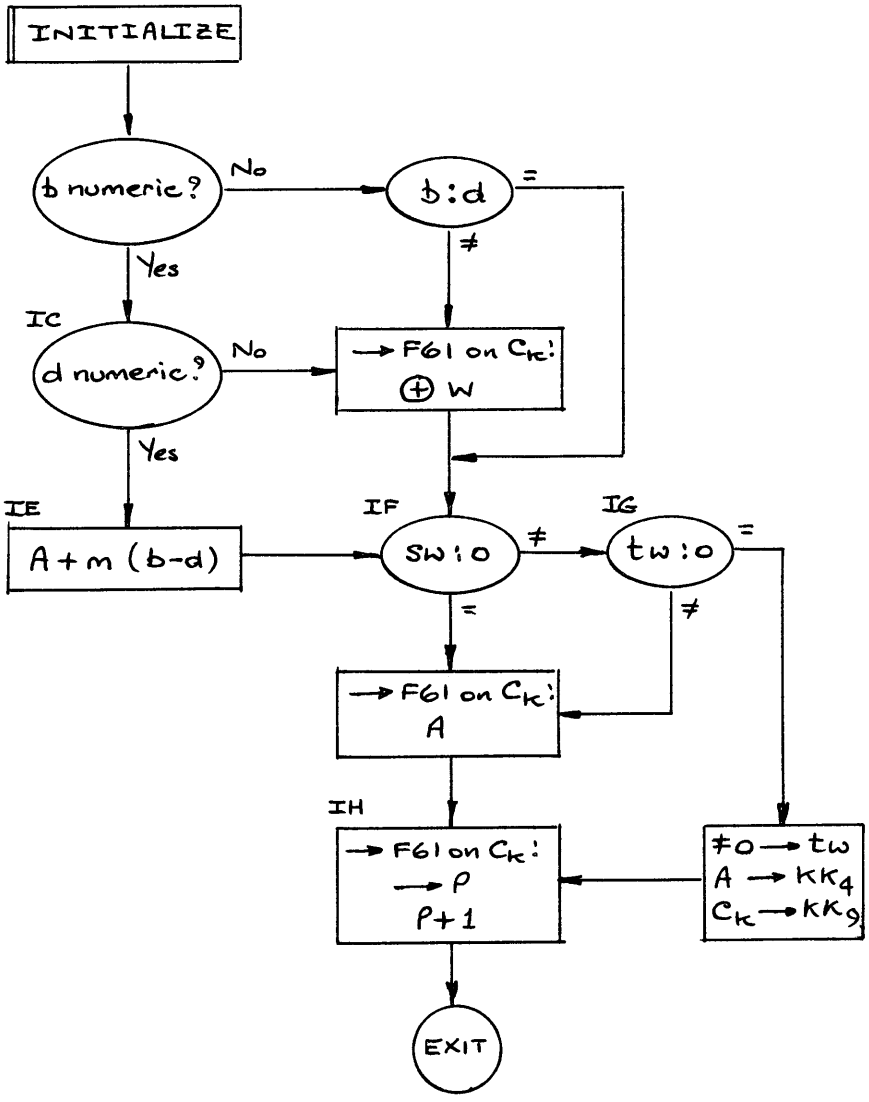



z table item

| | Wj |
|---|---|
| (Wj+V) | |


SUBSCRIPT
0 → A, z
c : 0
≠
100000 → c
≠0 → sw
≠0 → tw
=
0 → sw
RK
L z → K
sw
0
1
K10
1
3
md : 9999
≤
i-1
i : 0
0
2
4
>
1
RC
RC
→ F61 on DK:
AX P md
RF
RF
i-1
i : 0
=
≠
S+C
RF+1
Asgn
Init
EXIT
RL
RGA
next term
next loop.
≠
K8 : 0
≠
=
RL
→ F61 on CK:
⊕ [P']
Init.
z+2
0 → A
RK
Get r
RU
r : 0
=
≠
A : mt
≠
=
RKA
(KH)10 : 3
≠
0 → A
=
RR
RR
z+2
Lz → K
RF
RV
→ F61 on CK:
AX P r
RAD
RL
Asgn
=
K10 : 3
≠
i+1
RC
≠
b : d
=
RAA
=
K8 : 0
≠
next term
next loop.
≠
=
Asgn
Get r
r : 0
=
1 → K10
RR
≠
2 → K10
→ F61 on CK:
(AX / NX) P (BB k or)
RR



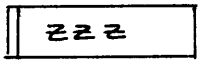




Subroutine




combines up to 3 4-digit W fields into one word

Subroutine



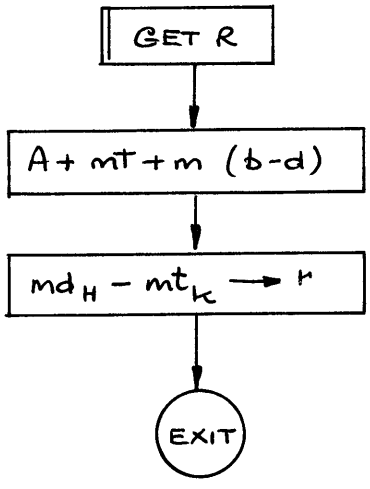

unpacks up to 3 4-digit W fields and forms Table 3

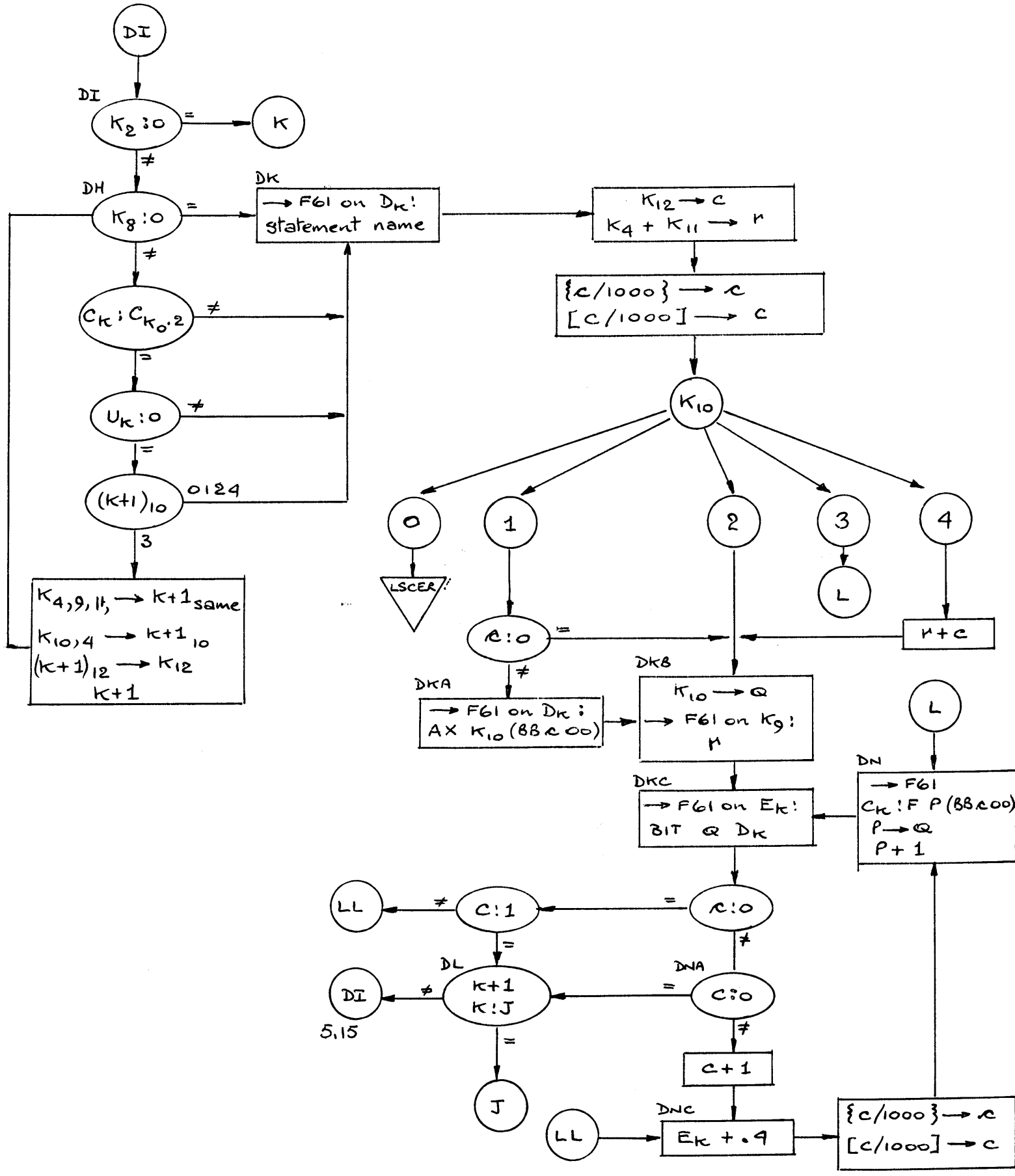
Table 3 Item

| | W |
|---|---|
| F54 item # | W |

⊕ fixed point add operation

[P+1] next A-register name




DI
DI
K2 : 0
=
K
≠
DH
K8 : 0
=
DK
→ F61 on Dk!
statement name
K12 → c
K4 + K11 → r
≠
{c/1000} → c
[c/1000] → c
Ck : Ck0.2
≠
=
Uk : 0
≠
=
(k+1)10
0124
3
K10
0
1
2
3
4
LSCER
L
c : 0
=
r + c
≠
DKA
→ F61 on Dk :
AX K10 (BBc00)
DKB
K10 → Q
→ F61 on K9 :
M
L
DN
→ F61
Ck : F P (BBc00)
P → Q
P + 1
DKC
→ F61 on Ek :
BIT Q Dk
LL
≠
C : 1
=
c : 0
≠
DL
k+1
k : J
DI
≠
=
DNA
C : 0
5.15
=
≠
c + 1
J
DNC
LL
Ek + .9
{c/1000} → c
[c/1000] → c

# PHASE VI

## PURPOSE

Phase VI reduces the space and time required by the object program by marking arithmetic computations which occur more than once as redundant. Phase VII can then generate code to compute just once, quantities which are used more than once. These quantities are called "common subexpressions" hereafter, and are of the form ABC where A is an arithmetic operation or : or , and B and C are operands (or other common subexpressions).

Phase VI also performs what arithmetic it can, preventing generation of unnecessary code and loss of time in the object program.

The marking of subexpressions takes place throughout each block of code (where a block of code is that object code between two successive points to which transfers can be made). Phase VI does no flow analysis of the program, as such an analysis would be costly in compiling time; hence if a subexpression is required in more than one block, it will be recomputed in each.

## PROCESSING

A "block" extends from one block header to the next. Each block of the program is processed in turn. When there are no more blocks, Phase VI is through.

The following items are recognized as block headers:

1. Referenced statement names.
2. Arithmetic function statements.
3. Items indictating presence of SAL code.
4. Statement names generated by Phases IV and V before the first executable statement of the program and at the beginning of each loop.

## The First Pass

The first pass of each block differs from the later passes in these two ways.

1. Items from Files 60 and 61 are input and merged during the pass. Thereafter the block is held in memory.

2. Subexpressions with known or computable values are replaced by constants representing those values. (The constants may then become the operands of other known or computable subexpressions, until all such expressions are eliminated).

   A. +, -, *, /, ** operations are performed on single-precision constant operands. (If a contingency occurs, an error item is filed in F92).

   B. Known values are substituted as shown, where X represents a variable or function, and the constants are single-precision.

      1. 0 for: X*0, 0*X, 0/X, 0**X.

      2. 1 for: X/X, 1**X, X**0.

      3. X for: X+0, 0+X, X*1, 1*X, X/1.

      4. X for: X(0) where X is not a function.

      5. X for: FLOAT (X) (The mode of X made correct).

      6. 1 for: X/0 (an error item is filed).

The mode word of the result is marked 1 (integer), 2 (floating), or 3 (double-precision) as the greatest mode word of the three terms.

Example: The expression

```
X = 3   (N*Y)   + (1-N)*Z + 3.14159265 ** 2
```

reduces to `X = 3Y + 9.86960438` when PARAMETER N = 1 is given

`X = 3Z + 9.86960438` when PARAMETER N = 0 is given

Four instructions of code are generated.

Every Pass

During each pass the program sets up a table containing the relative memory locations of the operation and two operands of each subexpression in the block. The operations recognized are the arithmetic ones and colon and comma. The operands are constants, variables, functions, or (on the second and succeeding passes) marked subexpressions. The operands of commutative operations are arranged in ascending order (thus A*B and B*A are treated as identical).

The table is sorted and then scanned. During the scan the second and third words of each operation are modified as follows.

1. The u digit of the mode word.

   A. Marked 5 if the subexpression is not common. This mark prevents later passes from "processing the subexpression).

   B. Marked 1 throughout first major sequence number in which common subexpression occurs.

Marked 2 thereafter.

2. The f digits of the mode word.

   Marked with the "name" (0000-9999) of the subexpression.

3. The third word.

   Contains the total number of occurrences in every case.

The next pass then begins. It will seek redundancies only among subexpressions containing smaller marked subexpressions. When none exist, the block is output to File 7.

Output Processing

On output, operations marked 5 are remarked 0.

Operations marked 1 or 2 are examined carefully. If one of these contains as an operation and a small subexpression which occurs the same number of times, the smaller operation is remarked 0.

The operations comma, colon (preceding a subscripted and variable preceding zero), are special cases. They are always re-marked 0, since nothing is to be gained in the object code by treating them as subexpressions. For example, in the statement

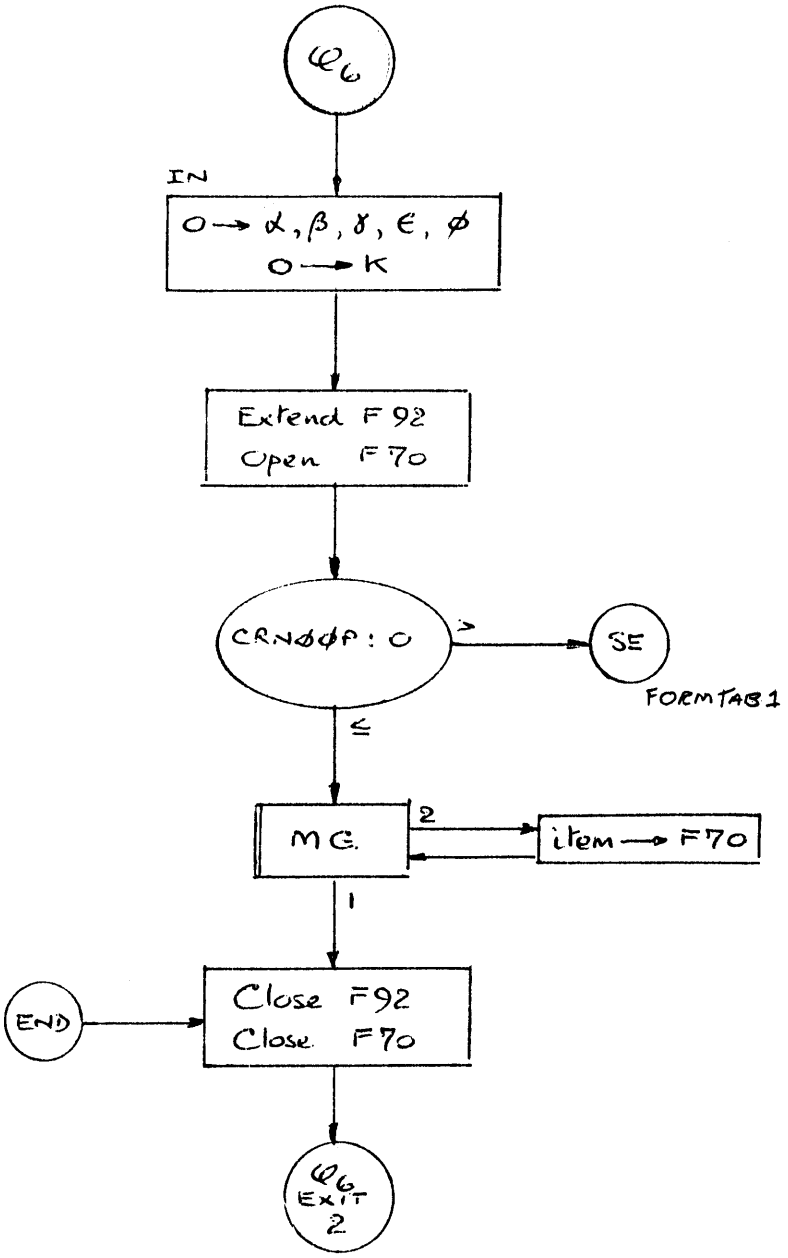
$$X = ABCF\ (A,B+C) * ABCF\ (A\ ,\ B + C) * DEFF\ (A,\ B + C)$$

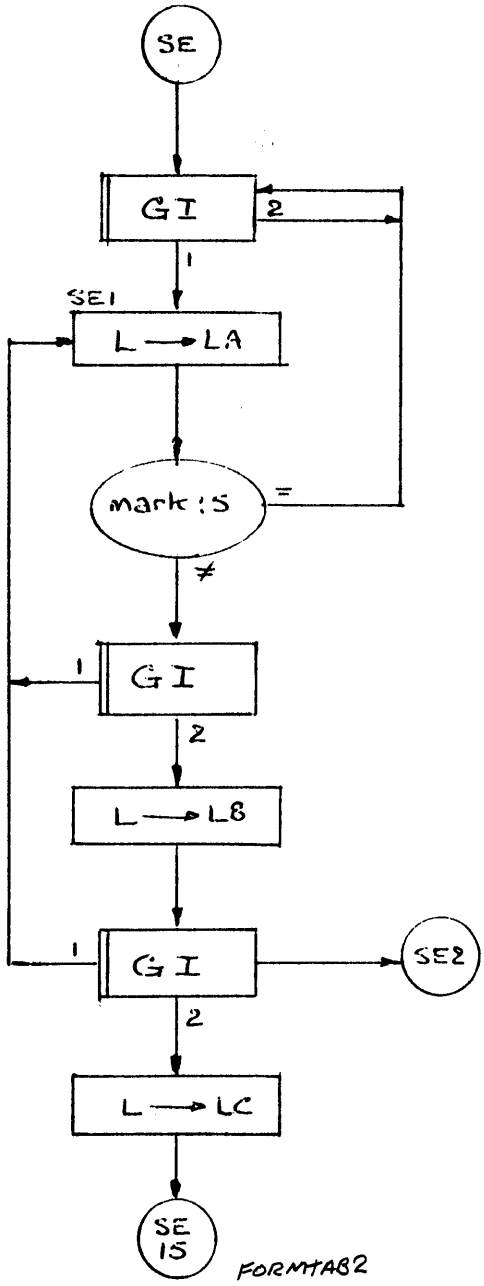
the subexpression B + C is not re-marked even though it occurs as often as the larger A, B + C; because it occurs more often than ABCF ( A, B + C ).

When the last block is output, Phase VI returns to the control program.

FORM TABLE OF COMMON SUB EXPRESSIONS

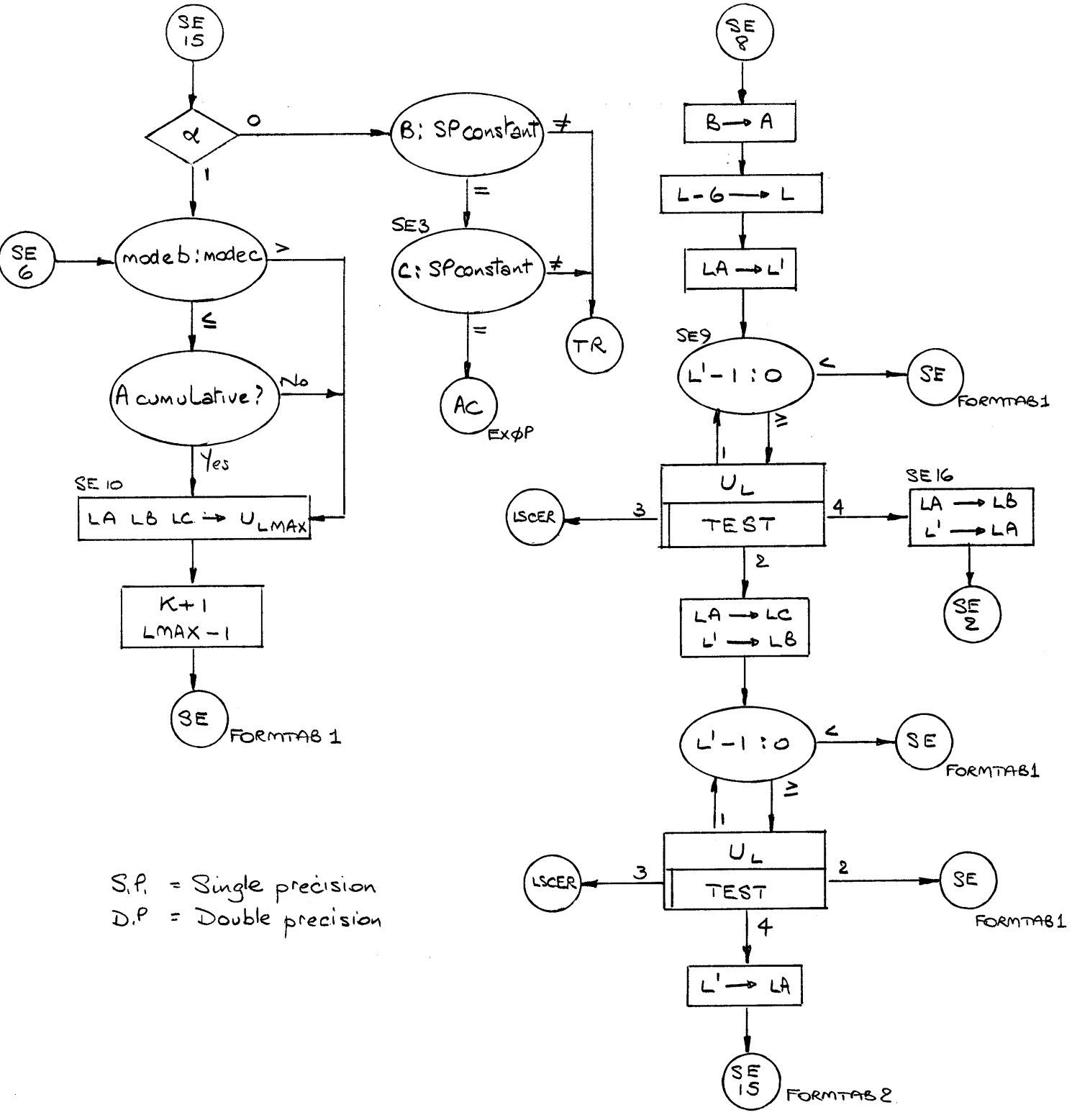
FORMTAB 1



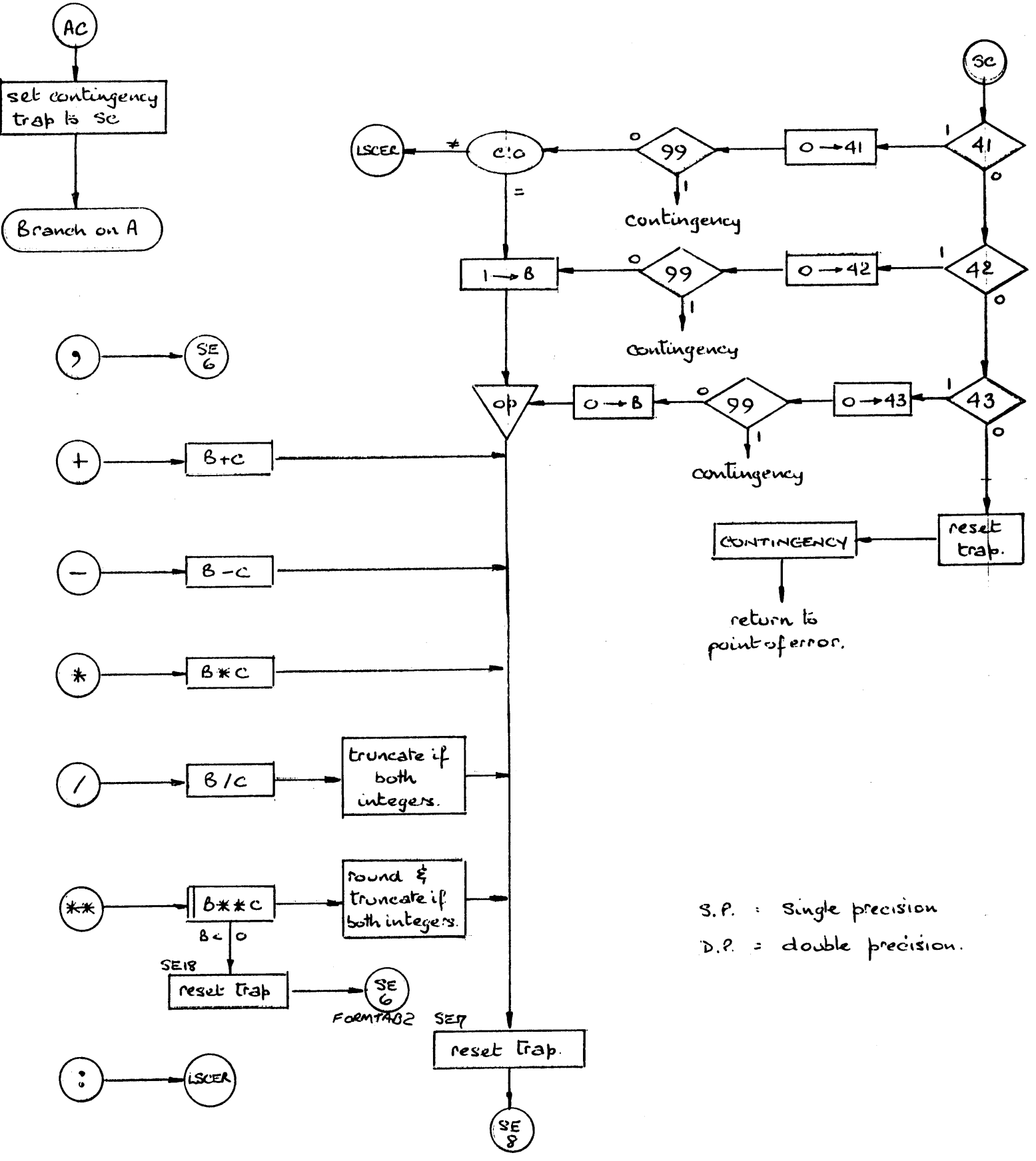
FORM TABLE OF COMMON SUB-EXPRESSIONS (cont.)

FORMTAB2



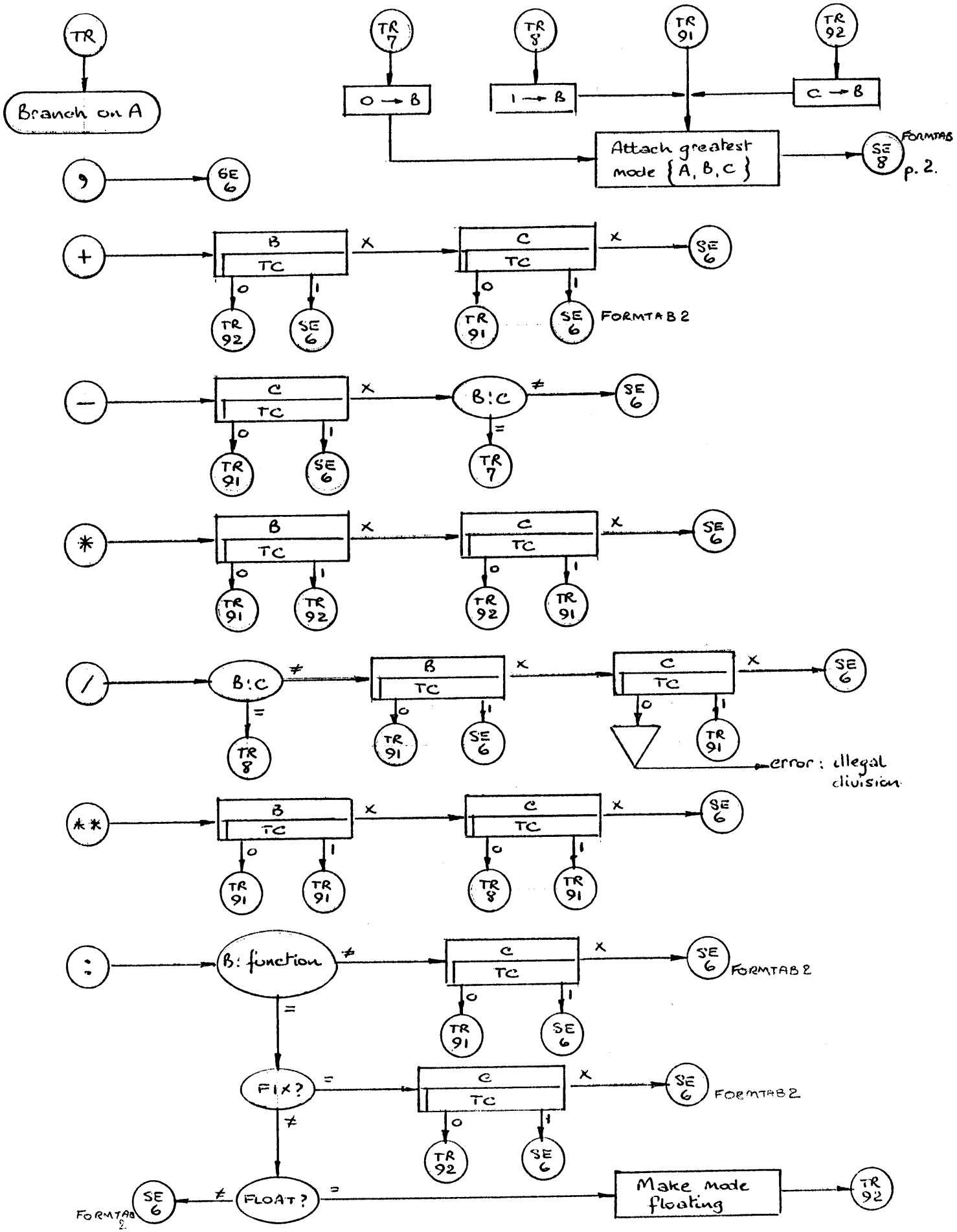

S.P. = Single precision
D.P = Double precision

EXECUTE OPERATIONS WITH DETERMINATE RESULTS EXØP

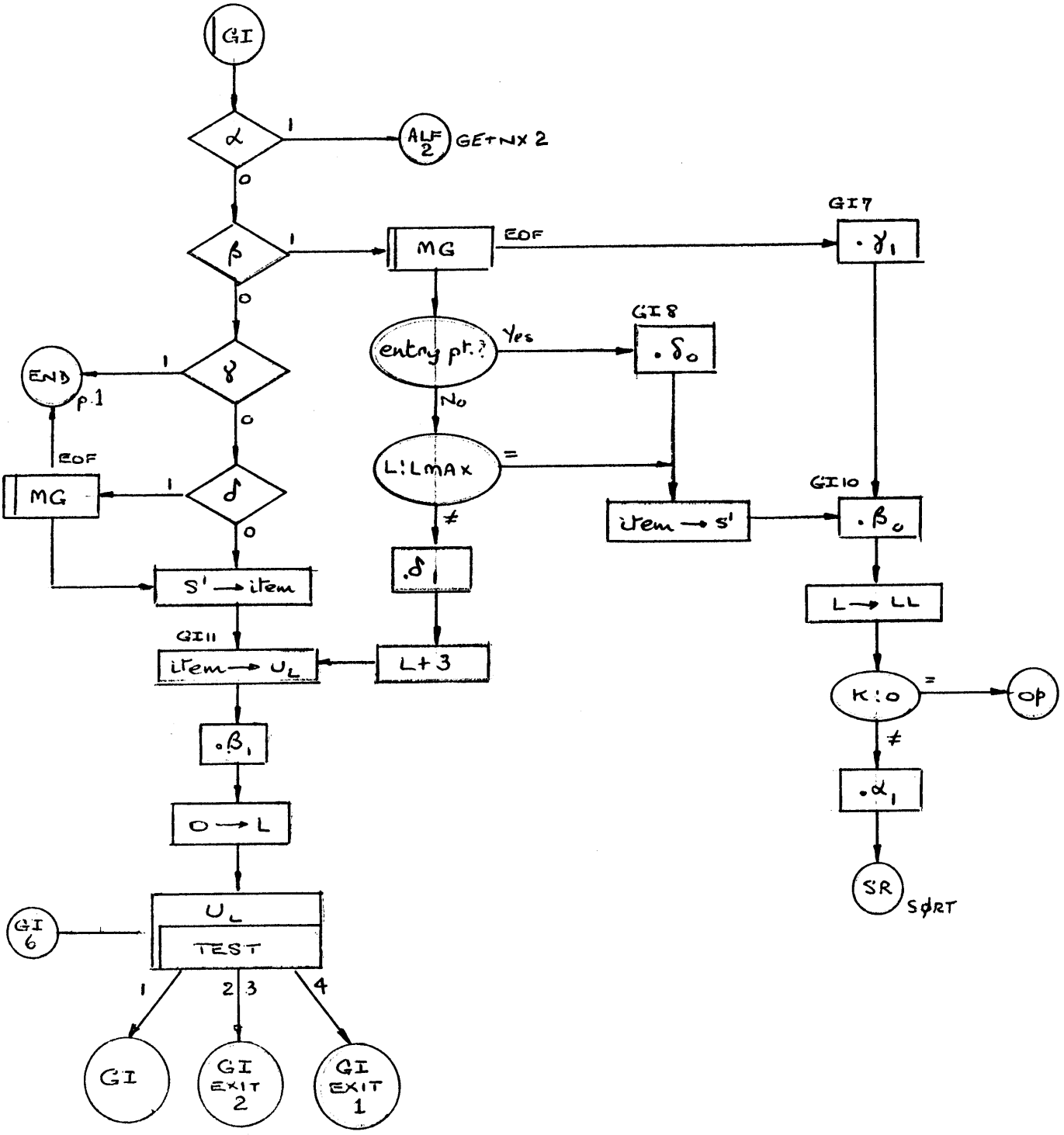



S.P. : Single precision

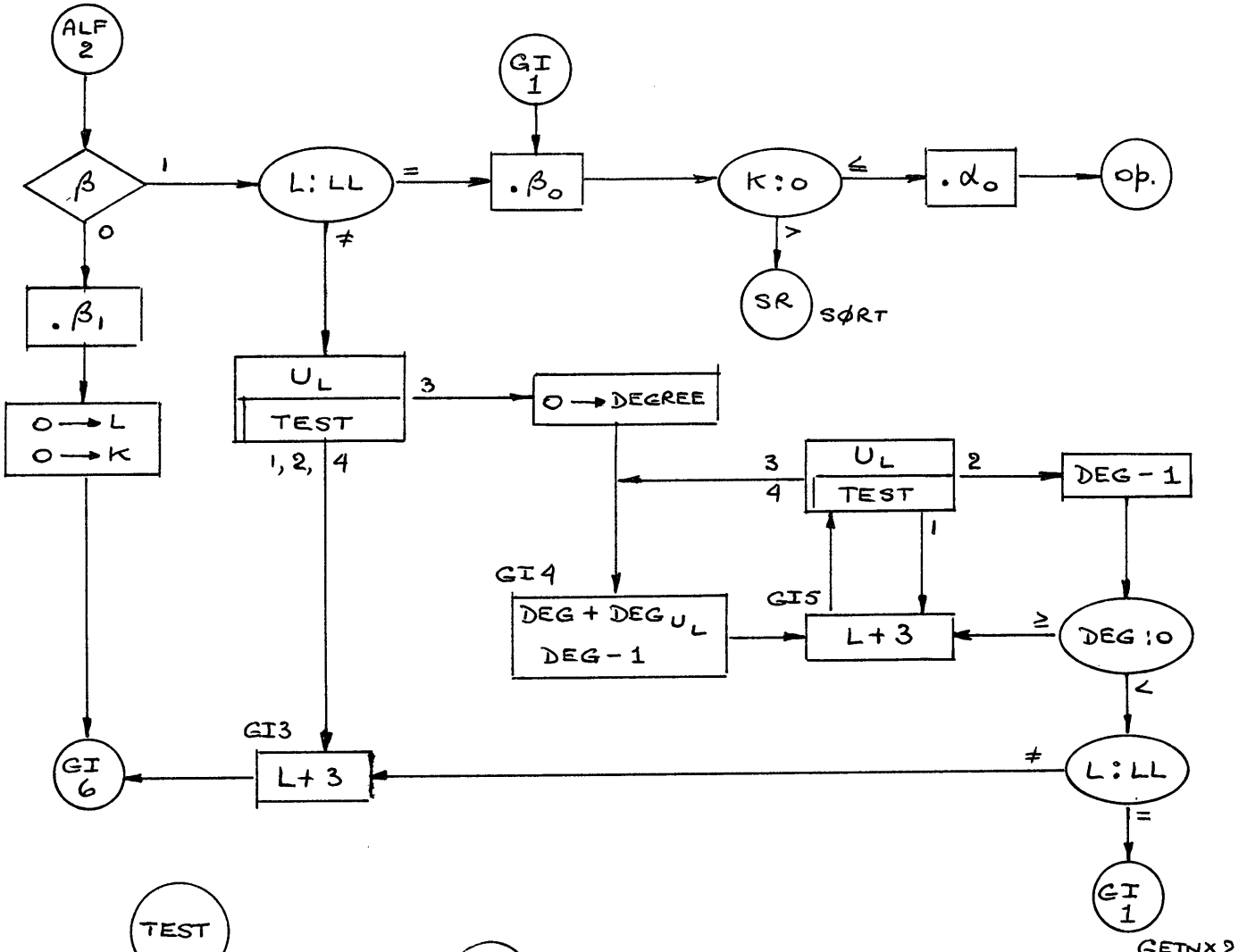
D.P. = double precision.

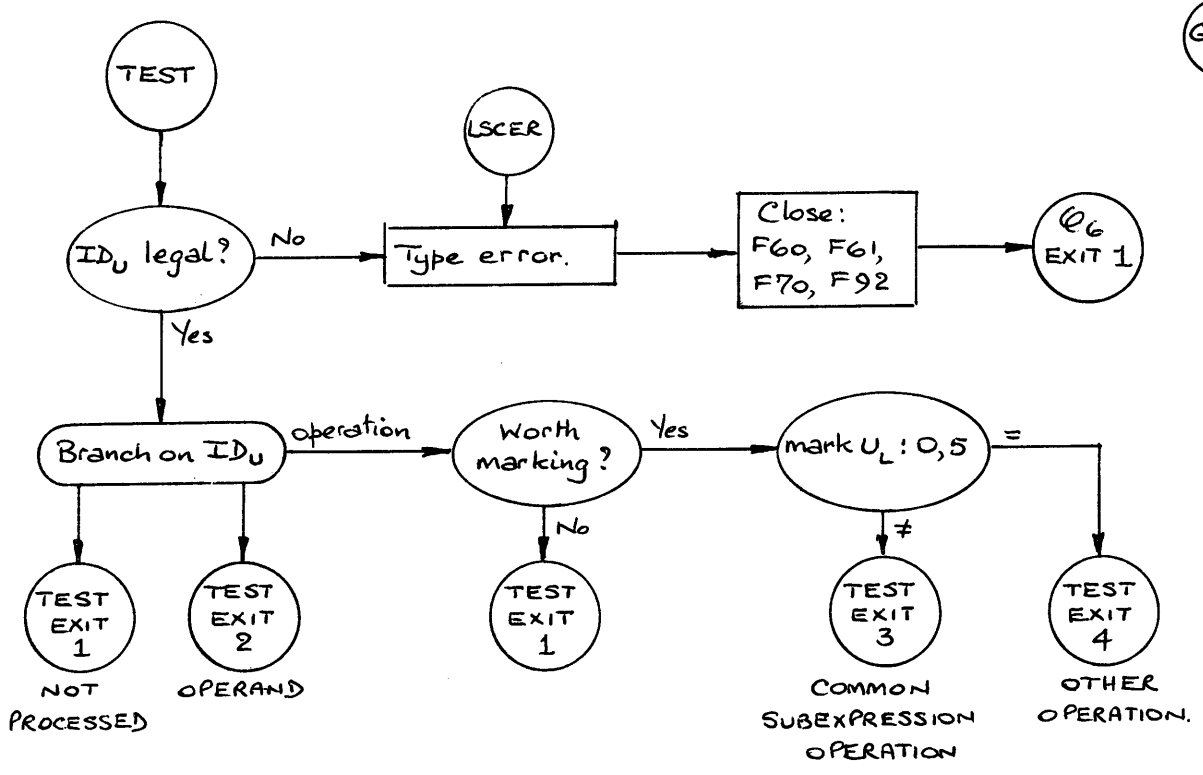

TR
Branch on A
TR 7
0 → B
TR 8
1 → B
TR 91
TR 92
C → B
Attach greatest mode {A, B, C}
SE 8
FORMTAB p. 2.
,
SE 6
+
B
TC
C
TC
X
SE 6
TR 92
SE 6
TR 91
SE 6
FORMTAB 2
−
C
TC
B:C
≠
=
SE 6
TR 91
SE 6
TR 7
*
B
TC
C
TC
X
SE 6
TR 91
TR 92
TR 92
TR 91
/
B:C
≠
=
TR 8
B
TC
C
TC
X
SE 6
TR 91
SE 6
TR 91
error : illegal division
**
B
TC
C
TC
X
SE 6
TR 91
TR 91
TR 8
TR 91
:
B: function
≠
=
C
TC
X
SE 6
FORMTAB 2
TR 91
SE 6
FIX?
=
≠
C
TC
X
SE 6
FORMTAB 2
TR 92
SE 6
FLOAT?
≠
=
SE 6
FORMTAB 2
Make mode floating
TR 92

GET NEXT ITEM GETNX 1

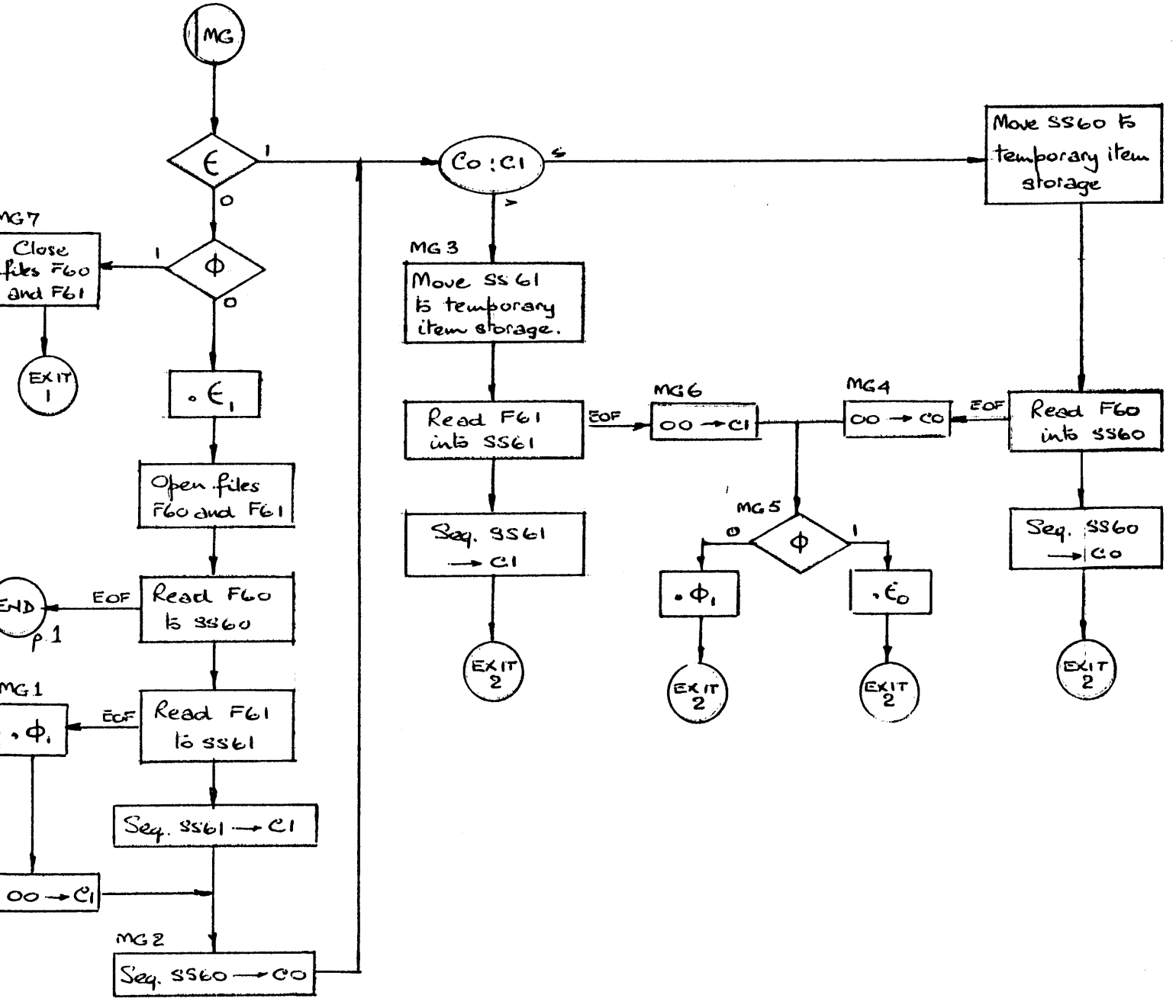


GET NEXT ITEM (continued) GETNX 2



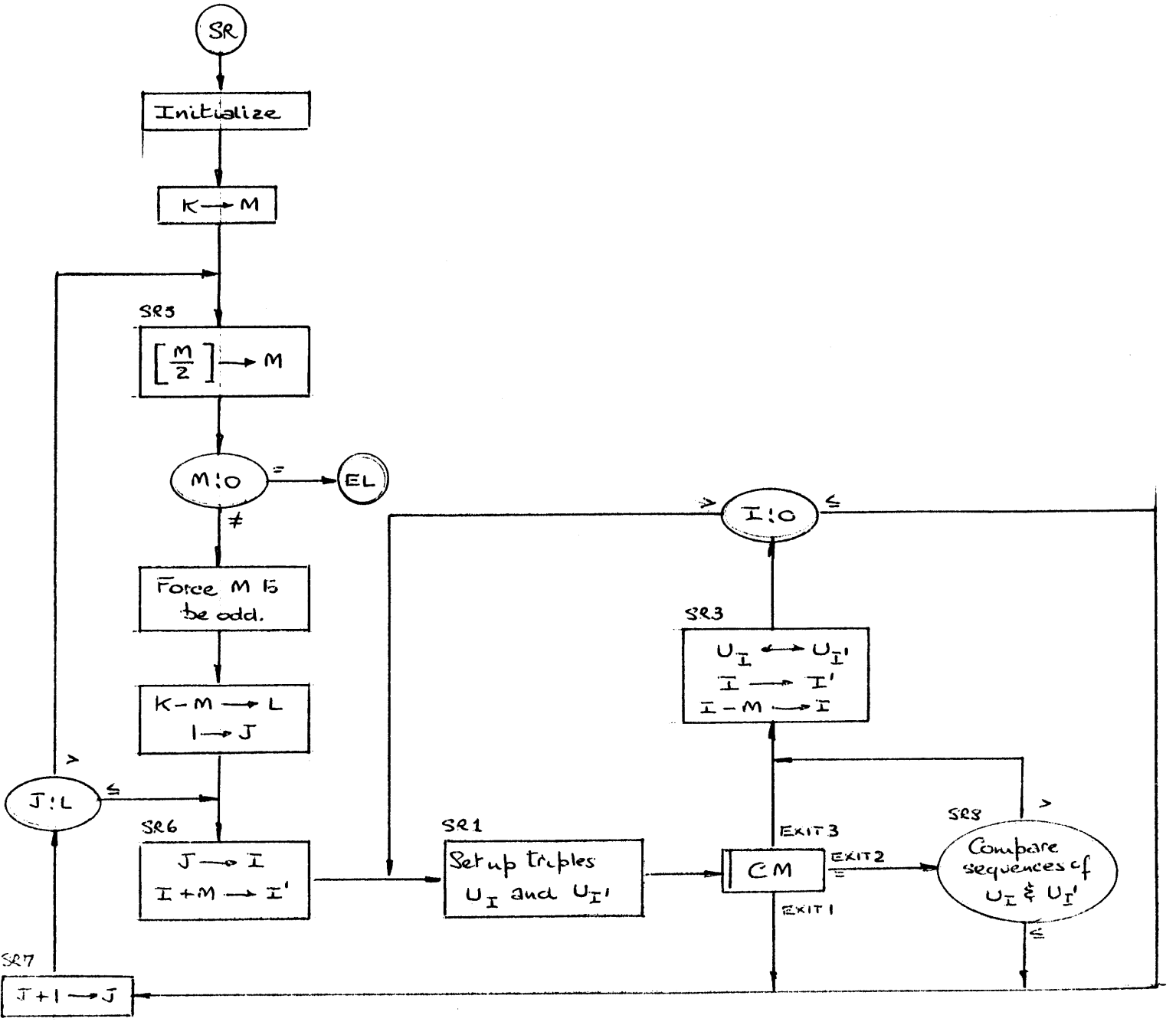



MERGE SUBROUTINE.



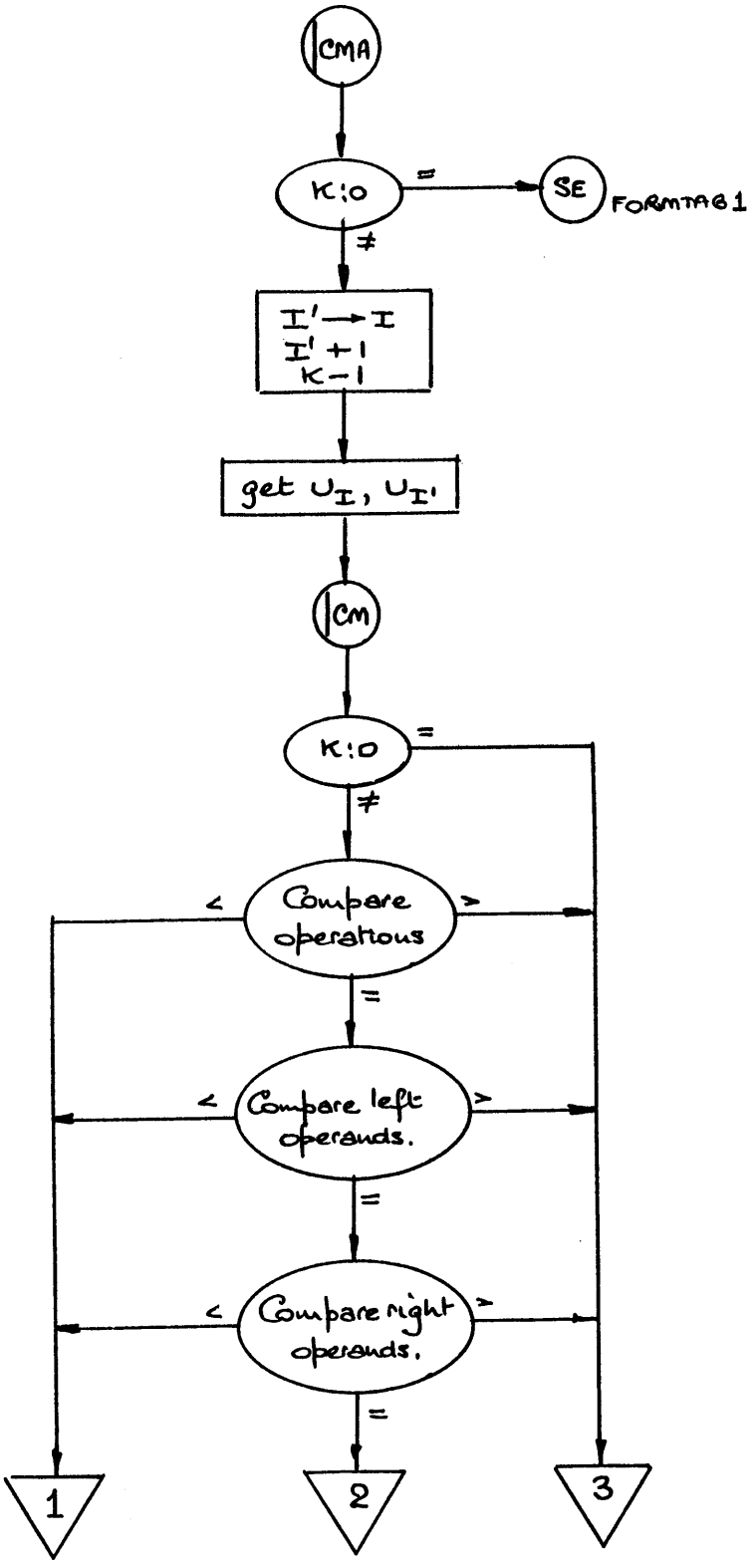
SORT ROUTINE

SORT

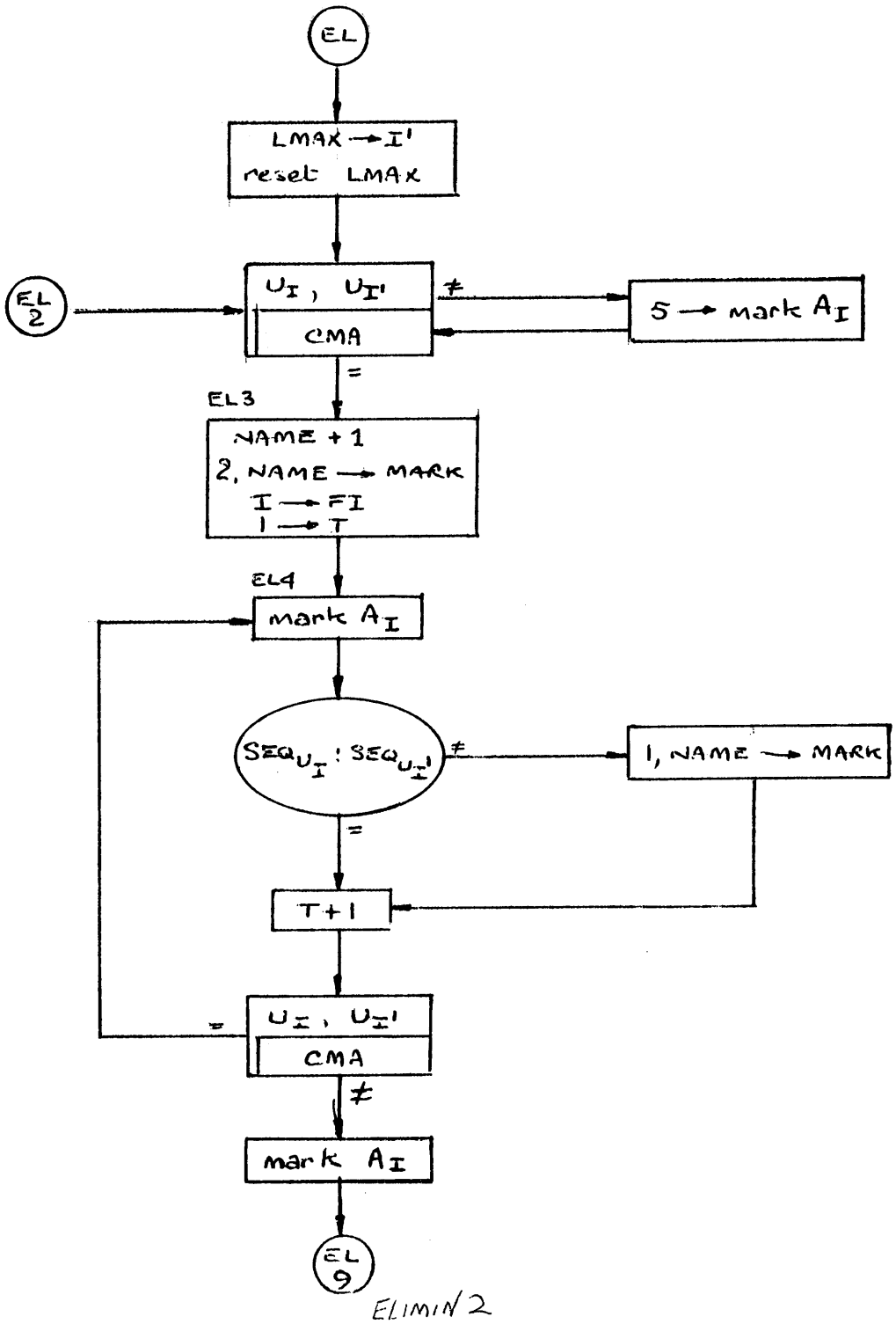


COMPARISON SUBROUTINE

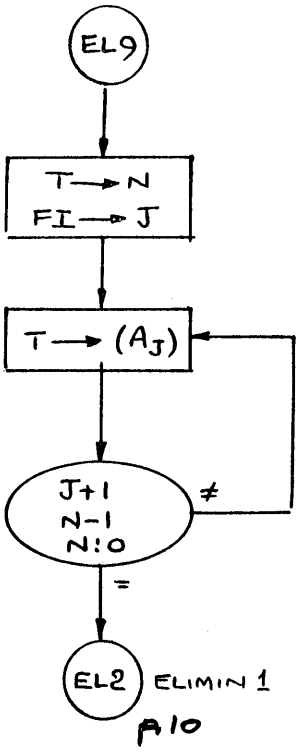


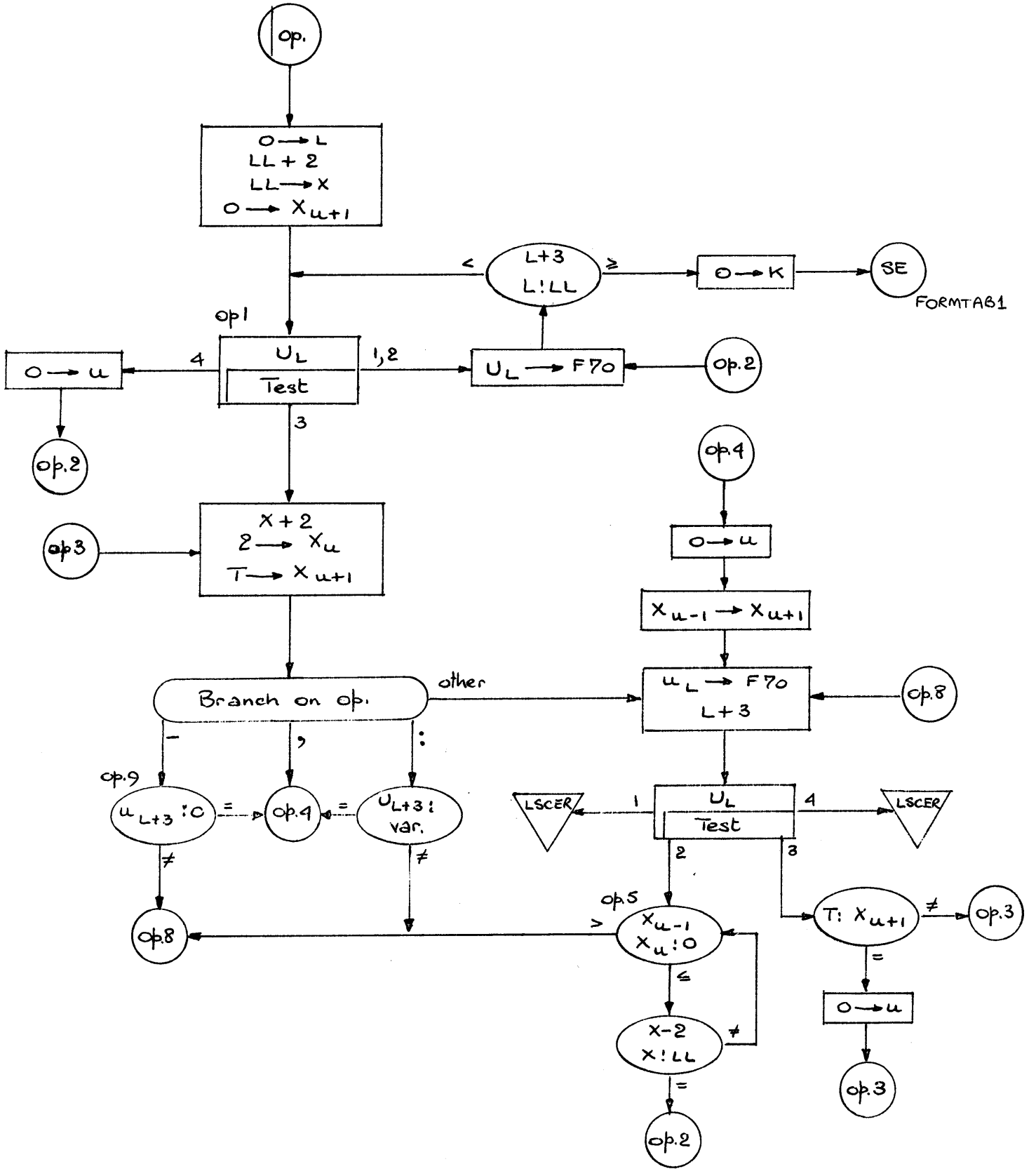


"ELIMINATE" SUB-EXPRESSION ELIMIN 1




ELIMIN 2


EL9
T → N
FI → J
T → (AJ)
J+1
N-1
N:0
≠
=
EL2
ELIMIN 1
P10


op.
0 → L
LL + 2
LL → X
0 → X u+1
L+3
L ! LL
<
≥
0 → K
SE
FORMTAB1
op1
U L
Test
4
1,2
3
0 → u
U L → F70
op.2
op.2
op.4
op3
X + 2
2 → X u
T → X u+1
0 → u
X u-1 → X u+1
Branch on op.
other
U L → F70
L+3
op.8
–
,
:
op.9
u L+3 : 0
=
op.4
=
U L+3 : var.
≠
≠
LSCER
1
U L
Test
4
LSCER
2
3
op.8
op.5
>
X u-1
X u : 0
≤
X-2
X ! LL
≠
=
op.2
T: X u+1
≠
op.3
=
0 → u
op.3

PHASE VII - LSC

General Description

Prior to Phase VII, file 7 has been ordered according to sequence number. The items contain complete mode words and other information necessary to the generation of machine code. Phase VII then processes the items in file 7 in one forward pass, generating "proto" instruction items for file 8. These instructions are complete with operation code, dictionary reference for the symbolic names, and A-Register assignments. These A-Register assignments are given assuming that an infinite number of A-Registers exist. It is necessary then, for Phase VIII to reassign the A-Register designations under the restriction of the number available. Other information is passed on to Phase VIII such as "successor" items and "end of list" items. The successor items give information about transfer points and the end of list items mark the end of Input-Output calling sequences. File 85 is also formed during Phase VII and contains all items not in sequence with file 8. Thus only file 85 need be sorted and merged with the larger file 8, which was produced in proper sequence. Examples of file 85 items include library call items, and instructions necessary for initializing input data addresses for subroutine and function subprogram compilations.

## Phase VII Generator

Phase VII generation is facilitated by an interpretive system which offers a certain convenience when operating on Phase VII items.

A. Linkage to Generator

TR 7GEN — This instruction transfers to the interpretive mode and is followed by SK instructions which give the parameters to be used. The first instruction to be interpreted is the first word following the last SK following the TR.

SK 0 P B — This instruction's following a TR gives a parameter to the interpreter. The parameter is a standard three-word Phase VII item and the first word is in P + (B). If the location of the TR is L and the location of the SK is L + i, the parameter is transferred to in the interpretive mode as Pi (it is the i'th parameter).

T 7GEN — This causes a transfer to the interpretive mode. The next word to be interpreted is the next word. (No parameters can be given and the subroutine level is not changed.)

B. Linkage from Interpreter

TB 0 LOC B — Transfer back to machine language. This interpreter instruction causes the next instruction to be taken from LOC + (B) in machine language.

C. Subroutine Linkage Within Interpreter

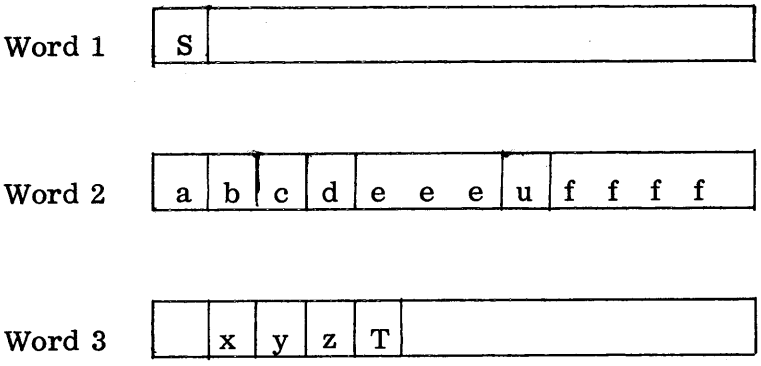
TR SUBR — Transfer and return. This interpreter instruction causes the next interpreter instruction to be taken from (SUBR). The subroutine level is increased by 1. Parameters to the subroutine is given by following SK instructions. Return is to the location following the last SK instruction following the TR.

SK 0 P — This gives a parameter to the interpretive subroutine called by the TR. If the SK is in the i'th word following the TR, the i'th parameter to the subroutine is the P'th parameter given to this program (or subroutine).

Exit from subroutine. Reduce level by one and transfer back to that level return.

D. Data Manipulation

Interpretive data manipulation instructions operate on fields of standard Phase VII items. Phase VII items have the following format:




S is the sign of the quantity referred to and tells whether it is the quantity or the negative of the quantity.

a, b, c, d, e, u, and f are the fields of the standard mode word as described elsewhere.

x, y, z, and T are fields defined by Phase VII and describe data in index registers.

x tells whether the quantity is single or double precision (x=0 implies single, x=1 implies double).

y tells whether the quantity is in a left or right-hand register or a pair (single or double) of registers. y=0 implies left, y=1 implies right.

z tells whether the register currently contains the second or third operand and is a specialized field used only by Phase VII.

T is the class of register, T=0, 1, 2, 3, or 4 for absolute, arithmetic, index (see description of Phase VIII items).

Also word one, two or three can be referred to as an entire field by W1, W2, or W3.

In data manipulation, the field name is followed by the parameter number (as in A1, Y3, or W22). Parameter number 0 implies the result which is an index word being constructed. This parameter has only one word, W3, and so has fields W30, X0, Y0, Z0, and T0 only. Parameters number 1 through 5 refer to parameters as indicated above.

| | | | | |
|---|---|---|---|---|
| M | $fP_1$ | $fP_2$ | | Move $fP_1$ to $fP_2$. |
| S | k | fP | | Store k (a constant) in fP. |
| C | $fP_1$ | $fP_2$ | | Compliment of R $fP_1$ to $fP_2$. Compliment is defined here as: 0 replaces 1 and 1 replaces everything else. |

E. Data Tests

| | | | | |
|---|---|---|---|---|
| TE | $fP_1$ | LOC | $fP_2$ | Transfer to LOC is $fP_1 = fP_2$. Either $fP_1$ or $fP_2$ can be a constant < 10. |
| TG | $fP_1$ | LOC | $fP_2$ | Transfer to LOC if $fP_1 > fP_2$. Either $fP_1$ or $fP_2$ can be a constant < 10. |
| TZ | fP | LOC | | Transfer to LOC if fP = 0. |

F. Transfers

| | | | | |
|---|---|---|---|---|
| T | LOC | | | Transfers to LOC. |
| BIT | 0 | LOC | B | Transfers to LOC + (B) |

G. Generate

| | | | | | |
|---|---|---|---|---|---|
| TF | k | LOC | 0 | T | Transfer to file. Transfer k items to file 8 starting at LOC with tracing digit T. (If the tracing digit of items should be T is 7, tracing digit cannot be 7.) Description of items to be generated appears as follows: |

H. Items to be Generated

Items to be generated have the following form:

OP A M B T

Items will have the tracing digit specified by the TF instructions.

OP is the operation mode of the item

A is the A field of the item. A is either the absolute A field or P0,..., P5 for Parameter 0 through 5.

B is the B field of the item. B is either the absolute B field or P0,..., P5 for Parameter 0 through 5 or F, S, D, or B which specifies that the M field is a literal and is fixed, single precision, double precision or B box respectively.

T describes the M field (unless B is F, S, D, or B in which case T is ignored) and no T implies M is a parameter number (P0,..., P5).

T=R implies M is relative and T=A implies M is absolute.

T=ES or NES implies that an extensible or non-extensible successor is to be generated for M. (See successor item generation.)

M is the M field. M is P0,..., P5 or if T is R or A, M the relative or absolute amount.

I. Example: Divide Parameter 2 by Parameter 3

Calling Sequence

```
TR      7GD
SK      7EXP  7J                          loc of P1
SK      7EXP  7I                          loc of P2
SK      7EXP+7IL    7I                    loc of P3

7GD     TG    B2    7GD1    1   } double precision => GD1
        TG    B3    7GD1    1   }
        TR          7GFIF                 P2 and P3 are integers
        SK          2                     Fetch P2
        TF    4     7GD2                  Generate 7GD2
        T           7GD8
```

```
7GD2    ME      P0      (l. 0)      S                   Make P0 D. P.
        DSE     P0      P3                              Divide
        CX      P0      61 0        A                   Truncate
        C       P0      61 0        A

7GD1    TG      B1      7GD3        B2                  Maximum of B1, B2, B3 to B1
        M       B2      B1

7GD3    TG      B1      7GD4        B3
        M       B3      B1

7GD4    TG      3       7GD5        B1
        S       1       X0                              Make result D. P.

7GD5    TR              TGFIF                           Fetch P2
        SK      0       2
        TE      B1      7GD6        3
        TF      1       (DR         P0      P3)         Generate S. P. Divide
        T               7GD8

7GD6    TE      B3      7GD7        3
        TF      1       (DSE        P0      P3)         P2, D. P. , P3 S. P.
        T               7GD8

7GD7    TF      1       (DD         P0      P3)         P2 and P3 D. P.

7GD8    M       W30     W31                             Move result to P1 index word
        TE      S2      7GD9        S3
        S       1       S1                              1---> S1 if S2 = S3
        TB              1  7Z1                          Exit

7GD9    S       0       S1                              0--> S1 if S2 = S3
        TB              1  7Z1                          Exit
```

This generator program uses subroutine 7GFIF which will fetch its 1st parameter and make the result double precision (1—> X0) if it is double precision and expand the result to double precision if X0=1 and the parameter is single precision.

PHASE VII - Successor Item Generation

1. Within a macro, the tracing mode designation on an instruction to be generated controls the successor item generation. If the tracing mode designation is:

   a. ES, then an extensible successor item for the M address parameter is generated and sent to file 8:

| | | |
|---|---|---|
| Word 1 | S(exit) - 1 | 99999 00 |
| Word 2 | 02000 | 00 s (cont) |
| Word 3 | 00000 | 00 000 00 |
| Word 4 | | |

   where S(exit) is the first five digits of the first word of the parameter item and S(cont) is the first five digits of the third word of the parameter item.

   b. NES, then a non-extensible successor item for the M address parameter is generated and sent to file 8:

| | | |
|---|---|---|
| Word 1 | S(exit) | 99999 00 |
| Word 2 | 02000 | 00 S(cont) |
| Word 3 | 00000 | 00000 01 |
| Word 4 | | |

## Generation of Arithmetic Expressions

The portion of phase 7 that generates code for arithmatic expressions is divided into three sections, 7A, 7B and 7C. 7A converts the Lukasiewitz notation into a "tree notation," 7B determines the order of computation, and 7C generates the code.

### I. 7A

Items are read in one at a time and sequence numbers are generated to sort into tree notation. The sequence numbers are divided into two parts: the high order part is a level number and the low order part is the ordinal number of the item within the expression. The level number of an item following an operation is the level number of the preceding item plus one; and, if the preceding number is an operand, the level number is equal to the highest unmatched level number (and causes that level number to be matched).

If a common-subexpression is encountered, which is not a first occurence, it is skipped over in 7A.

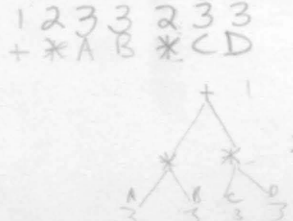
When open subroutines are encountered, they are converted to special operations; e.g., the Lukasiewitz string

: MAX , X Y (from the source MAX (X, Y) is converted to

OP86 X Y where operation code 86 is the phase 7 code for MAX.

When closed subroutines are encountered, the ordinal parameter number is inserted into the corresponding parameter item and the number of parameters is inserted into the function's name item.

After all items have been read, they are sorted on the new sequence numbers. Following the sort, both operands of an operation appear together and must be related to the operand. The operation of two operands is always the closest preceding operation. At this time, if the operation is "-", it is changed to "+" and the second operand is made minus.



II. 7B

This section determines order of computation (except as regards where a common subexpression is computed within an expression). This is done by assigning "Q" numbers to all items as follows:

The "Q" of all simple operands (operands which are not subexpressions) is zero, and if operation j has operands i and i+1, then $Q_j = \max (Q_i, Q_{i+1} + S(Q_i, Q_{i+1}))$ where $S(Q_i, Q_{i+1})$ is the Kronecker delta. The "Q" of the lowest level operation is usually the number of A registers required to compute the expression, if the entire expression is single precision and has no first occurrence of common subexpressions.

The computational order is now determined by starting at the lowest level operation. First, compute the operand with the higher "Q" and repeat the step for the lowest level operation in this subexpression, and repeat this step until an operation is found where both of its operands have a "Q" of zero. Now this operand may be computed. After this is computed, the "Q" of this subexpression may be replaced by zero and the process repeated for the other operand of the operation involving this operand.

The above description would indicate that this is a two-pass process. Actually, this is not the case, but it is easier to understand the process in this way. In the implementation of the above algorithm, an associative list is built up at the same time the "Q's" are generated, which give the order of computation. If the starting points for computing two subexpressions joined by an operation to form a new subexpression are known, then the starting point for computing the new subexpression is the same as the starting point for computing the operand with the higher "Q" (or the right-hand operand if they have equal Q's).

Symbolically, the actual algorithm employed is as follows:

If A is a subexpression, let

S(A) = the first operation in A to be computed.

L(A) = the last operation in A to be computed.

If X and Y are two operations to be computed, let

Y = F(X) mean that Y is to be computed immediately after X.

If operation "A" joins the operands B and C, then there are two cases:

Case 1 $Q(B) \leq Q(C)$ in which case

$S(C) \longrightarrow S(A)$

$S(B) \longrightarrow F(L(C))$

$A \longrightarrow F(L(B))$

$A \longrightarrow L(A)$

Case 2 $Q(B) > Q(C)$ in which case B and C are interchanged in the above.

Finally, $\max(Q(B), Q(C)) + S(Z(B), Q(C)) \longrightarrow Q(A)$

Consequently, the starting and ending points for two subexpressions are known, the starting and ending points for the operation joining them are determined, and as each operation (except the last or lowest level one) has a follower (F), the computation order is determined because F is determined for each operation.

To start the process is actually a separate case. If $\max(Q(B), Q(C)) = 0$, then

$A \longrightarrow S(A)$

$A \longrightarrow L(A)$

When a common subexpression is encountered, its Q is set to zero.

III. 7C

Here the actual generation is done. If the lowest level operation is A and Z is the next operation to be generated, then $S(A) \longrightarrow Z$ and thereafter $F(Z) \longrightarrow Z$ unless one of the operands of an operation is a common subexpression in which case $S(B) \longrightarrow Z$ if B is the subexpression. If both operands are common subexpressions, S of the right-hand operand goes to Z first.

After a common subexpression is generated, it is left in an A register (which may require an F or FF to be generated) and the mode information, including the index number is entered in a table. When a subexpression is already computed, instead of S of that subexpression replacing Z in the above, the mode information is placed in the item from the table and Z modified in the usual way ($F(Z) \longrightarrow Z$.

Once a particular operation is to be generated, that operation is placed in an index register, and a jump to the proper interpretive routine is made.

Sometimes the negative of an expression is computed instead of the expression desired; i.e., consider the expression A-B*C, where A, B, and C are single precision variables, the code generated will be:

```
F  α  B
M  α  C
N  α  A
```

leaving the negative of the expression desired in A register $\alpha$. Therefore, all items in 7C are signed and the sign of an operation (as well as the code generated) is a function of the operation and the signs of its two operands. Although no extra instruction can ever be generated in computing an expression in this way, the sign of the resulting expression may be negative.

If a negative expression results in generating an arithmetic statement, a SN or SSN will be used to store the result instead of a S or SS. However, if the expression results in generating an IF statement, an extra SN or SSN may be generated. Nevertheless, ending up with a negative expression means that at least one instruction was saved in generating the expression.

Phase VII Sample Codes

Generation of object code for arithmetic combinations of quantities with different modes.

I. Addition and Subtraction: a + b

A. If the signs of a and b are the same, v = A and the operation gets the same sign.

B. If only one of a and b are positive, V = N, let a be the positive one, and b the negative one. The operation gets the opposite sign of the quantity in the M address of the last instruction.

| | a in M store<br>b in M store | a in M store<br>b in A store | a in A store<br>b in M store | a in A store<br>b in A store |
|---|---|---|---|---|
| a single<br>b single<br>(op normal) | F δ a<br>V δ b | V B a | V a b | V a b |
| a single<br>b single<br><br><br>(op double) | F δ a<br>F β b<br>SM 0 δ+1<br>SM 0 β+1<br>VV δ β | SM 0 b+1<br>F δ a<br>SM δ+1<br><br>VV δ b | SM 0 a+1<br>F β b<br>SM 0 β+1<br><br>VV a β | SM 0 a+1<br>SM 0 b+1<br>VV a b |
| a single<br>b double | F δ a<br>SM 0 d+1<br>VV δ b | F δ a<br>SM 0 d+1<br>VV δ b | SM 0 a+1<br>VV a b | SM 0 a+1<br>VV a b |
| a single<br>b single | F β b<br>SM 0 β+1<br>VV b a | SM 0 b+1<br>VV b a | F β b<br>SM 0 β+1<br>VV a β | SM 0 b+1<br>VV a b |
| a double<br>b double | FF δ a<br>VV δ b | VV b a | VV a b | VV a b |

II. Multiplication: a x b

A. If signs of a and b are the same, the operation is tagged positive.

B. If the signs of a and b differ, the operation is tagged negative.

| | a in M store<br>b in M store | a in M store<br>b in A store | a in A store<br>b in M store | a in A store<br>a in A store |
|---|---|---|---|---|
| a single<br>b single<br>(op normal) | F δ a<br>M δ b | M b a | M a b | M a b |
| a single<br>b single<br>(op double) | F δ a<br>ME δ b | ME b a | ME a b | ME a b |
| a single<br>b double | F δ a<br>S δ δ+1<br>PR δ+1 11<br>MM δ b | F δ a<br>S δ δ+1<br>PR δ+1 11<br>MM b | S δ δ+1<br>PR δ+1 11<br>MM δ b | S δ δ+1<br>PR δ+1 11<br>MM δ b |
| a double<br>b single | F β b<br>S β β+1<br>PR β+1 11 | S β β+1<br>PR β+1 11<br>MM β a | F β b<br>S β β+1<br>PR β+1 11 | S β β+1<br>PR β+1 11<br>MM a β |
| a double<br>b double | FF δ a<br>MM δ b | MM b a | MM a b | MM a b |

III. Division: a/b

A. If signs of a and b are the same, the operation is tagged positive.

B. If the signs of a and b differ, the operation is tagged negative.

| | a in M store<br>b in M store | a in M store<br>b in A store | a in A store<br>b in M store | a in A store<br>a in A store |
|---|---|---|---|---|
| a single<br>b single<br>(op normal) | F δ a<br>DR δ b | F δ a<br>DR δ b | DR a b | DR a b |
| a single<br>b single<br>(op double) | F δ a<br>SM 0 δ<br>DSE δ b | F δ a<br>SM 0 δ+1<br>DSE δ b | SM 0 a+1<br>DSE a b | SM 0 a+1<br>DSE a b |
| a single<br>b double | F δ a<br>SM 0 δ+1<br>DD δ b | F δ a<br>SM 0 δ+1<br>DD δ b | SM 0 a+1<br>DD a b | SM 0 a+1<br>DD a b |
| a double<br>b single | FF δ a<br>DSE δ b | FF δ a<br>DSE δ b | DSE a b | DSE a b |
| a double<br>b double | FF δ a<br>DD δ b | FF δ a<br>DD δ b | DD a b | DD a b |

For integer divide use corresponding macro for "a single, b single, op double" followed by:

```
CX   δ   61
C    δ   61
```

## LIBRARY ROUTINES

1. Phase III detects occurrences of a reference to a sub-program (including library routines) and transmits them to Phase IV. The list of subroutines generated by Phase II allows the distinction of open subroutines.

2. Phase IV produces LIB items for F9.4 as requested.

3. Phase VII marks a table of library routines over which it has control.

   a. Introduced routines

   b. Convertable routines

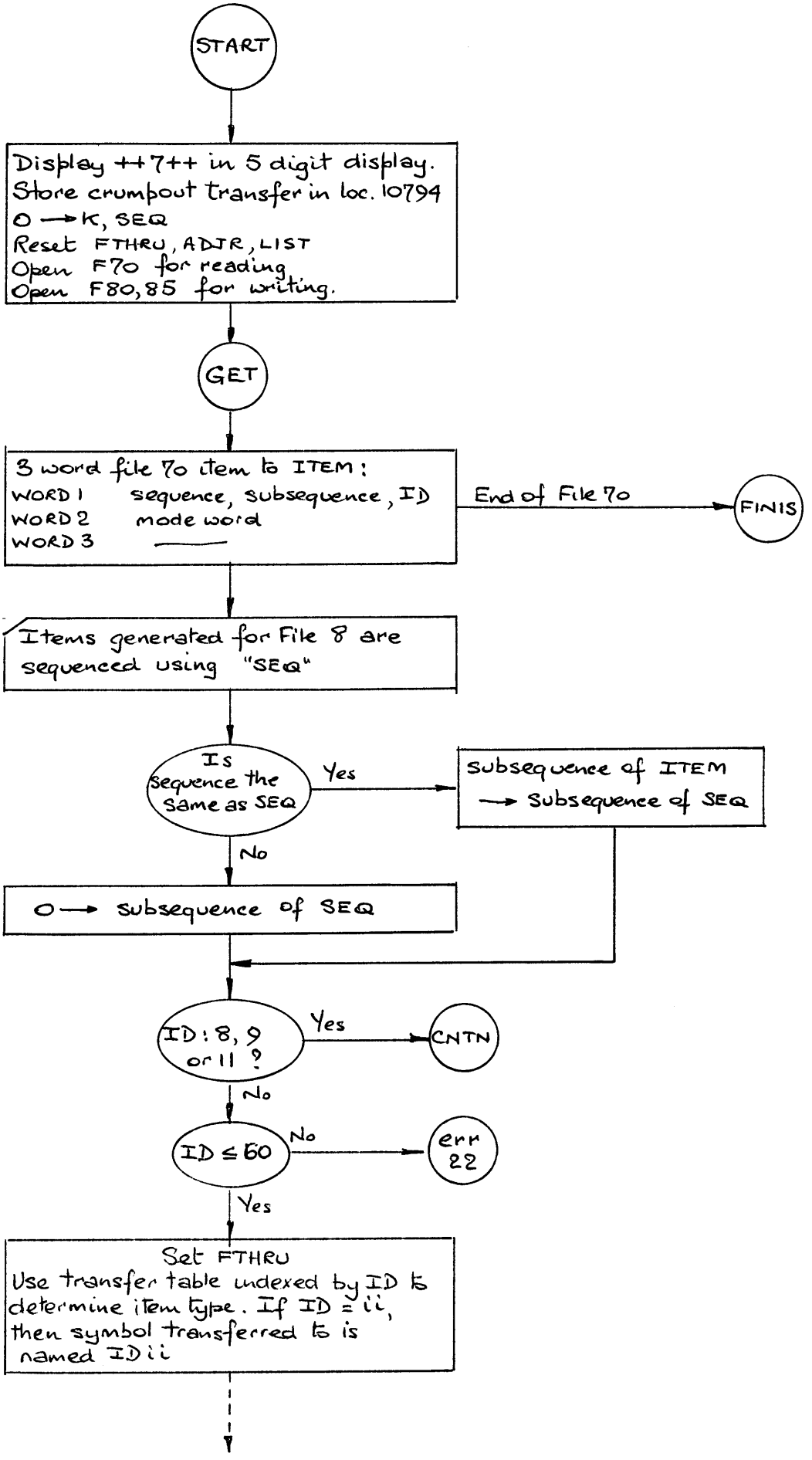
   At the end of file 7, Phase VII sends LIB items for each marked entry in the table to F9.4. They will be of the same form as those produced by Phase IV.

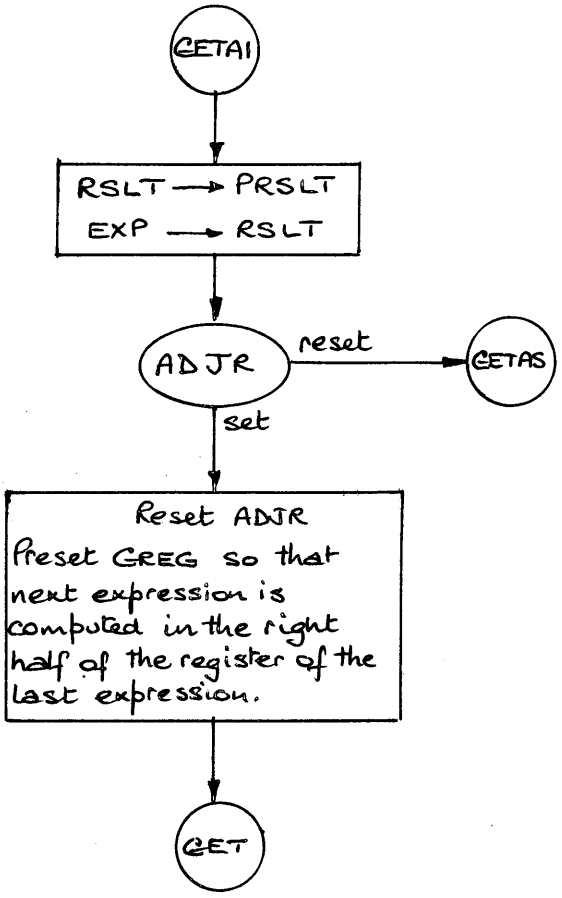
   All references to subroutines will be given an address mode of 06.

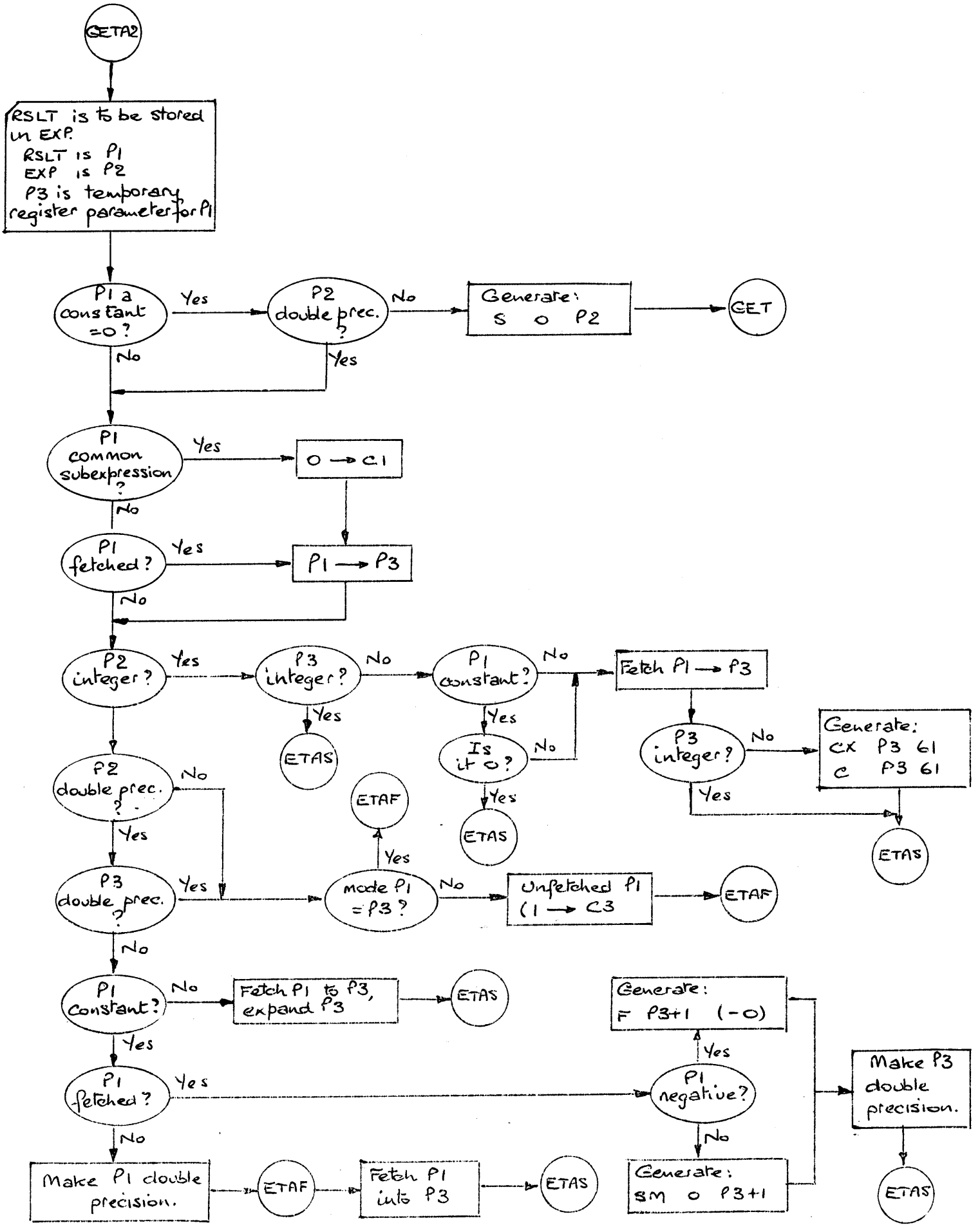
4. Phase IX will produce LIB lines in SAL as specified by F9.4. The address mode of 06 will force editing with a marker.

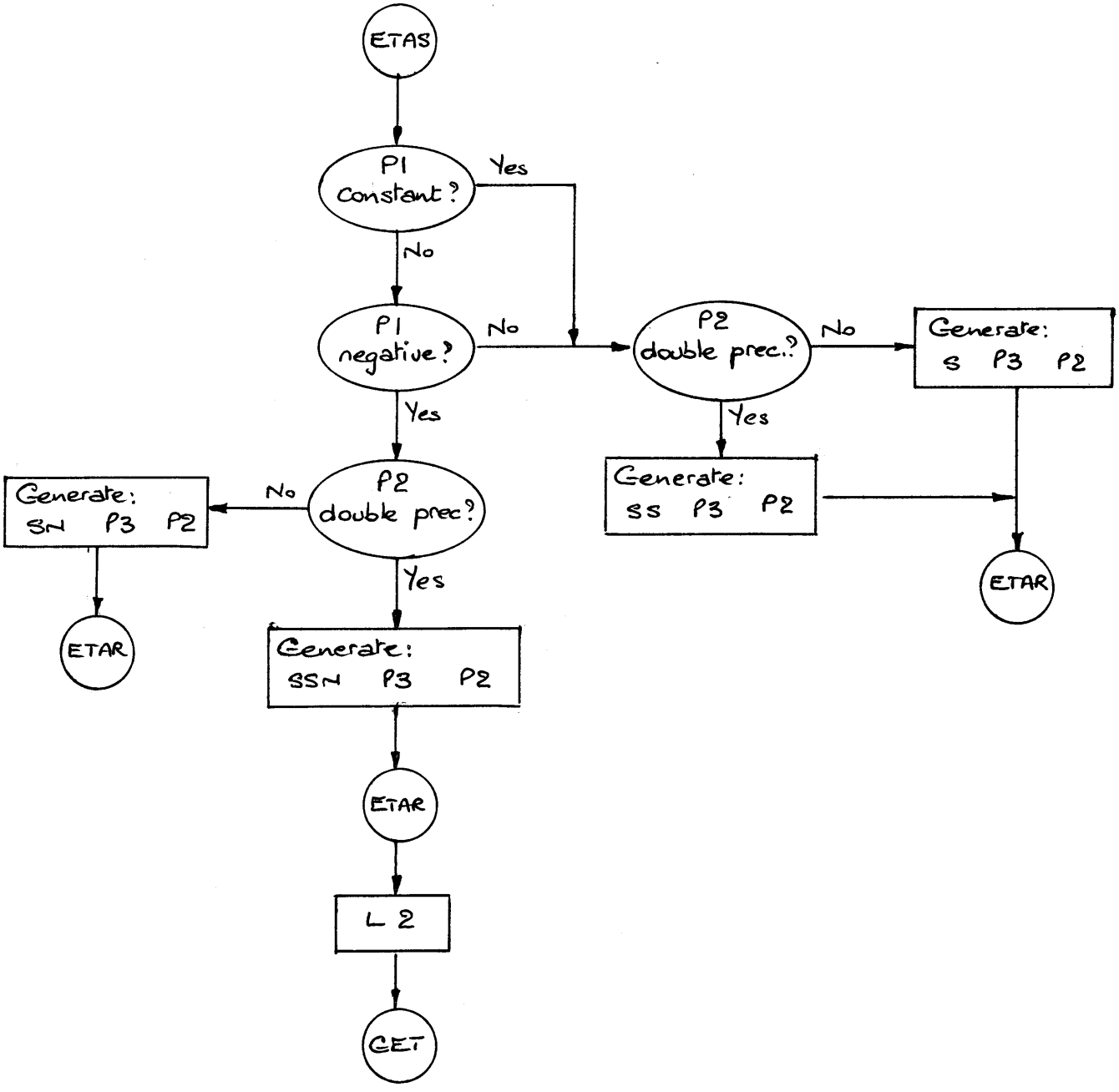
# LARC SCIENTIFIC COMPILER Phase VII

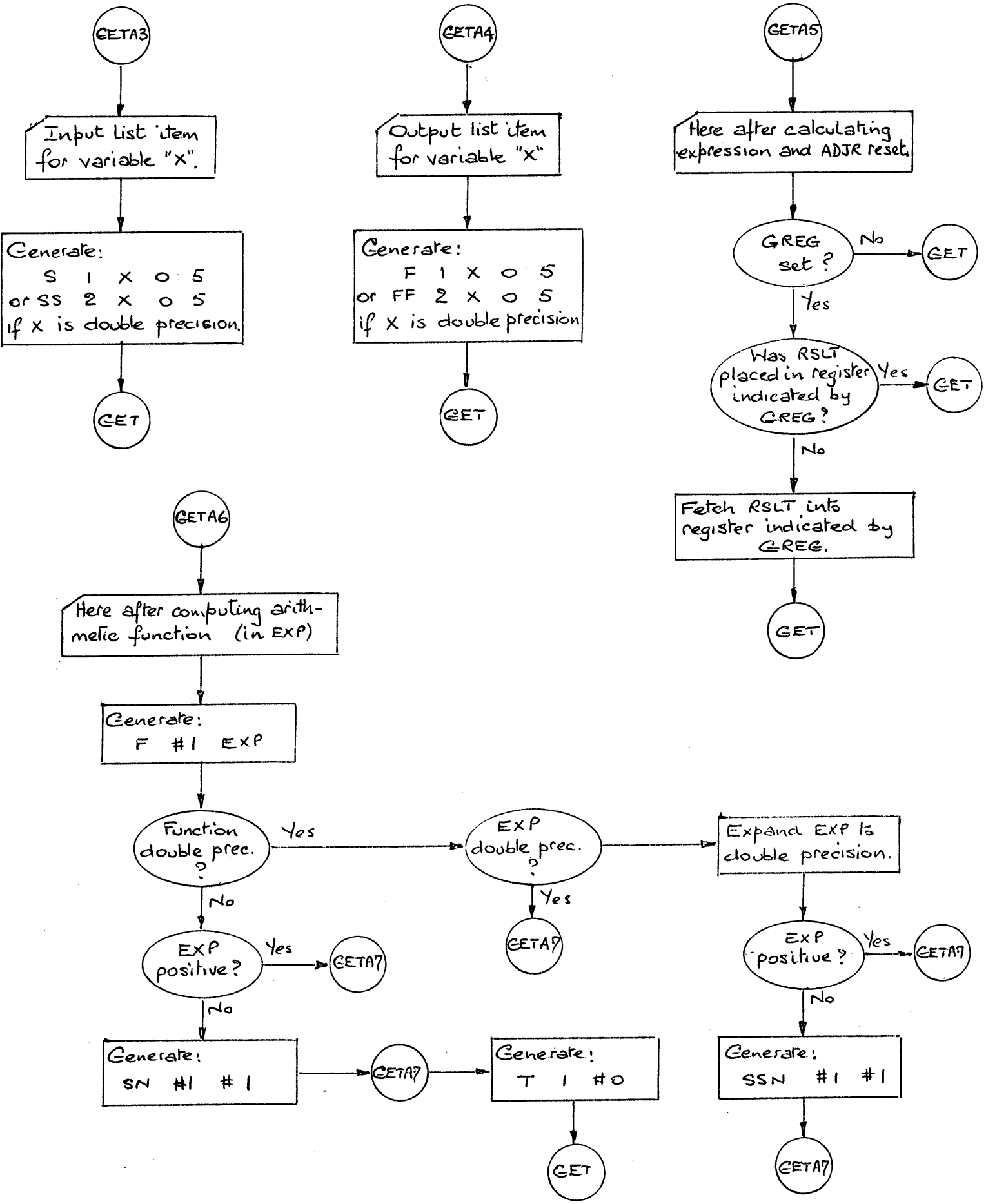
MAIN FLOW

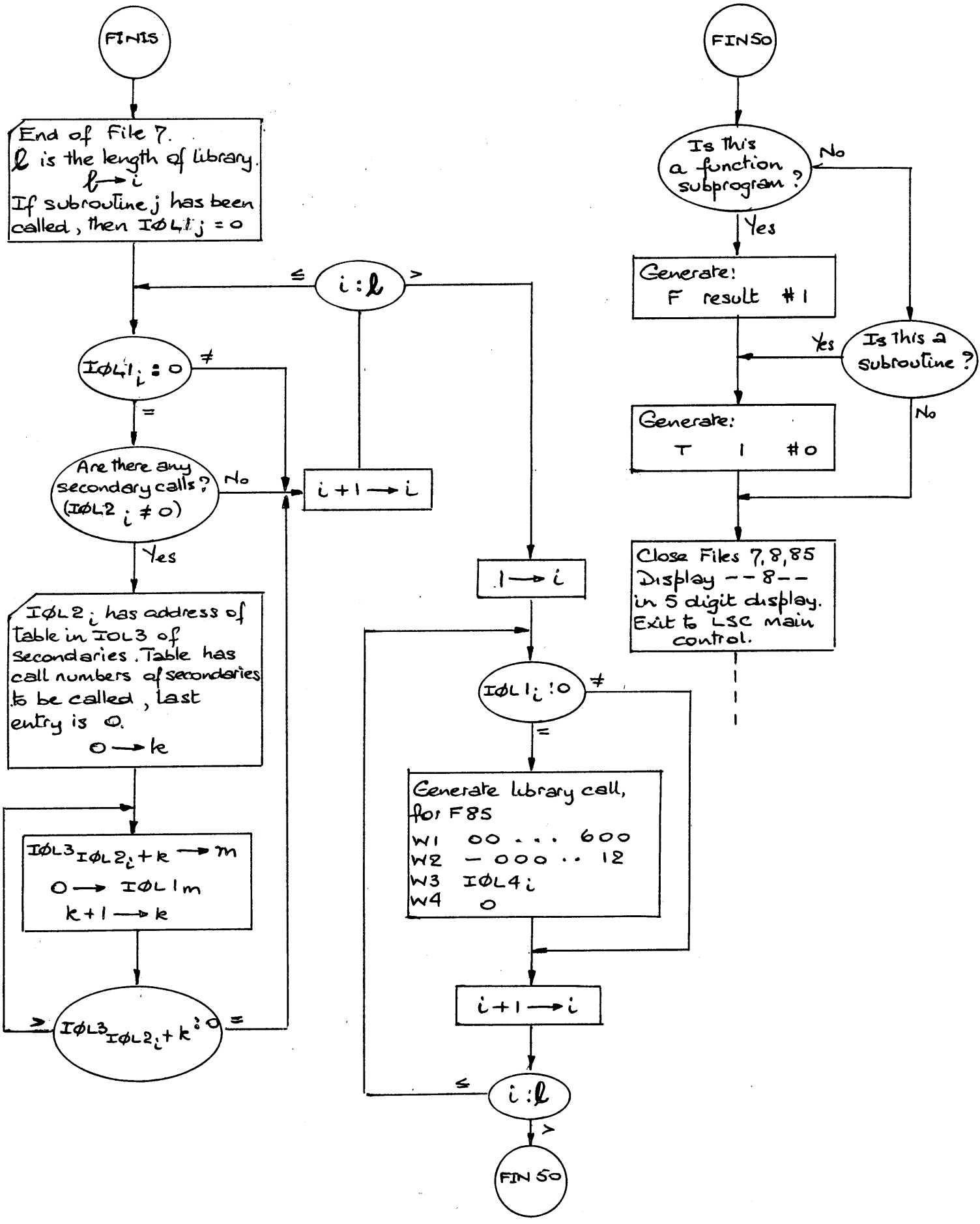


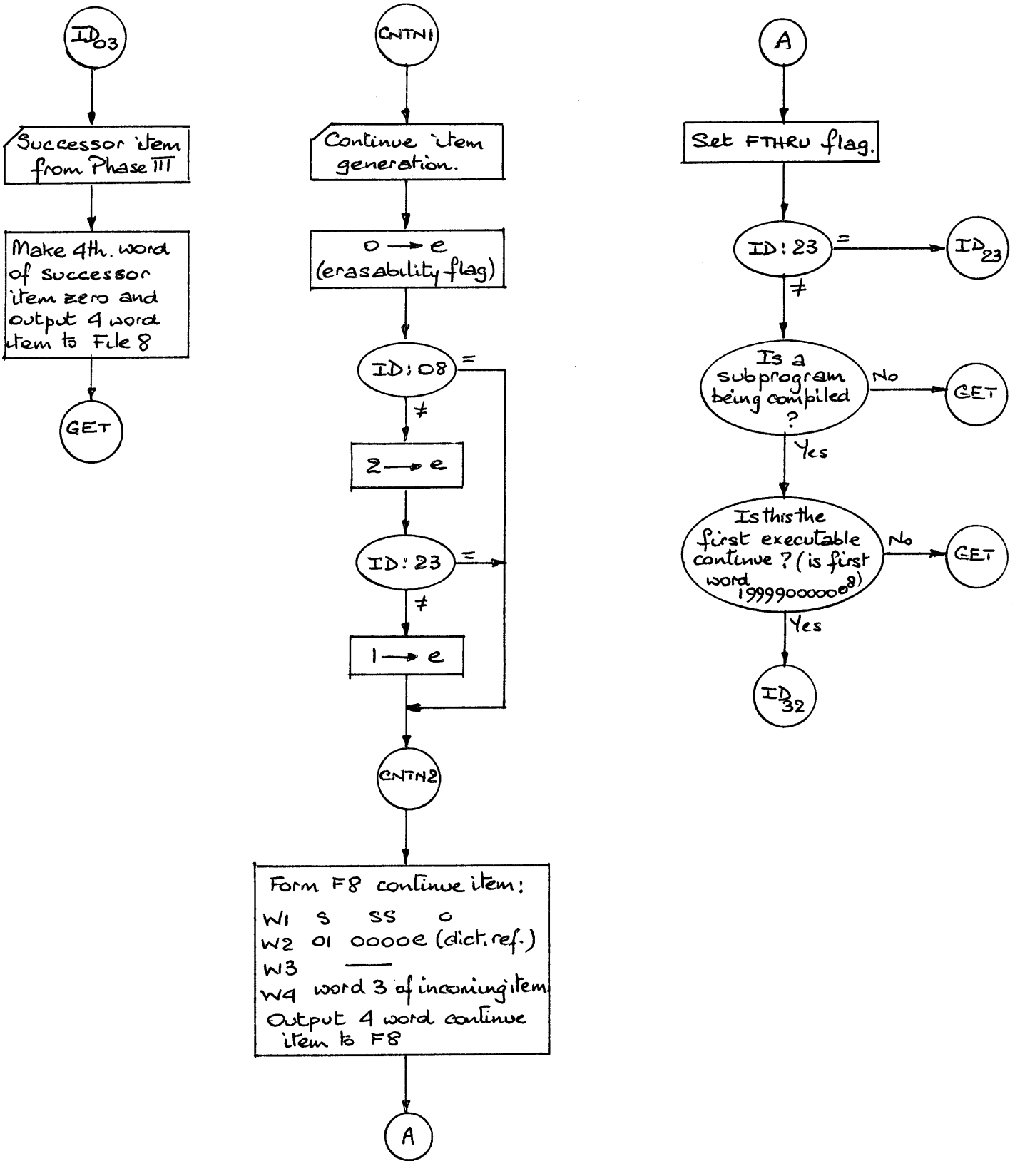

GETAI
RSLT → PRSLT
EXP → RSLT
ADJR
reset
GETAS
set
Reset ADJR
Preset GREG so that next expression is computed in the right half of the register of the last expression.
GET

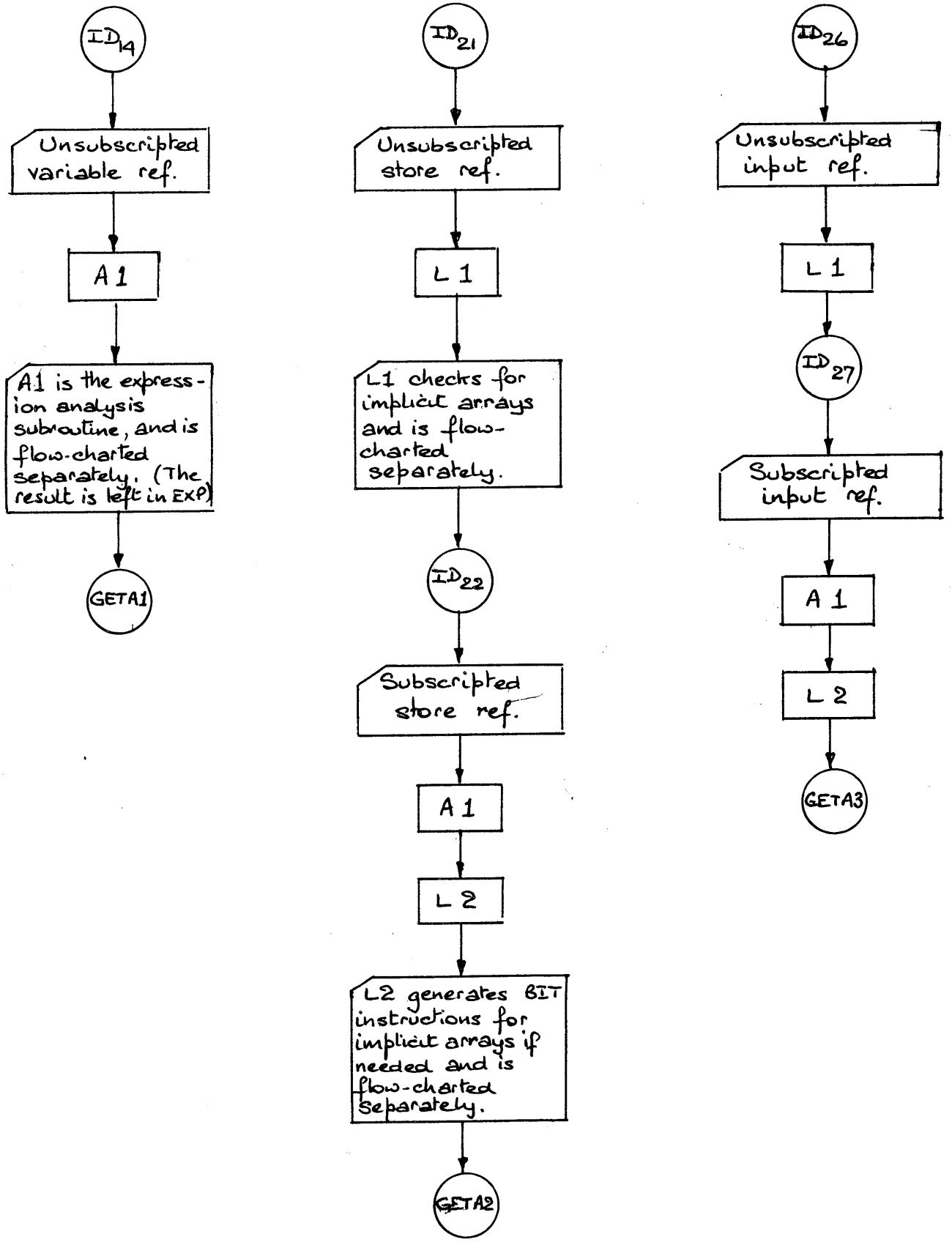

GETA2
RSLT is to be stored in EXP.
RSLT is P1
EXP is P2
P3 is temporary register parameter for P1
P1 a constant =0?
Yes
P2 double prec.?
No
Generate: S 0 P2
CET
No
Yes
P1 common subexpression?
Yes
0 → C1
No
P1 fetched?
Yes
P1 → P3
No
P2 integer?
Yes
P3 integer?
No
P1 constant?
No
Fetch P1 → P3
Yes
ETAS
Yes
Is it 0?
No
Yes
ETAS
P3 integer?
No
Generate: CX P3 61 C P3 61
Yes
ETAS
P2 double prec.?
No
Yes
ETAF
P3 double prec.?
Yes
mode P1 = P3?
Yes
No
Unfetched P1 (1 → C3
ETAF
No
P1 constant?
No
Fetch P1 to P3, expand P3
ETAS
Yes
P1 fetched?
Yes
Generate: F P3+1 (-0)
Yes
P1 negative?
No
Make P3 double precision.
No
Make P1 double precision.
ETAF
Fetch P1 into P3
ETAS
Generate: SM 0 P3+1
ETAS

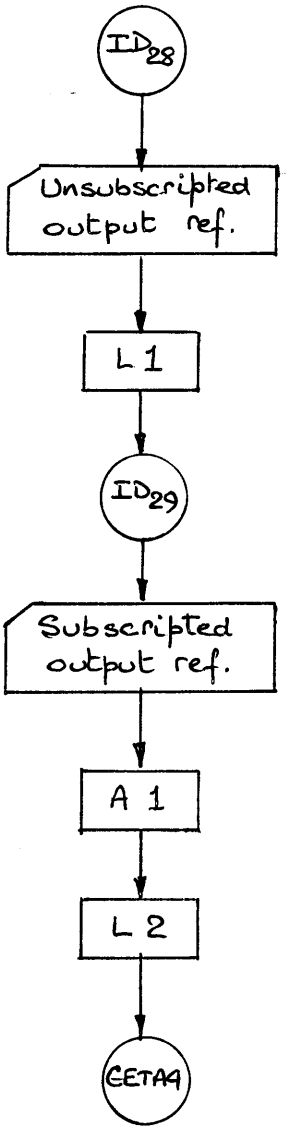

ETAS
P1 constant?
Yes
No
P1 negative?
No
P2 double prec.?
No
Generate: S P3 P2
Yes
Generate: SS P3 P2
Yes
P2 double prec?
No
Generate: SN P3 P2
Yes
Generate: SSN P3 P2
ETAR
ETAR
ETAR
L 2
CET

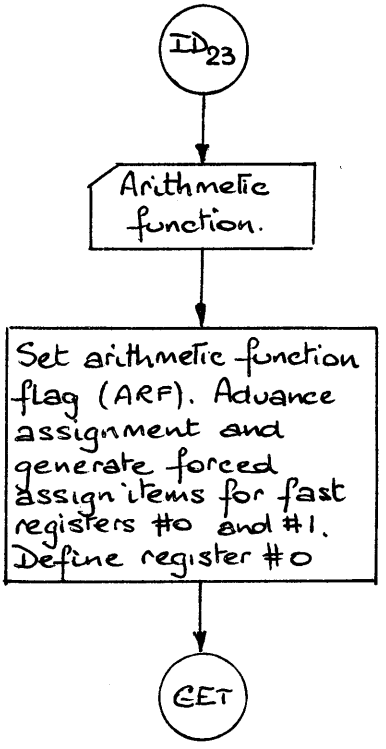

GETA3
Input list item for variable "X".
Generate:
S 1 X 0 5
or SS 2 X 0 5
if X is double precision.
GET
GETA4
Output list item for variable "X"
Generate:
F 1 X 0 5
or FF 2 X 0 5
if X is double precision
GET
GETA5
Here after calculating expression and ADJR reset.
GREG set ?
No
GET
Yes
Was RSLT placed in register indicated by GREG?
Yes
GET
No
Fetch RSLT into register indicated by GREG.
GET
GETA6
Here after computing arithmetic function (in EXP)
Generate:
F #1 EXP
Function double prec. ?
Yes
EXP double prec. ?
Yes
GETA7
Expand EXP is double precision.
EXP positive ?
Yes
GETA7
No
Generate:
SN #1 #1
GETA7
Generate:
T 1 #0
GET
No
Generate:
SSN #1 #1
GETA7

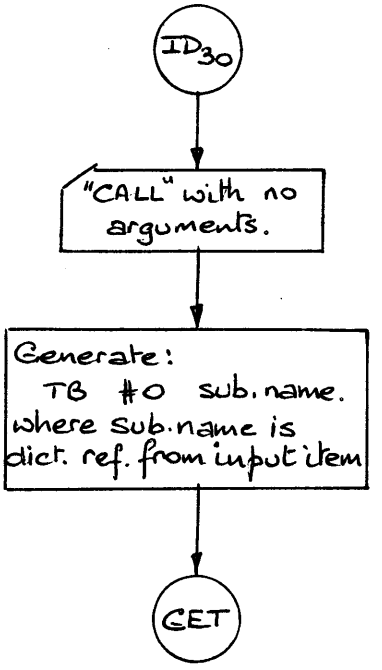

FINIS
End of File 7.
ℓ is the length of library.
ℓ → i
If subroutine j has been called, then IØL1 j = 0
i : ℓ
≤
>
IØL1 i : 0
≠
=
Are there any secondary calls? (IØL2 i ≠ 0)
No
Yes
i + 1 → i
IØL2 i has address of table in IOL3 of secondaries. Table has call numbers of secondaries to be called, last entry is 0.
0 → k
IØL3 IØL2 i + k → m
0 → IØL1 m
k + 1 → k
IØL3 IØL2 i + k : 0
1 → i
IØL1 i : 0
Generate library call, for F85
W1 00 . . . 600
W2 − 000 . . 12
W3 IØL4 i
W4 0
i + 1 → i
i : ℓ
FIN 50
FINSO
Is this a function subprogram?
No
Yes
Generate: F result #1
Is this a subroutine?
Yes
No
Generate: T 1 #0
Close Files 7, 8, 85
Display − − 8 − −
in 5 digit display.
Exit to LSC main control.

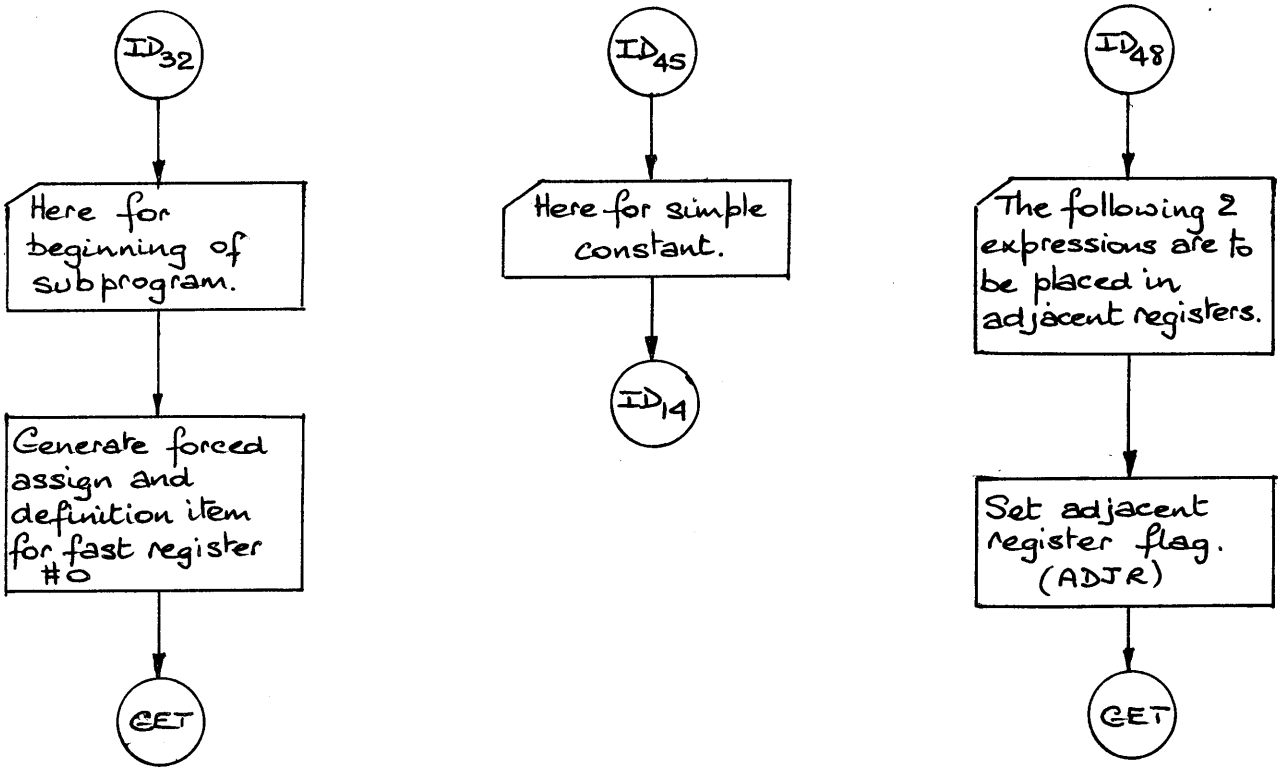

ID03
Successor item from Phase III
Make 4th. word of successor item zero and output 4 word item to File 8
GET
CNTN1
Continue item generation.
0 → e (erasability flag)
ID:08
=
≠
2 → e
ID:23
=
≠
1 → e
CNTN2
Form F8 continue item:
W1 S SS 0
W2 01 0000e (dict. ref.)
W3 —
W4 word 3 of incoming item
Output 4 word continue item to F8
A
A
Set FTHRU flag.
ID:23
=
ID23
≠
Is a subprogram being compiled ?
No
GET
Yes
Is this the first executable continue ? (is first word 1999900000e8)
No
GET
Yes
ID32

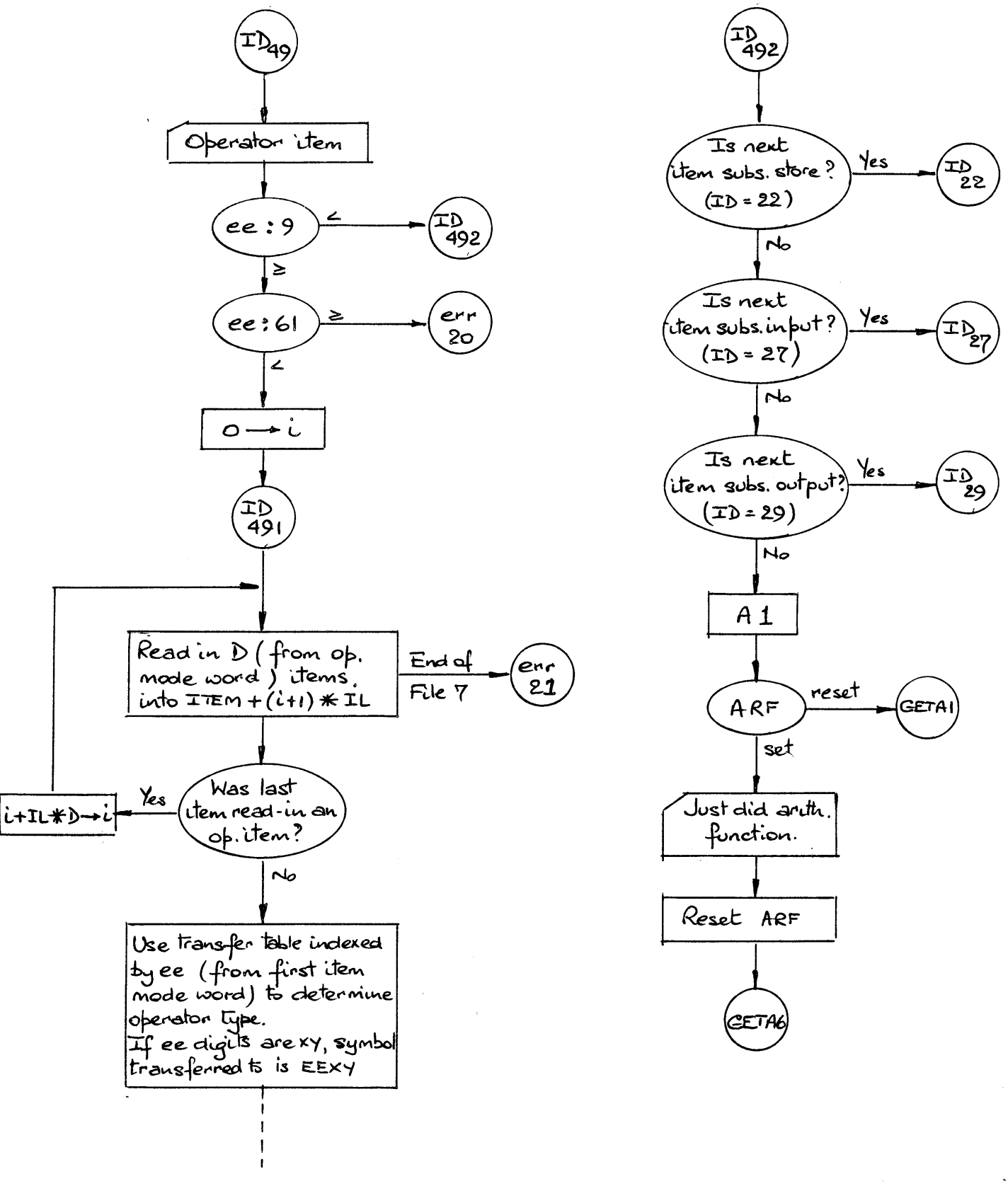

ID14
Unsubscripted variable ref.
A 1
A1 is the expression analysis subroutine, and is flow-charted separately. (The result is left in EXP)
GETA1
ID21
Unsubscripted store ref.
L 1
L1 checks for implicit arrays and is flow-charted separately.
ID22
Subscripted store ref.
A 1
L 2
L2 generates BIT instructions for implicit arrays if needed and is flow-charted separately.
GETA2
ID26
Unsubscripted input ref.
L 1
ID27
Subscripted input ref.
A 1
L 2
GETA3

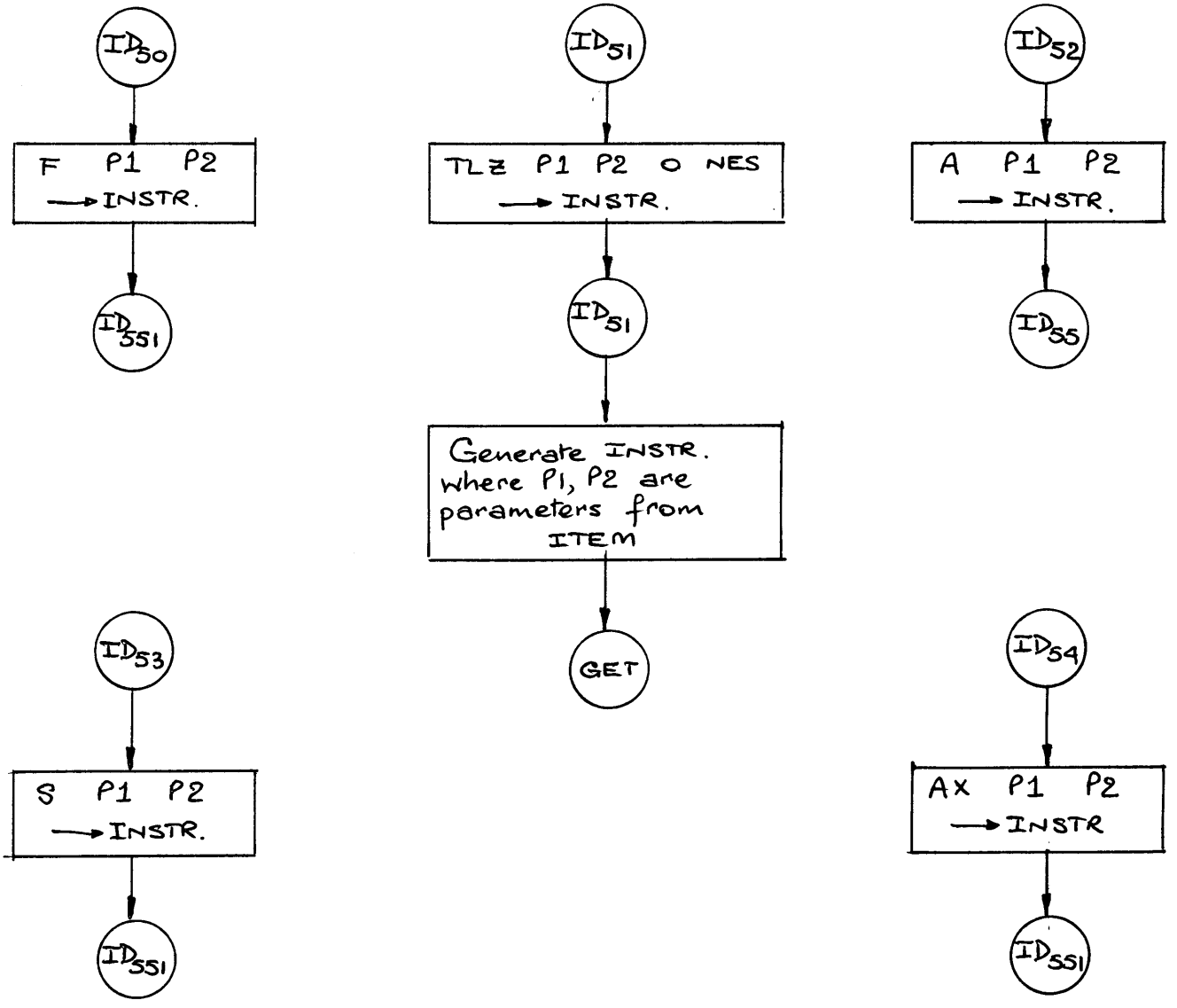

ID28
Unsubscripted output ref.
L 1
ID29
Subscripted output ref.
A 1
L 2
GETAA

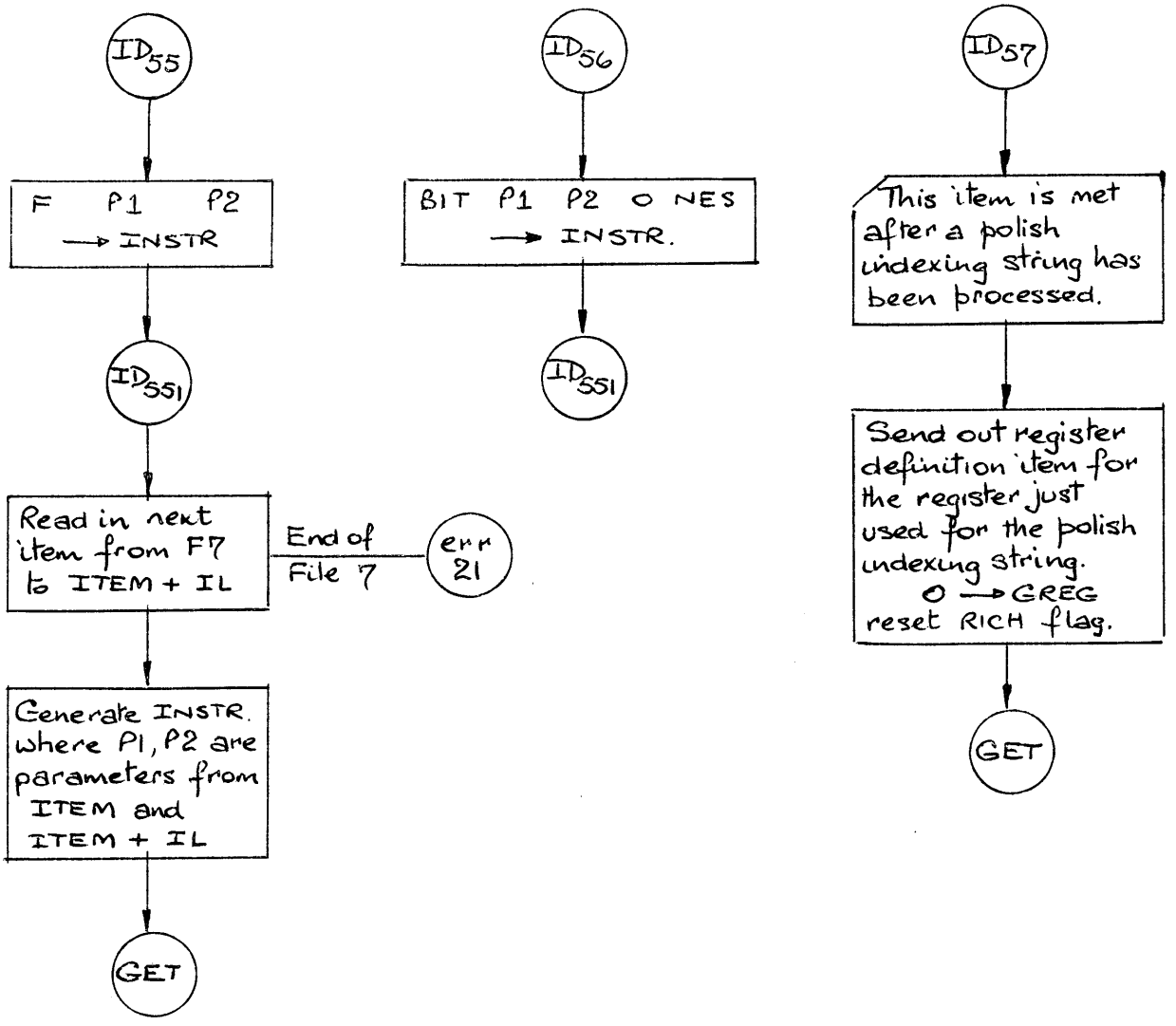

ID23
Arithmetic function.
Set arithmetic function flag (ARF). Advance assignment and generate forced assign items for fast registers #0 and #1. Define register #0
GET

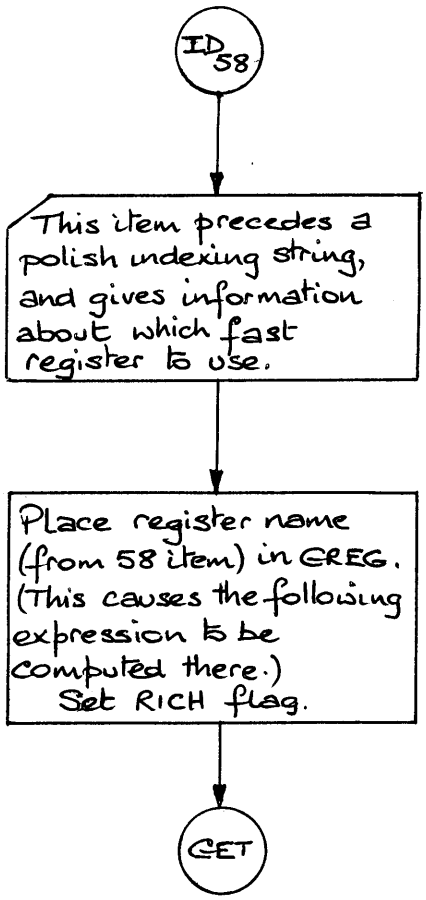

ID30
"CALL" with no arguments.
Generate:
TB #0 sub.name.
where sub.name is dict. ref. from input item
GET

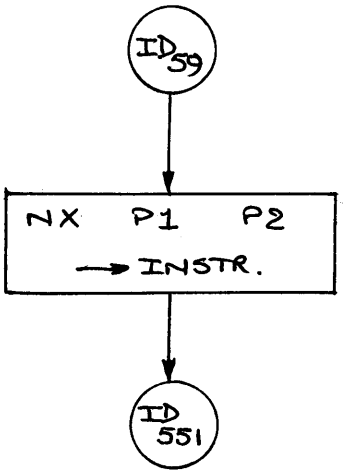

ID32
Here for beginning of subprogram.
Generate forced assign and definition item for fast register #0
GET
ID45
Here for simple constant.
ID14
ID48
The following 2 expressions are to be placed in adjacent registers.
Set adjacent register flag. (ADJR)
GET

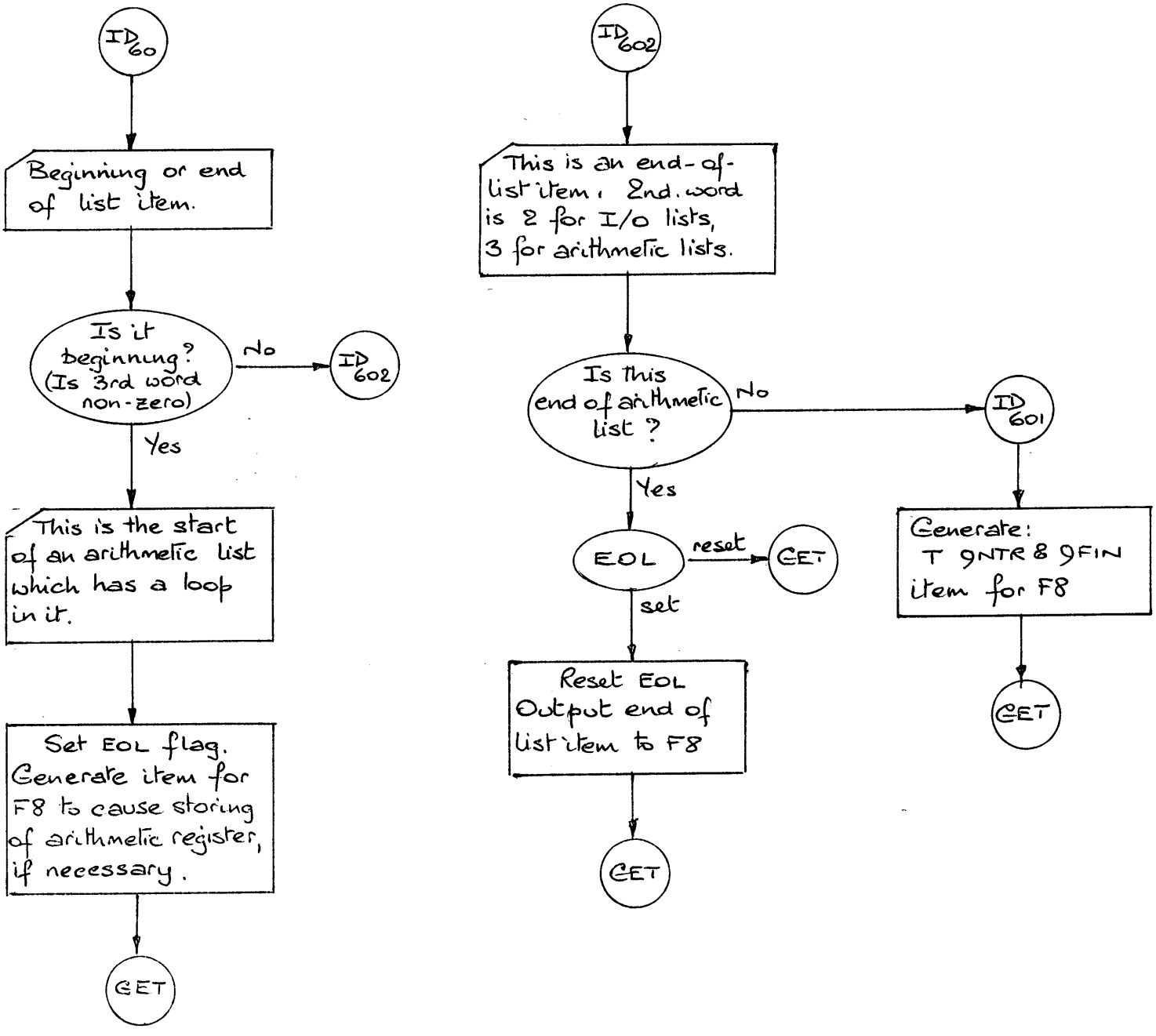

ID 49
Operator item
ee : 9
<
ID 492
≥
ee : 61
≥
err 20
<
0 → i
ID 491
Read in D ( from op. mode word ) items, into ITEM + (i+1) * IL
End of File 7
err 21
Was last item read-in an op. item?
Yes
i+IL*D→i
No
Use transfer table indexed by ee (from first item mode word) to determine operator type.
If ee digits are XY, symbol transferred to is EEXY
ID 492
Is next item subs. store? (ID = 22)
Yes
ID 22
No
Is next item subs. input? (ID = 27)
Yes
ID 27
No
Is next item subs. output? (ID = 29)
Yes
ID 29
No
A 1
ARF
reset
GETA1
set
Just did arith. function.
Reset ARF
GETA6

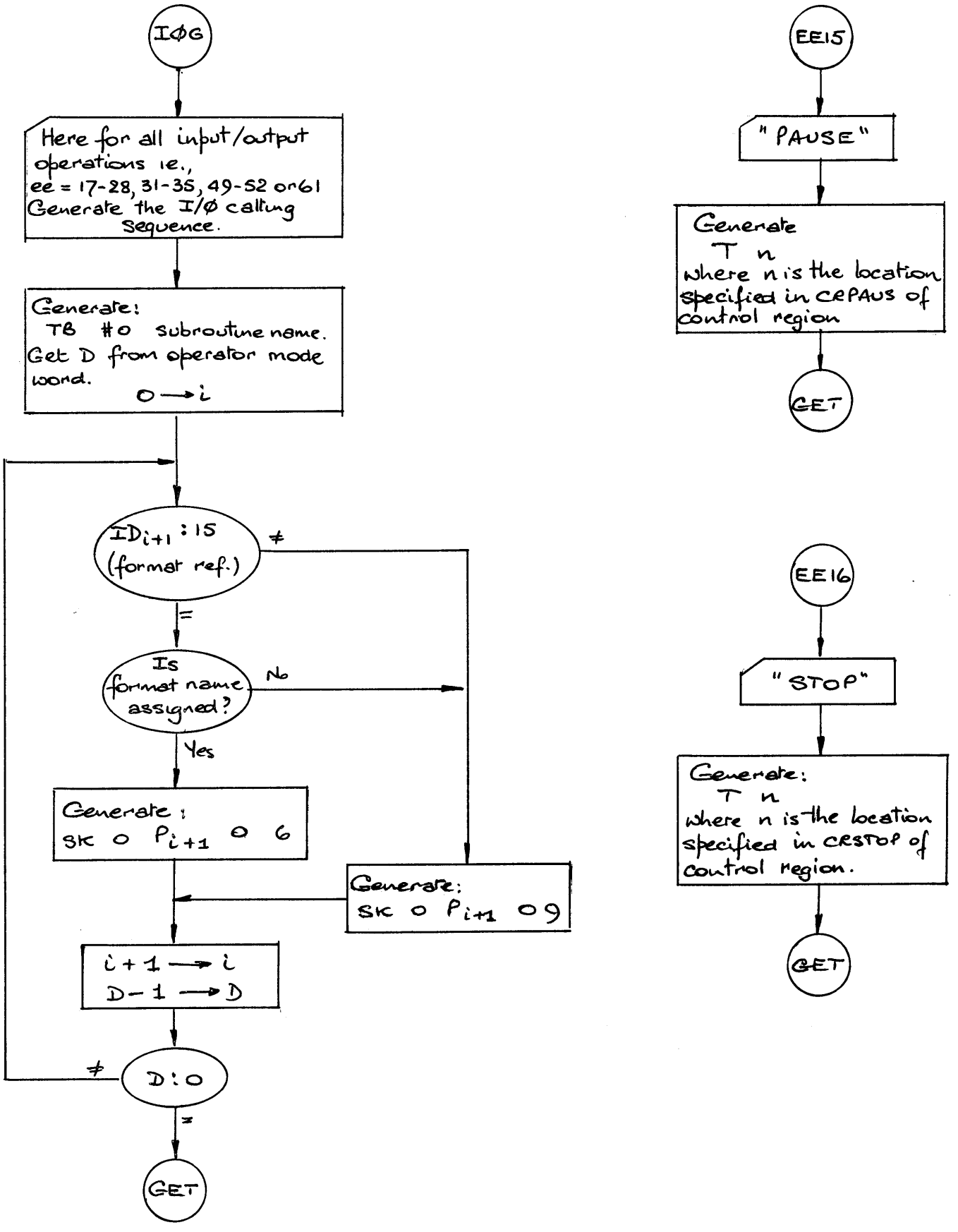

ID50
F P1 P2
→ INSTR.
ID551
ID51
TLZ P1 P2 0 NES
→ INSTR.
ID51
Generate INSTR. where P1, P2 are parameters from ITEM
GET
ID52
A P1 P2
→ INSTR.
ID55
ID53
S P1 P2
→ INSTR.
ID551
ID54
AX P1 P2
→ INSTR
ID551


ID55
F P1 P2
→ INSTR
ID551
Read in next item from F7 to ITEM + IL
End of File 7
err 21
Generate INSTR. where P1, P2 are parameters from ITEM and ITEM + IL
GET
ID56
BIT P1 P2 O NES
→ INSTR.
ID551
ID57
This item is met after a polish indexing string has been processed.
Send out register definition item for the register just used for the polish indexing string.
O → GREG
reset RICH flag.
GET

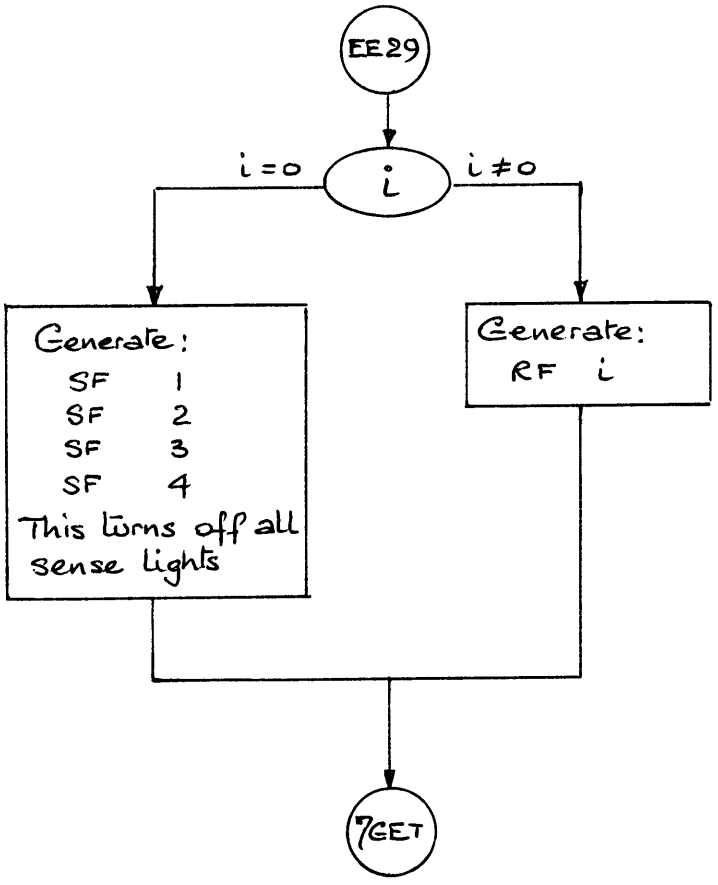

ID 58
This item precedes a polish indexing string, and gives information about which fast register to use.
Place register name (from 58 item) in GREG. (This causes the following expression to be computed there.)
Set RICH flag.
GET

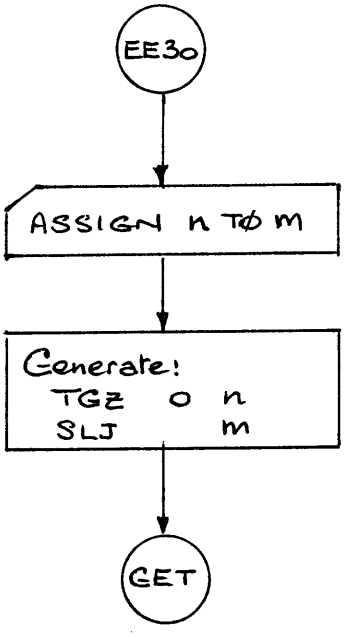

ID 59
NX P1 P2
→ INSTR.
ID 551


ID 60
Beginning or end of list item.
Is it beginning? (Is 3rd word non-zero)
No
ID 602
Yes
This is the start of an arithmetic list which has a loop in it.
Set EOL flag. Generate item for F8 to cause storing of arithmetic register, if necessary.
GET
ID 602
This is an end-of-list item; 2nd. word is 2 for I/O lists, 3 for arithmetic lists.
Is this end of arithmetic list ?
No
ID 601
Yes
EOL
reset
GET
set
Reset EOL Output end of list item to F8
GET
Generate: T 9NTR & 9FIN item for F8
GET


IØG
Here for all input/output operations ie.,
ee = 17-28, 31-35, 49-52 or 61
Generate the I/Ø calling sequence.
Generate:
TB #0 subroutine name.
Get D from operator mode word.
0 → i
IDi+1 : 15
(format ref.)
≠
=
Is format name assigned?
No
Yes
Generate:
SK 0 Pi+1 0 6
Generate:
SK 0 Pi+1 0 9
i + 1 → i
D - 1 → D
≠
D : 0
=
GET
EE15
" PAUSE "
Generate
T n
where n is the location specified in CRPAUS of control region
GET
EE16
" STOP "
Generate:
T n
where n is the location specified in CRSTOP of control region.
GET

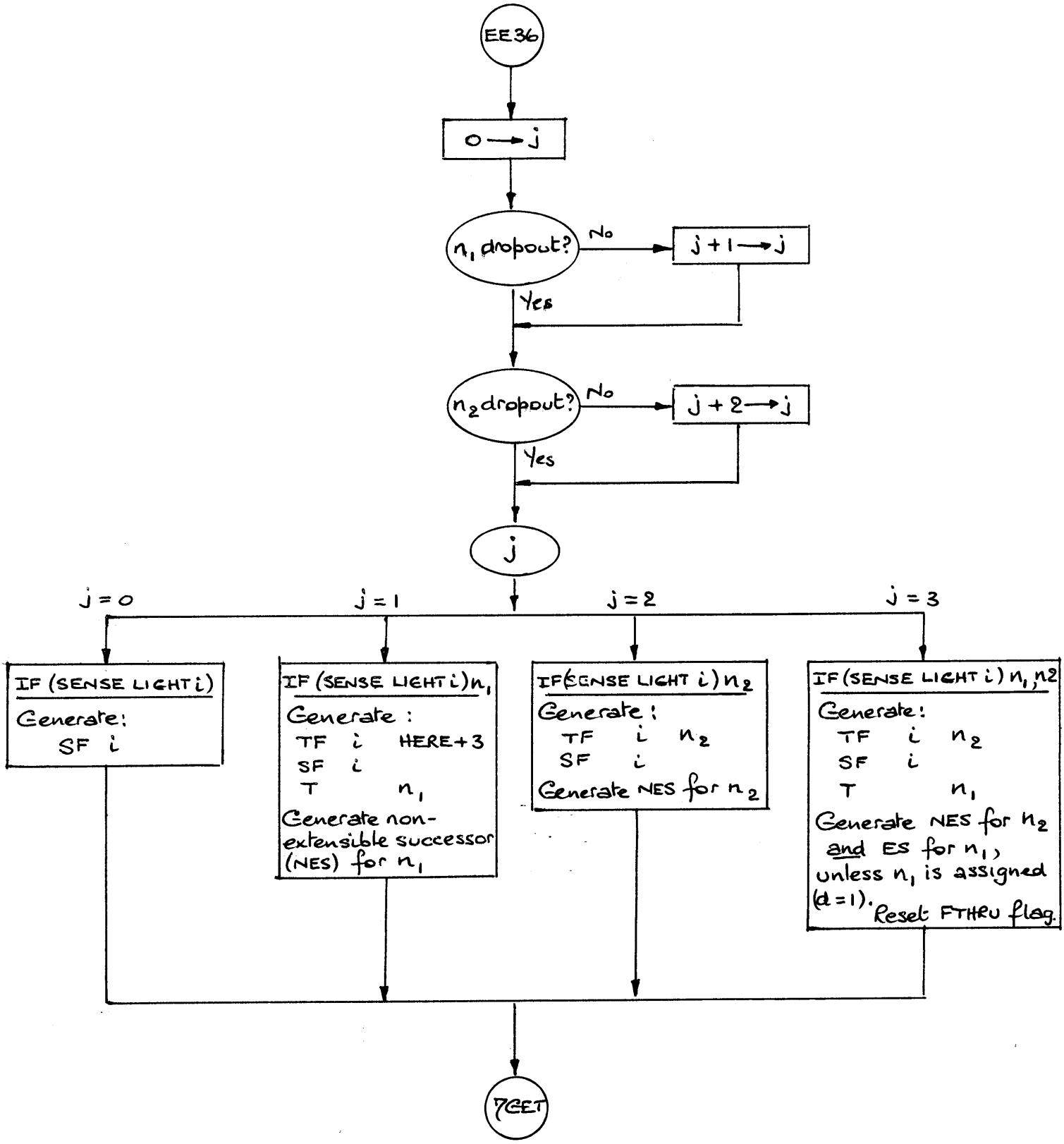
SENSE LIGHT i

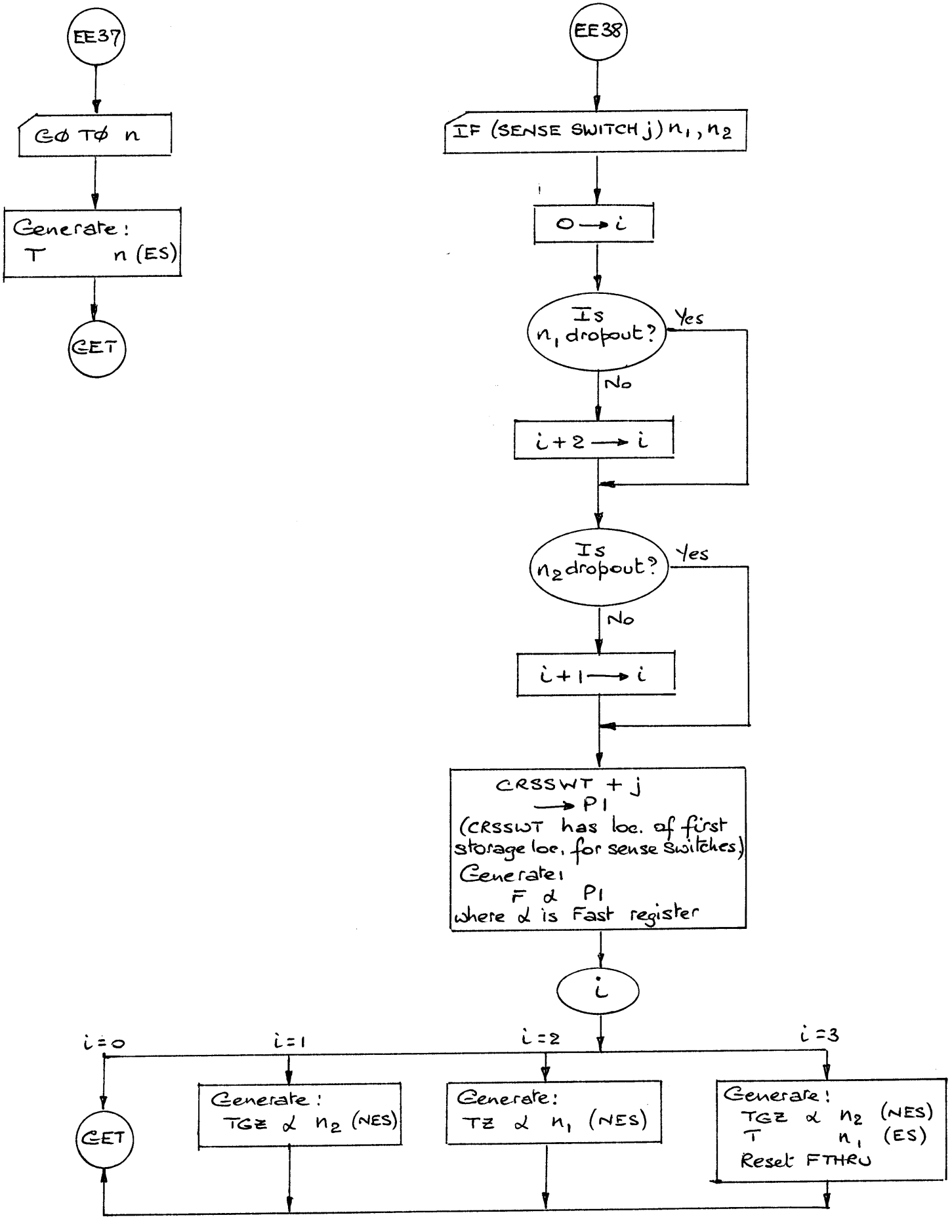


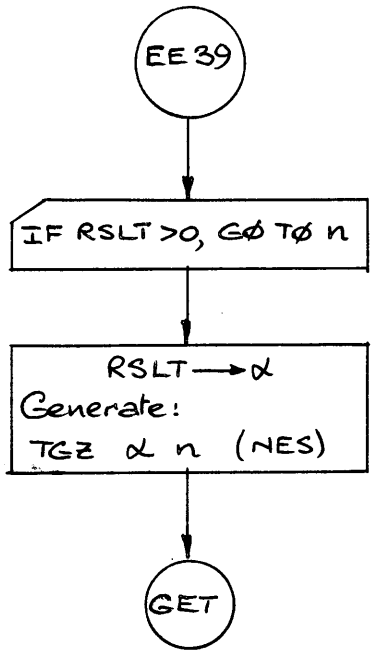



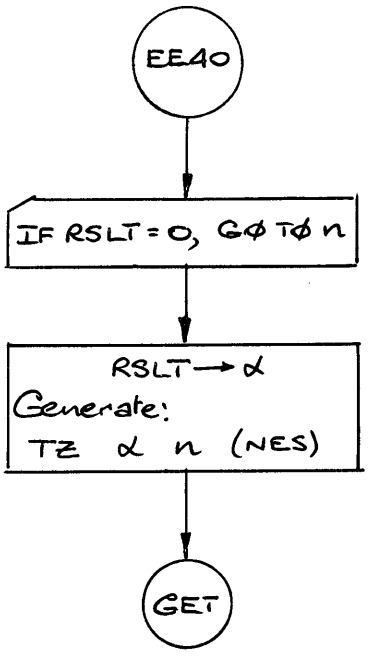

Note: Sense light is on if flip flop i is reset.

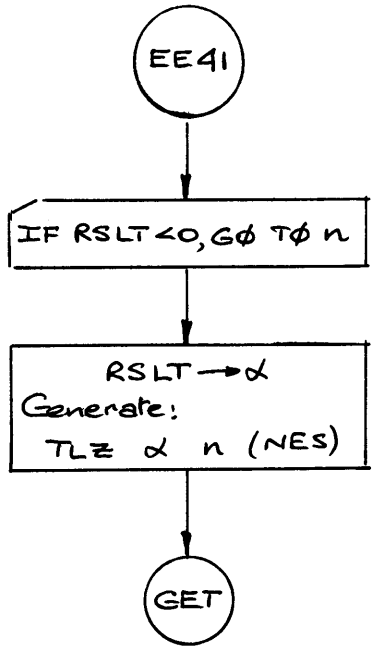
IF (SENSE LIGHT i) $n_1$, $n_2$

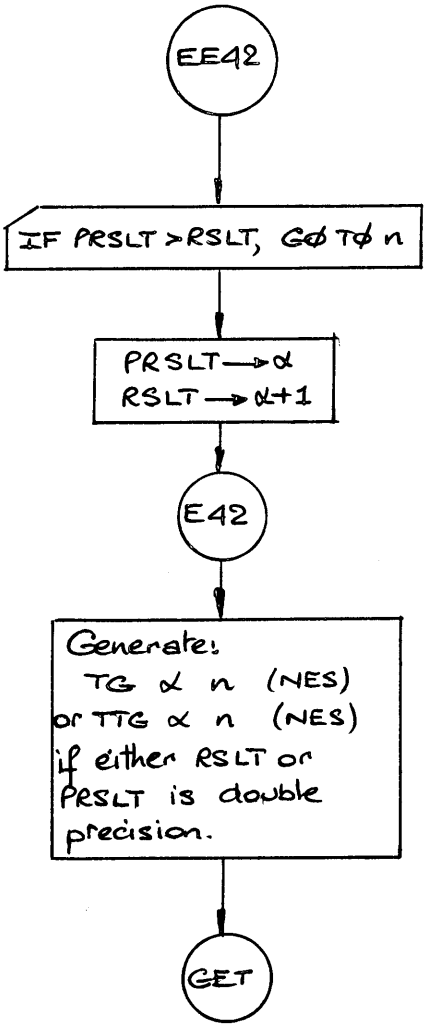


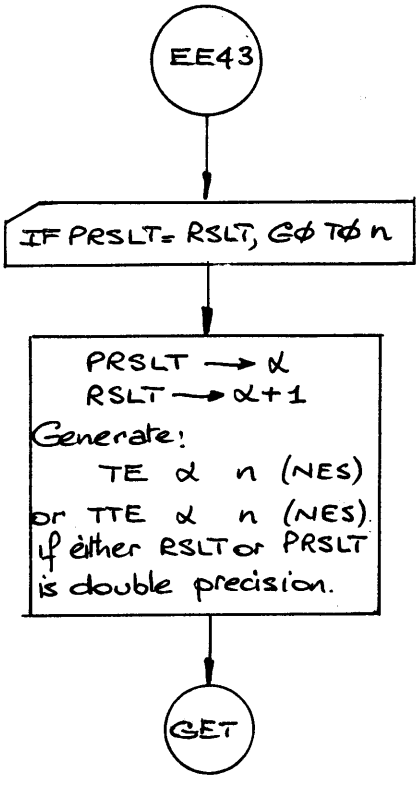

EE37
GO TO n
Generate:
T n (ES)
GET
EE38
IF (SENSE SWITCH j) n1, n2
0 → i
Is n1 dropout?
Yes
No
i+2 → i
Is n2 dropout?
Yes
No
i+1 → i
CRSSWT + j
→ P1
(CRSSWT has loc. of first storage loc. for sense switches)
Generate:
F α P1
where α is Fast register
i
i=0
i=1
i=2
i=3
GET
Generate:
TGE α n2 (NES)
Generate:
TE α n1 (NES)
Generate:
TGE α n2 (NES)
T n1 (ES)
Reset FTHRU

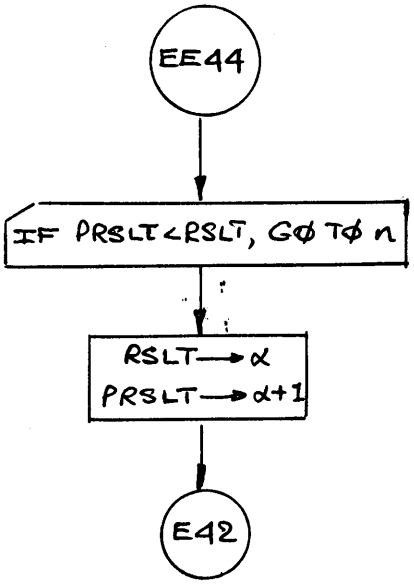

EE39
IF RSLT >0, GØ TØ n
RSLT → α
Generate:
TGZ α n (NES)
GET

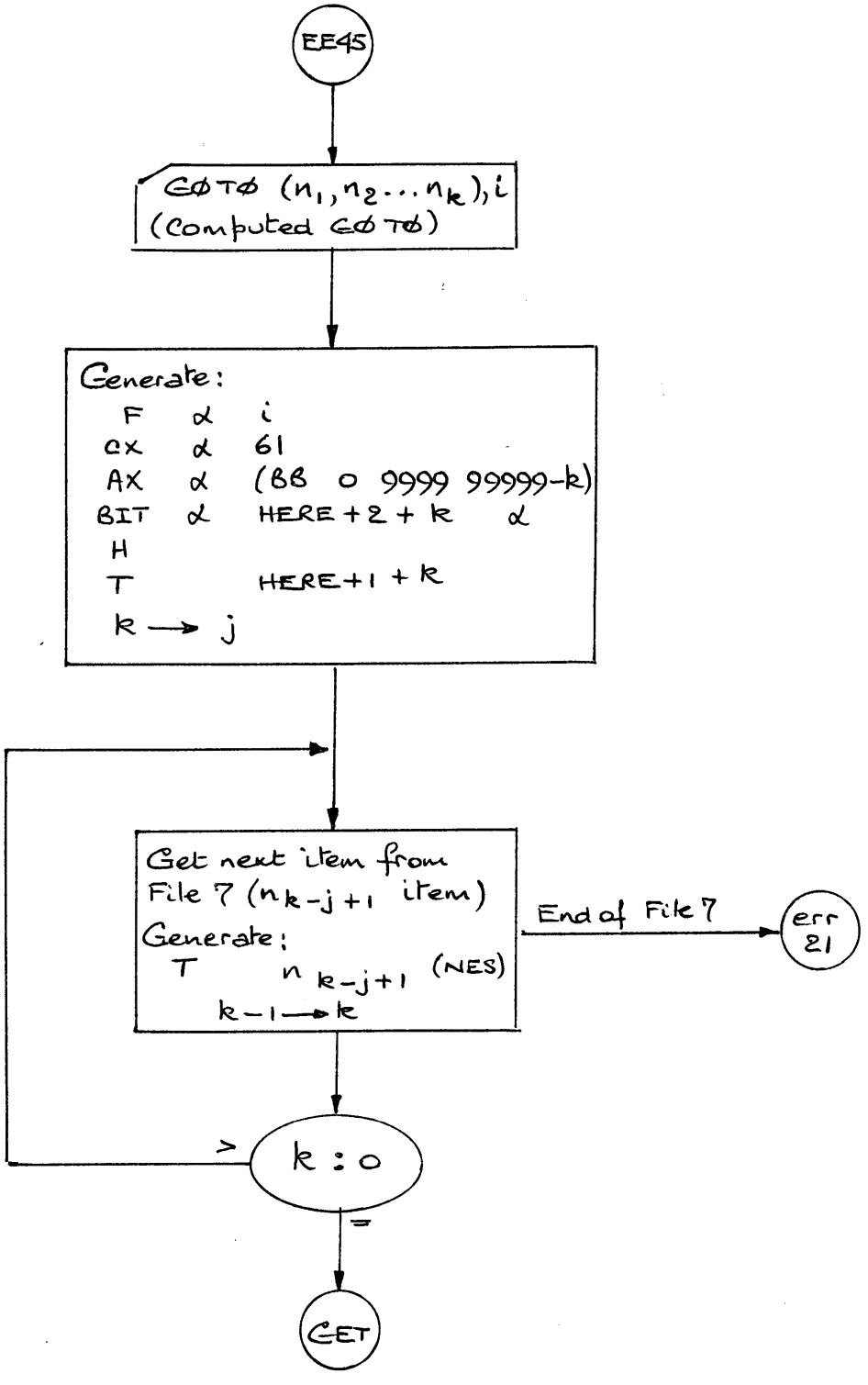

EE40
IF RSLT = 0, GØ TØ n
RSLT → α
Generate:
TZ α n (NES)
GET


EE41
IF RSLT <0, GØ TØ n
RSLT → α
Generate:
TLZ α n (NES)
GET

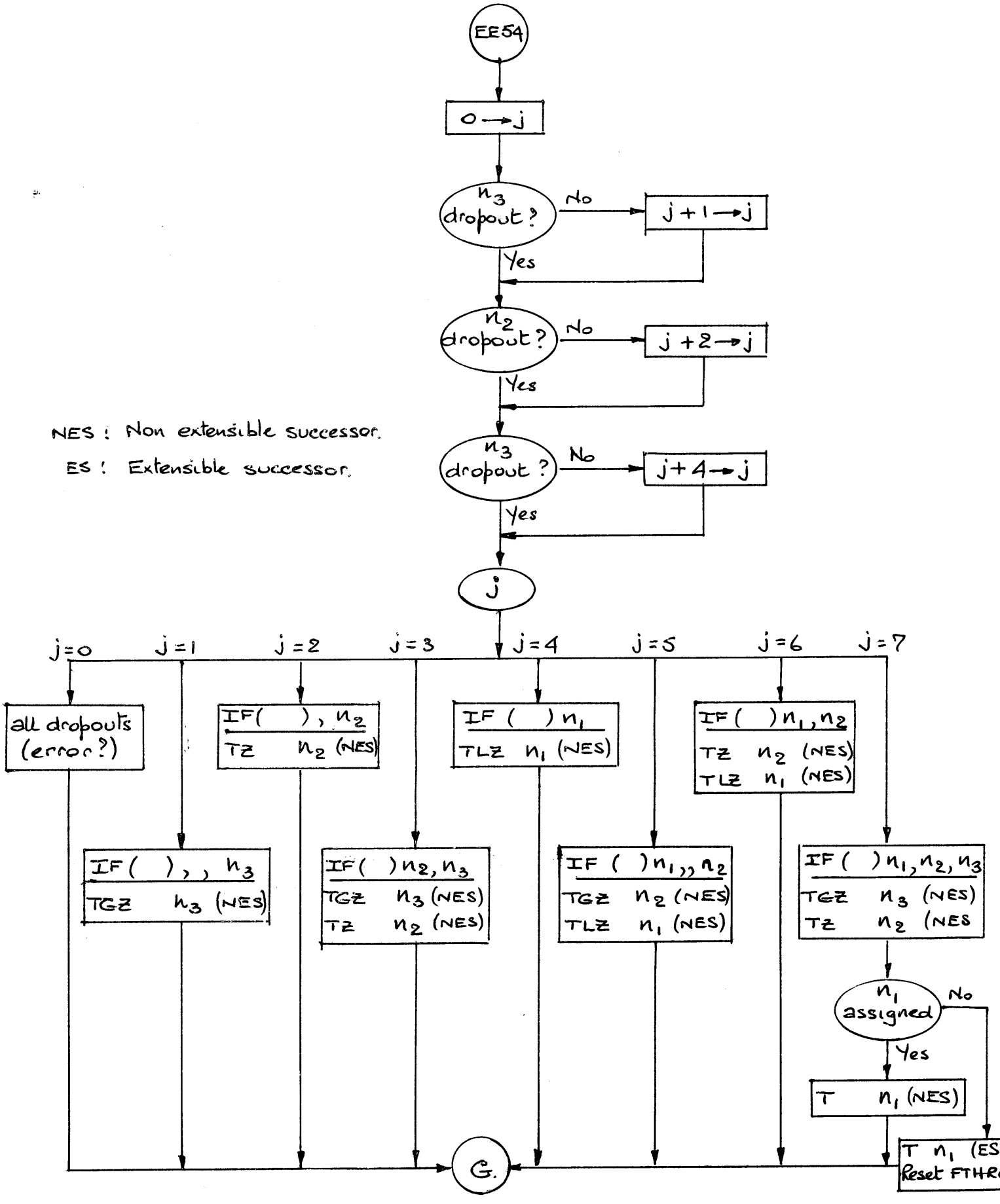

EE42
IF PRSLT >RSLT, GØ TØ n
PRSLT → α
RSLT → α+1
E42
Generate:
TG α n (NES)
or TTG α n (NES)
if either RSLT or
PRSLT is double
precision.
GET

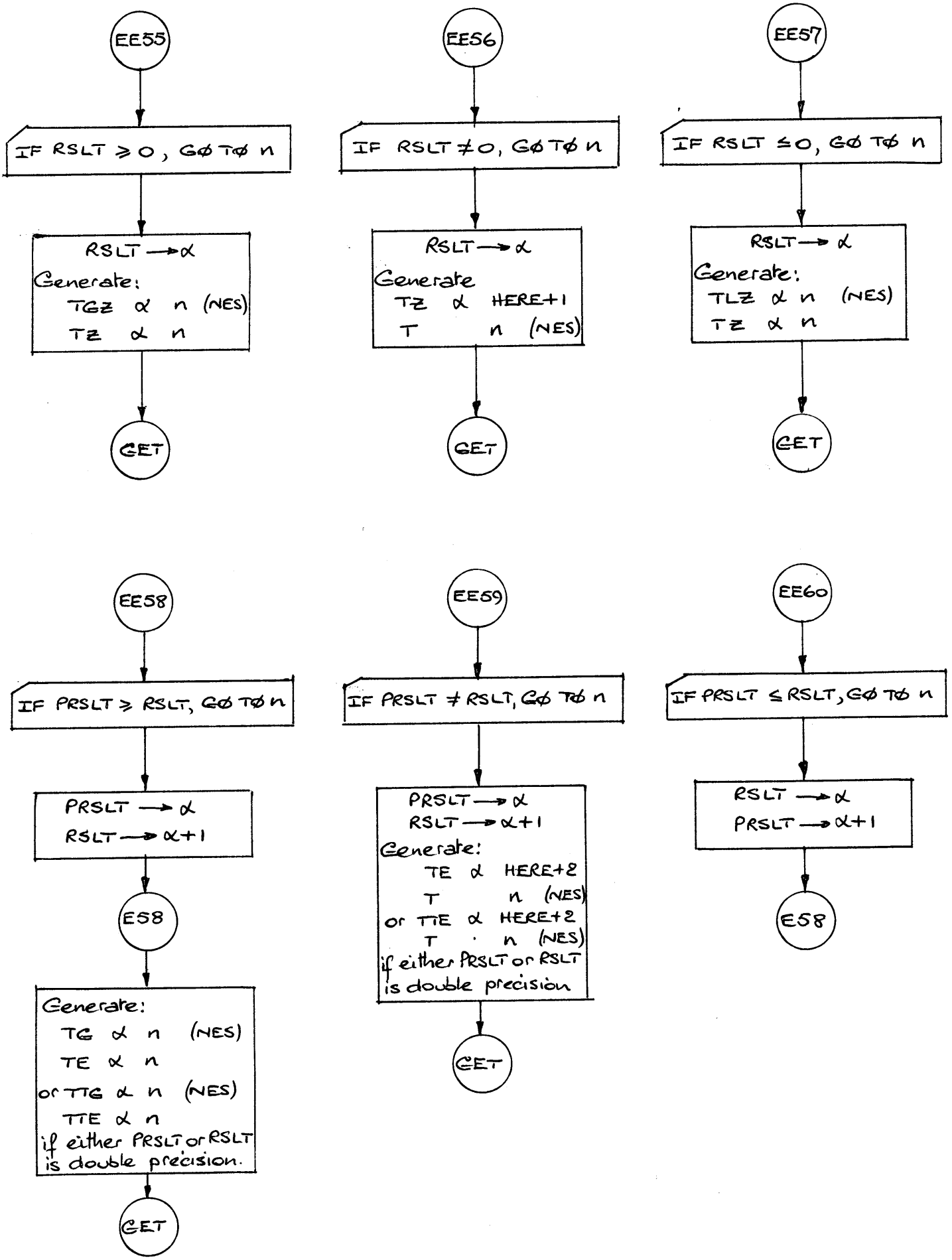

EE43
IF PRSLT= RSLT, GØ TØ n
PRSLT → α
RSLT → α+1
Generate:
TE α n (NES)
or TTE α n (NES)
if either RSLT or PRSLT
is double precision.
GET

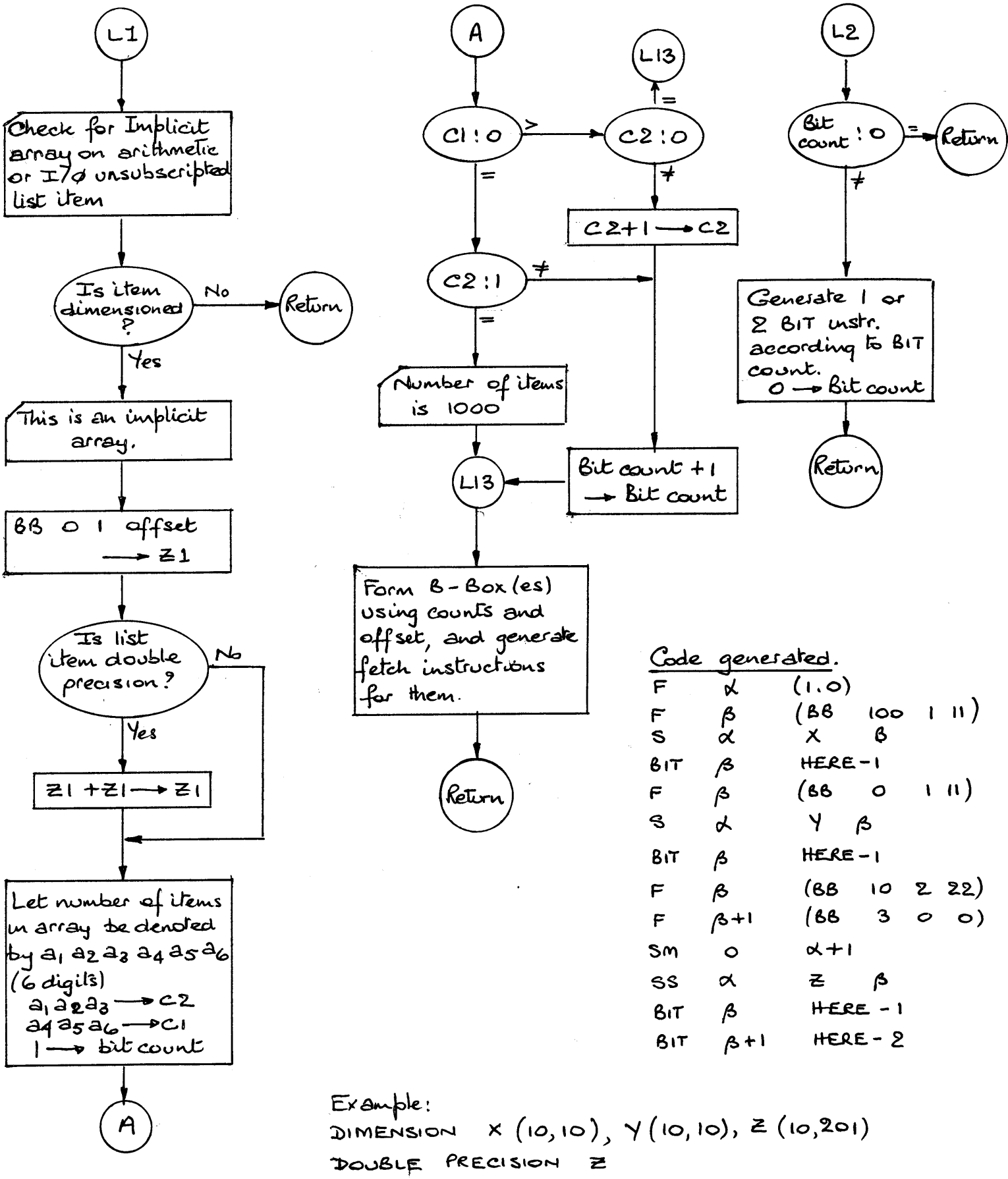

EE44
IF PRSLT <RSLT, GØ TØ n
RSLT → α
PRSLT → α+1
E42


EE45
GOTO (n1, n2 ... nk), i
(Computed GO TO)
Generate:
F α i
CX α 61
AX α (BB 0 9999 99999-k)
BIT α HERE+2+k α
H
T HERE+1+k
k → j
Get next item from
File 7 (nk-j+1 item)
Generate:
T nk-j+1 (NES)
k-1 → k
End of File 7
err
21
k : 0
>
=
GET

IF ( ) $n_1, n_2, n_3$


EE54
0 → j
$n_3$ dropout ?
No
j + 1 → j
Yes
$n_2$ dropout ?
No
j + 2 → j
Yes
$n_3$ dropout ?
No
j + 4 → j
Yes
j
NES : Non extensible successor.
ES : Extensible successor.
j=0
j=1
j=2
j=3
j=4
j=5
j=6
j=7
all dropouts (error?)
IF( ), $n_2$
TZ $n_2$ (NES)
IF ( ) $n_1$
TLZ $n_1$ (NES)
IF ( ) $n_1$, $n_2$
TZ $n_2$ (NES)
TLZ $n_1$ (NES)
IF( ), , $n_3$
TGZ $n_3$ (NES)
IF( ) $n_2, n_3$
TGZ $n_3$ (NES)
TZ $n_2$ (NES)
IF ( ) $n_1$, , $n_2$
TGZ $n_2$ (NES)
TLZ $n_1$ (NES)
IF ( ) $n_1, n_2, n_3$
TGZ $n_3$ (NES)
TZ $n_2$ (NES
$n_1$ assigned
No
Yes
T $n_1$ (NES)
T $n_1$ (ES)
Reset FTHRU
G.


EE55
IF RSLT ≥ 0, GØ TØ n
RSLT → α
Generate:
TGZ α n (NES)
TZ α n
GET
EE56
IF RSLT ≠ 0, GØ TØ n
RSLT → α
Generate
TZ α HERE+1
T n (NES)
GET
EE57
IF RSLT ≤ 0, GØ TØ n
RSLT → α
Generate:
TLZ α n (NES)
TZ α n
GET
EE58
IF PRSLT ≥ RSLT, GØ TØ n
PRSLT → α
RSLT → α+1
E58
Generate:
TG α n (NES)
TE α n
or TTG α n (NES)
TTE α n
if either PRSLT or RSLT is double precision.
GET
EE59
IF PRSLT ≠ RSLT, GØ TØ n
PRSLT → α
RSLT → α+1
Generate:
TE α HERE+2
T n (NES)
or TTE α HERE+2
T n (NES)
if either PRSLT or RSLT is double precision
GET
EE60
IF PRSLT ≤ RSLT, GØ TØ n
RSLT → α
PRSLT → α+1
E58




Code generated.

| | | |
|---|---|---|
| F | α | (1.0) |
| F | β | (BB 100 1 11) |
| S | α | X β |
| BIT | β | HERE-1 |
| F | β | (BB 0 1 11) |
| S | α | Y β |
| BIT | β | HERE-1 |
| F | β | (BB 10 2 22) |
| F | β+1 | (BB 3 0 0) |
| SM | 0 | α+1 |
| SS | α | Z β |
| BIT | β | HERE-1 |
| BIT | β+1 | HERE-2 |

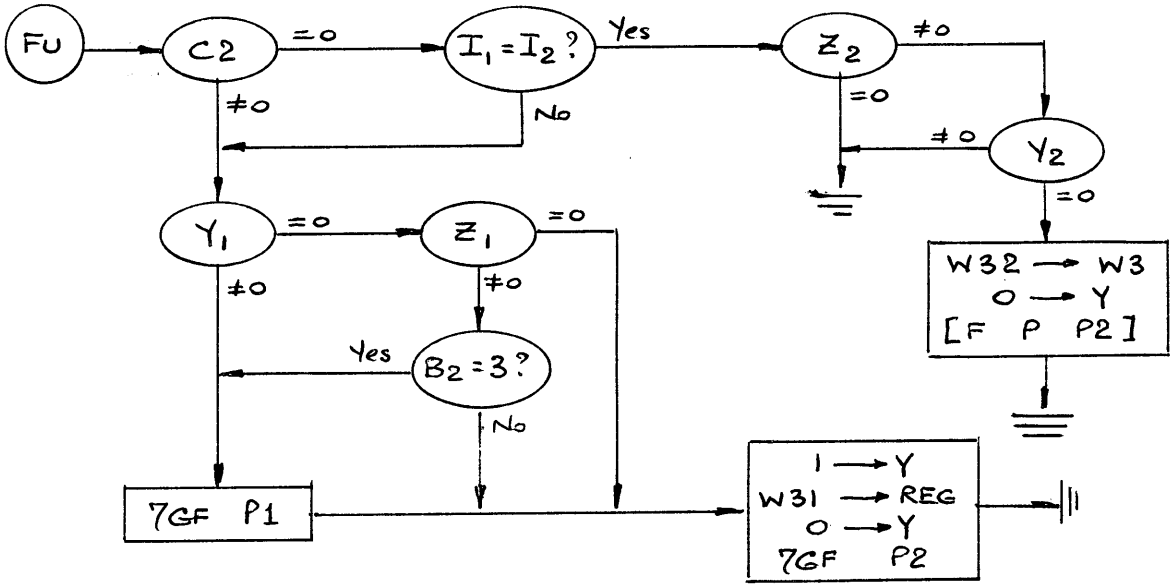
Example:

DIMENSION X(10,10), Y(10,10), Z(10,201)

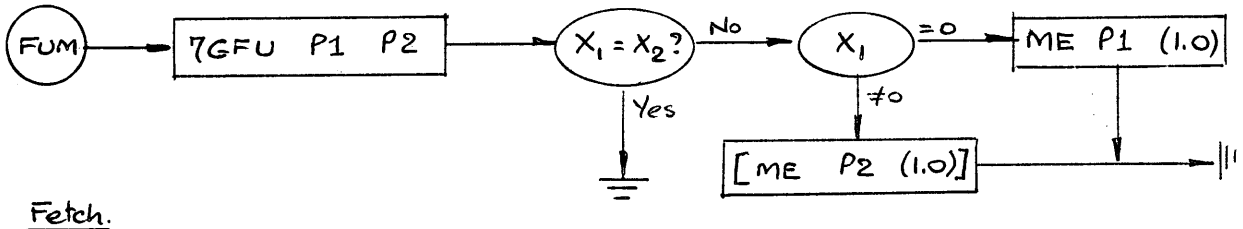
DOUBLE PRECISION Z

X, Y, Z = 1

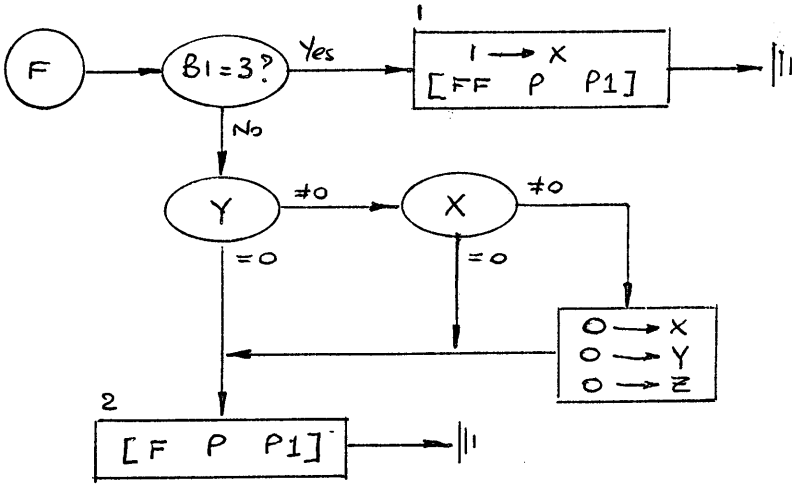
## Fetch Upper



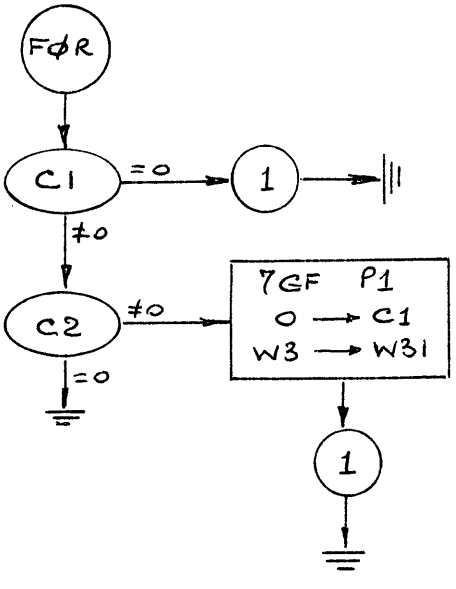

## Fetch Upper & Match.



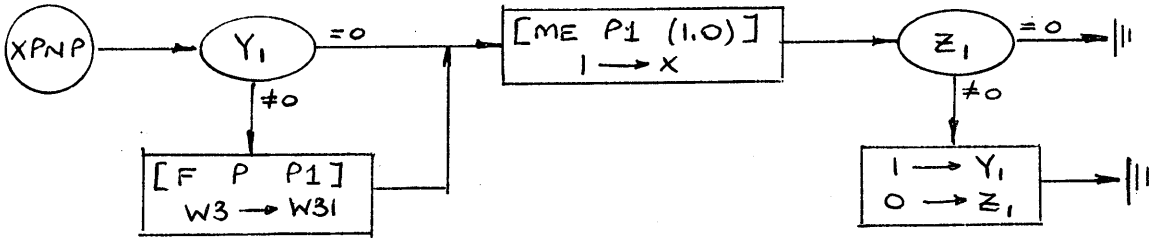

## Fetch.

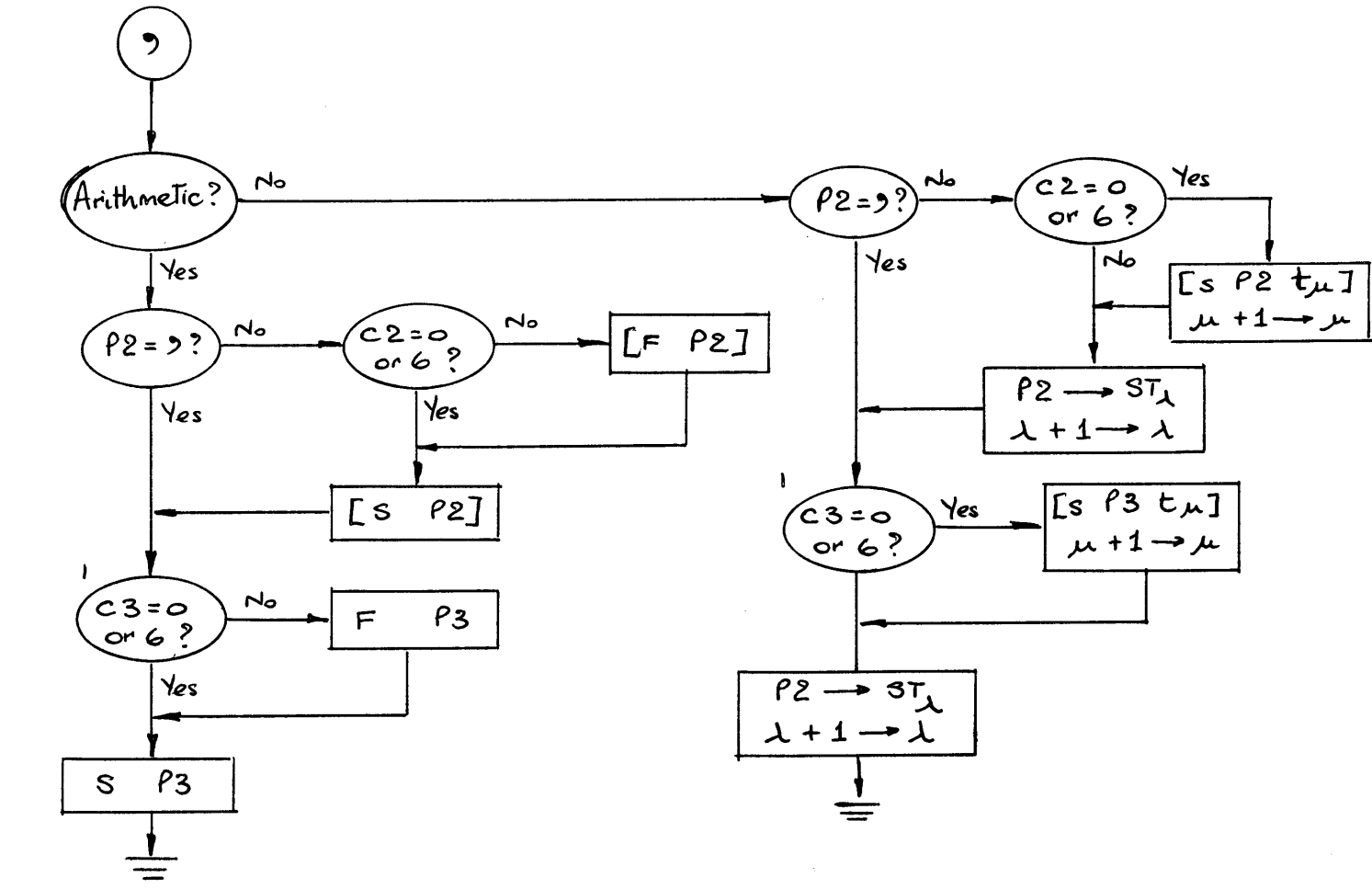


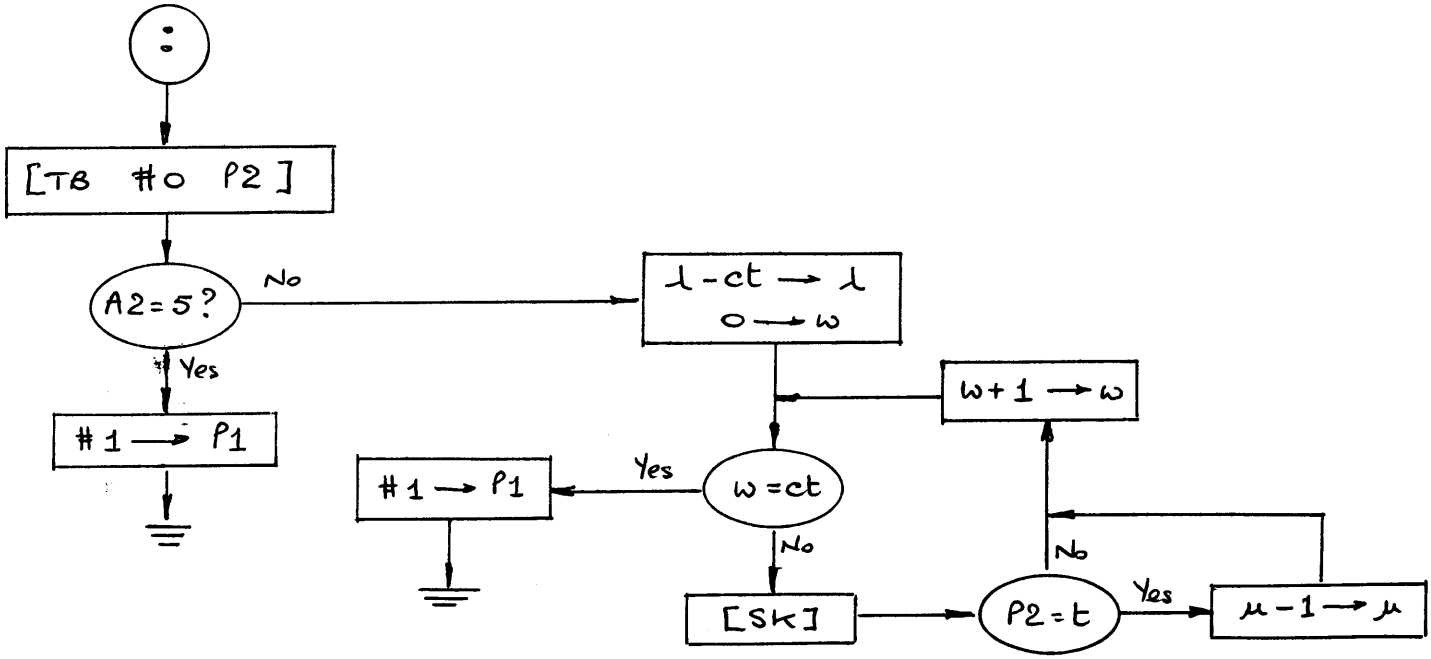

## Fetch ØR

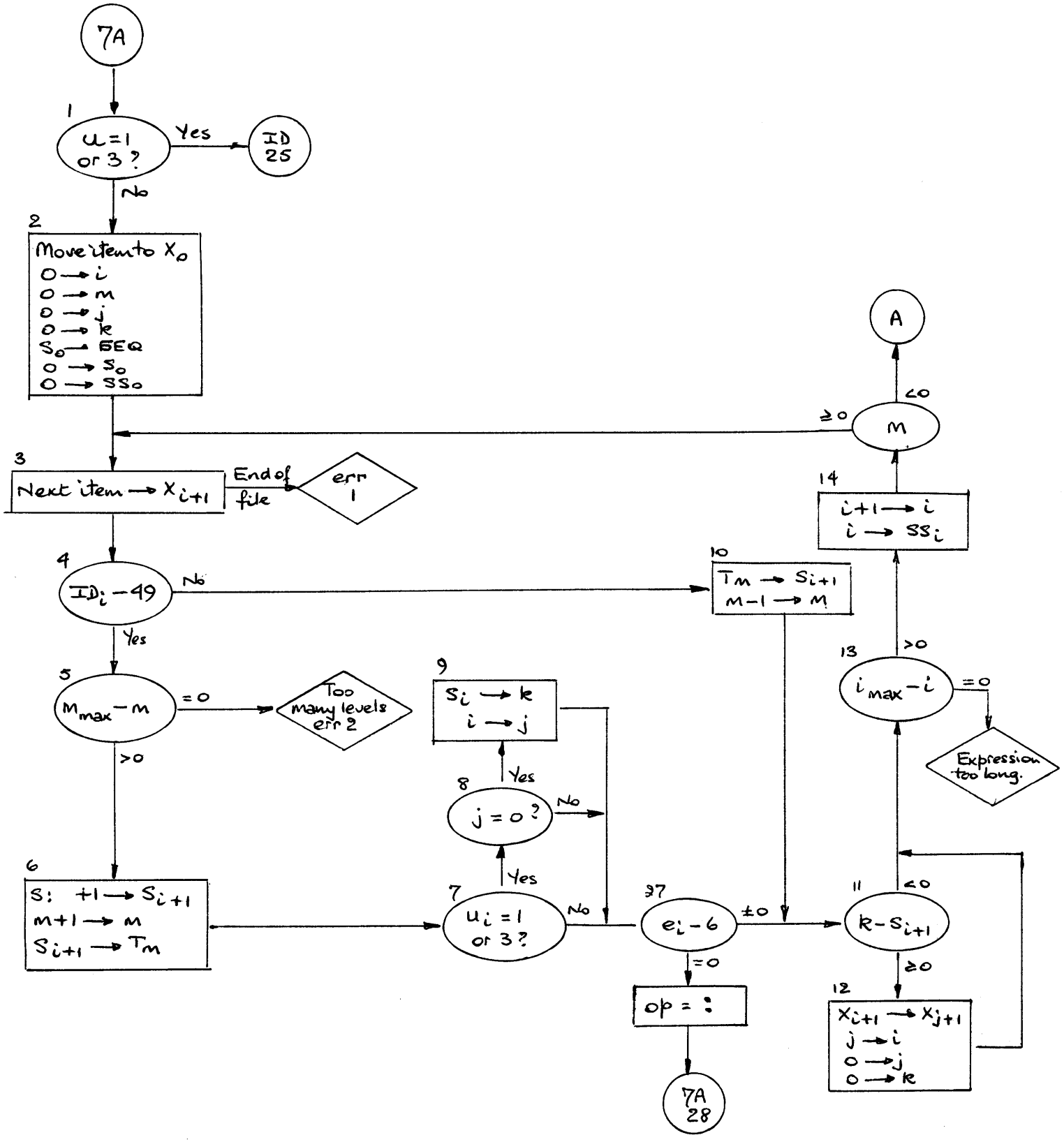


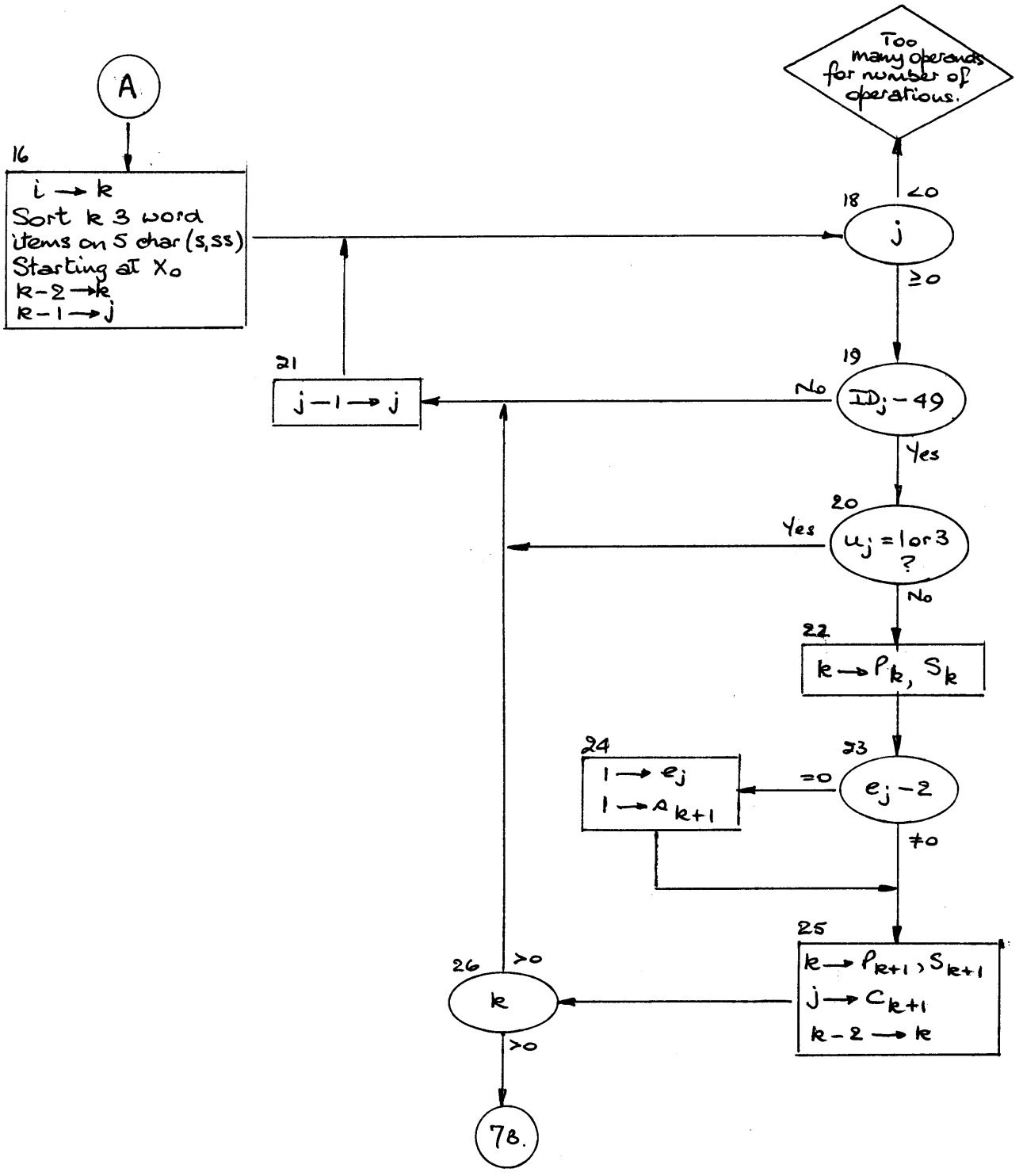

## Expand.

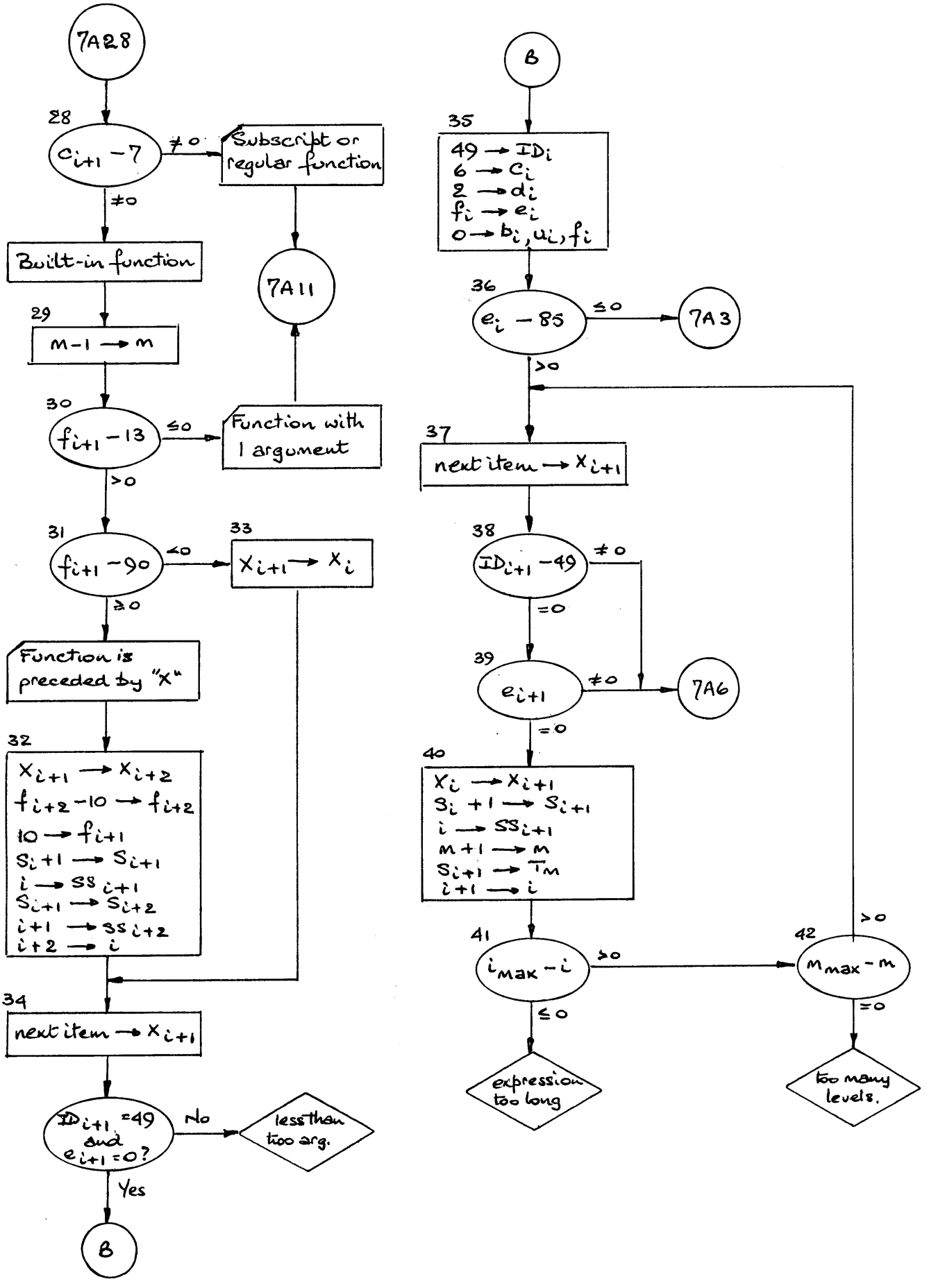


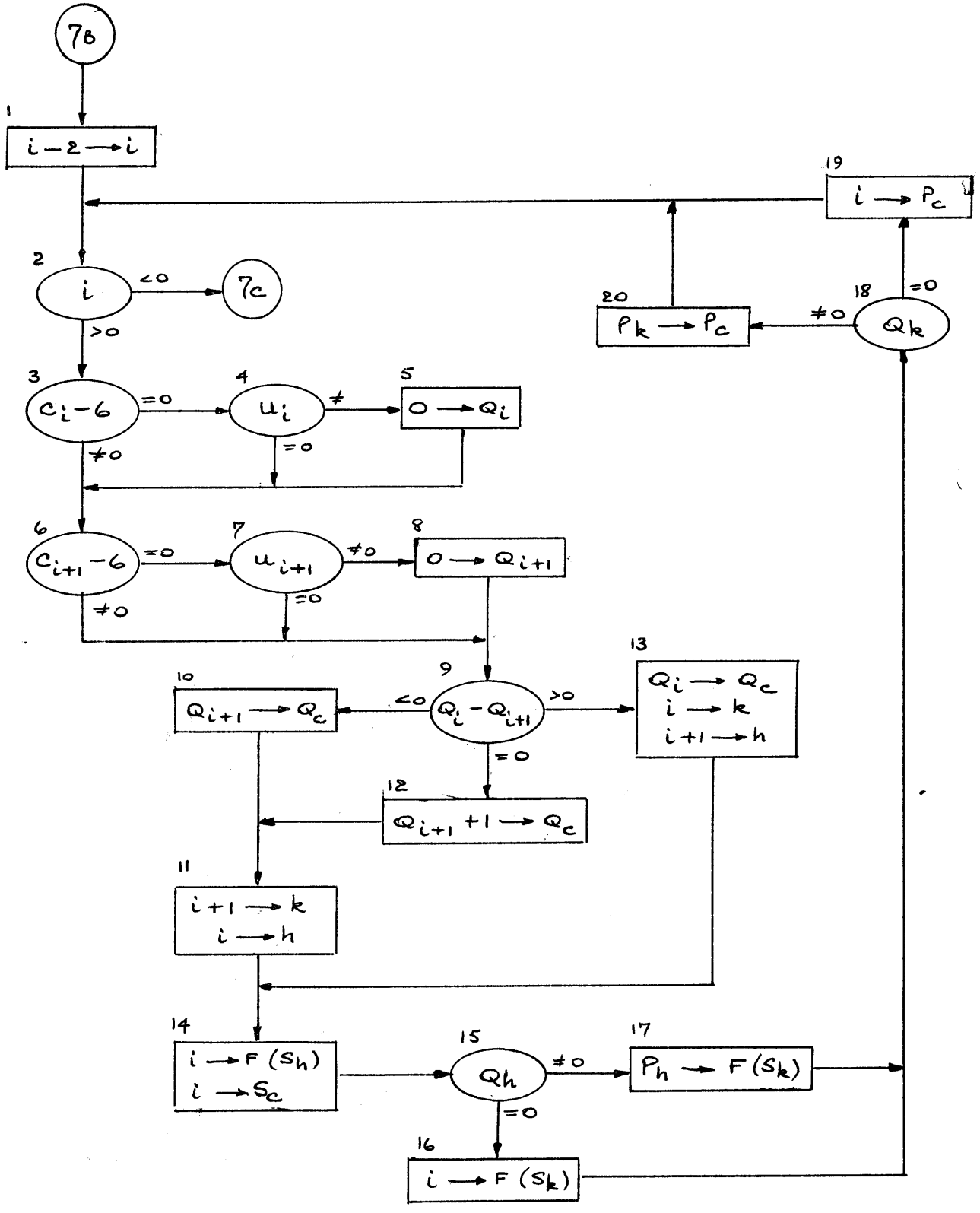

,
Arithmetic?
No
Yes
P2=,?
No
C2=0 or 6?
Yes
No
[s P2 tμ]
μ+1→μ
Yes
P2→STλ
λ+1→λ
P2=,?
No
Yes
C2=0 or 6?
No
[F P2]
Yes
[S P2]
C3=0 or 6?
Yes
[s P3 tμ]
μ+1→μ
C3=0 or 6?
No
F P3
Yes
P2→STλ
λ+1→λ
S P3

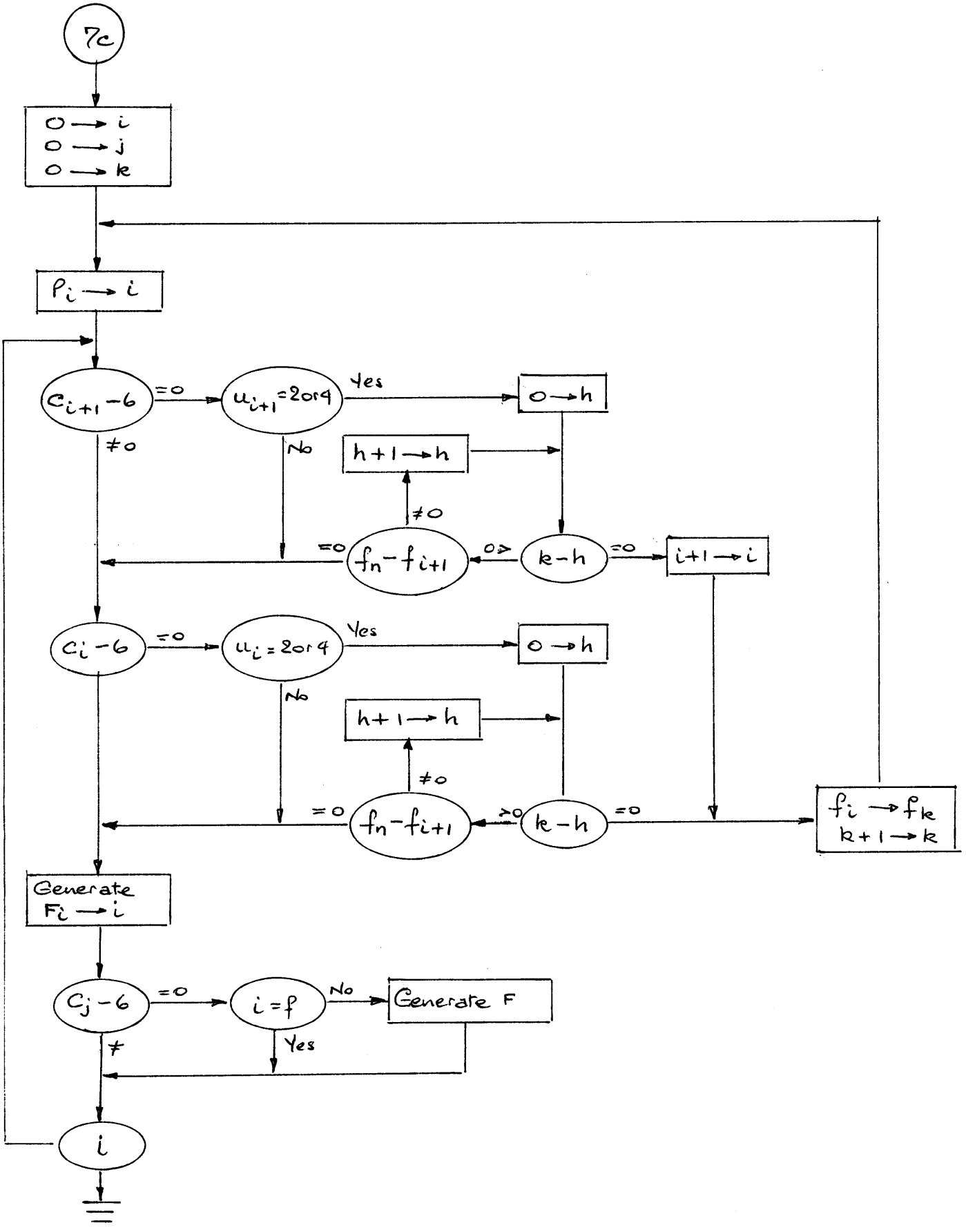

:
[TB #0 P2]
A2=5?
No
Yes
λ-ct→λ
0→w
w+1→w
#1→P1
#1→P1
Yes
w=ct
No
[SK]
P2=t
Yes
μ-1→μ
No


7A
1
u=1 or 3?
Yes
ID 25
No
2
Move item to $X_0$
$0 \rightarrow i$
$0 \rightarrow m$
$0 \rightarrow j$
$0 \rightarrow k$
$S_0 \rightarrow$ BEQ
$0 \rightarrow S_0$
$0 \rightarrow SS_0$
3
Next item $\rightarrow X_{i+1}$
End of file
err 1
4
$ID_i - 49$
No
Yes
5
$M_{max} - m$
=0
Too many levels err 2
>0
6
S: +1 $\rightarrow S_{i+1}$
$m+1 \rightarrow m$
$S_{i+1} \rightarrow T_m$
7
$u_i = 1$ or 3?
No
Yes
8
j = 0?
No
Yes
9
$S_i \rightarrow k$
$i \rightarrow j$
10
$T_m \rightarrow S_{i+1}$
$m-1 \rightarrow m$
11
$k - S_{i+1}$
<0
≥0
12
$X_{i+1} \rightarrow X_{j+1}$
$j \rightarrow i$
$0 \rightarrow j$
$0 \rightarrow k$
13
$i_{max} - i$
>0
=0
Expression too long.
14
$i+1 \rightarrow i$
$i \rightarrow SS_i$
M
<0
≥0
A
37
$e_i - 6$
±0
=0
op = :
7A 28

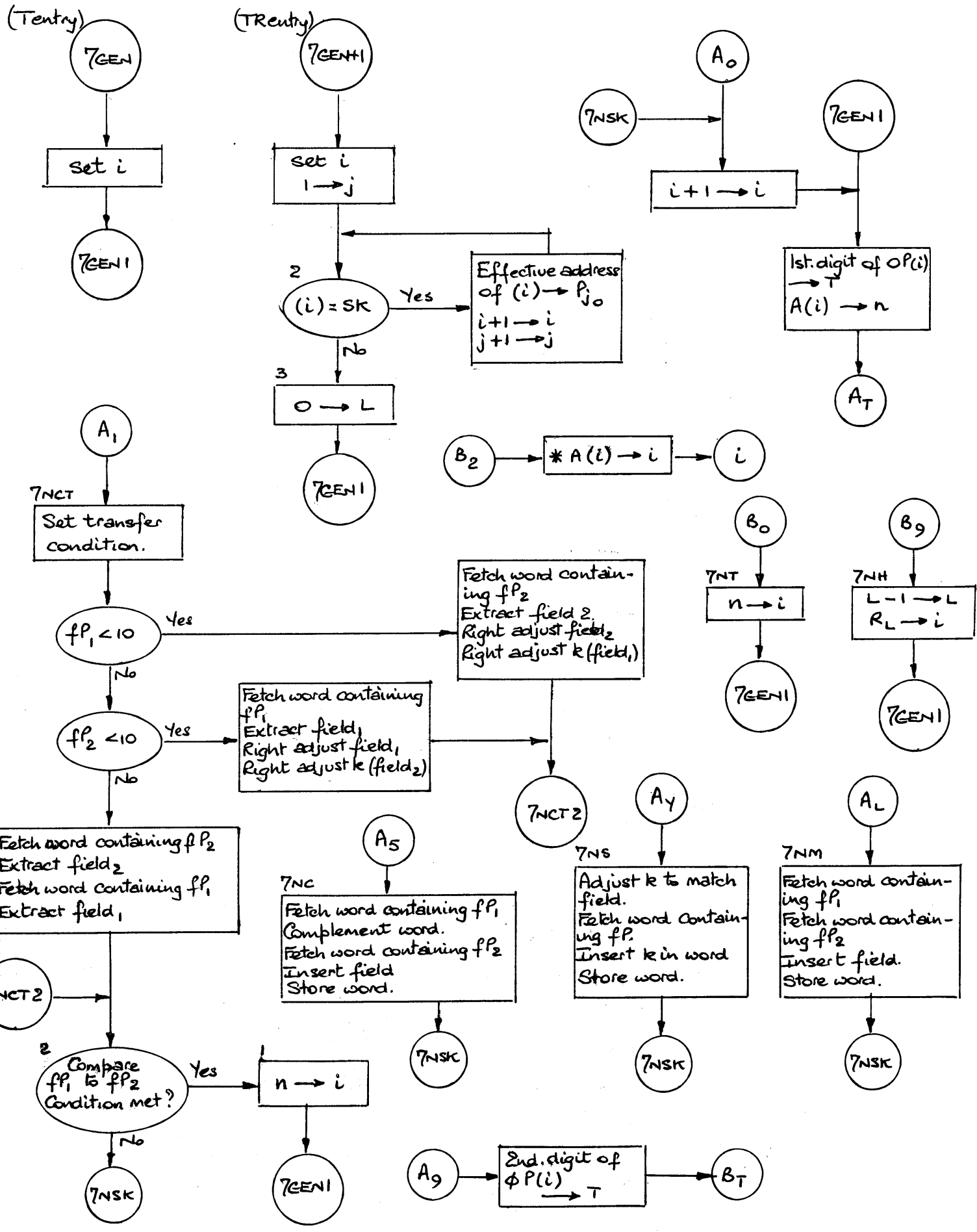

A
16
i → k
Sort k 3 word items on 5 char (5,55) Starting at X0
k-2 → k
k-1 → j
18
j
<0
≥0
Too many operands for number of operations.
19
IDj - 49
No
Yes
21
j-1 → j
20
uj = 1 or 3 ?
Yes
No
22
k → Pk, Sk
24
1 → ej
1 → Ak+1
23
ej - 2
=0
≠0
25
k → Pk+1, Sk+1
j → Ck+1
k-2 → k
26
k
>0
>0
78.


7A28
28
$c_{i+1} - 7$
≠0
Subscript or regular function
≠0
Built-in function
7A11
29
m-1 → m
30
$f_{i+1} - 13$
≤0
Function with 1 argument
>0
31
$f_{i+1} - 90$
<0
33
$X_{i+1} \rightarrow X_i$
≥0
Function is preceded by "X"
32
$X_{i+1} \rightarrow X_{i+2}$
$f_{i+2} - 10 \rightarrow f_{i+2}$
10 → $f_{i+1}$
$S_i + 1 \rightarrow S_{i+1}$
$i \rightarrow SS_{i+1}$
$S_{i+1} \rightarrow S_{i+2}$
$i+1 \rightarrow SS_{i+2}$
$i+2 \rightarrow i$
34
next item → $X_{i+1}$
$ID_{i+1}$ =49 and $e_{i+1}$ =0?
No
less than two arg.
Yes
B
B
35
49 → $ID_i$
6 → $c_i$
2 → $d_i$
$f_i \rightarrow e_i$
0 → $b_i, u_i, f_i$
36
$e_i - 85$
≤0
7A3
>0
37
next item → $X_{i+1}$
38
$ID_{i+1} - 49$
≠0
=0
39
$e_{i+1}$
≠0
7A6
=0
40
$X_i \rightarrow X_{i+1}$
$S_i + 1 \rightarrow S_{i+1}$
$i \rightarrow SS_{i+1}$
m+1 → m
$S_{i+1} \rightarrow T_m$
$i+1 \rightarrow i$
41
$i_{max} - i$
≥0
≤0
expression too long
42
$m_{max} - m$
>0
=0
too many levels.

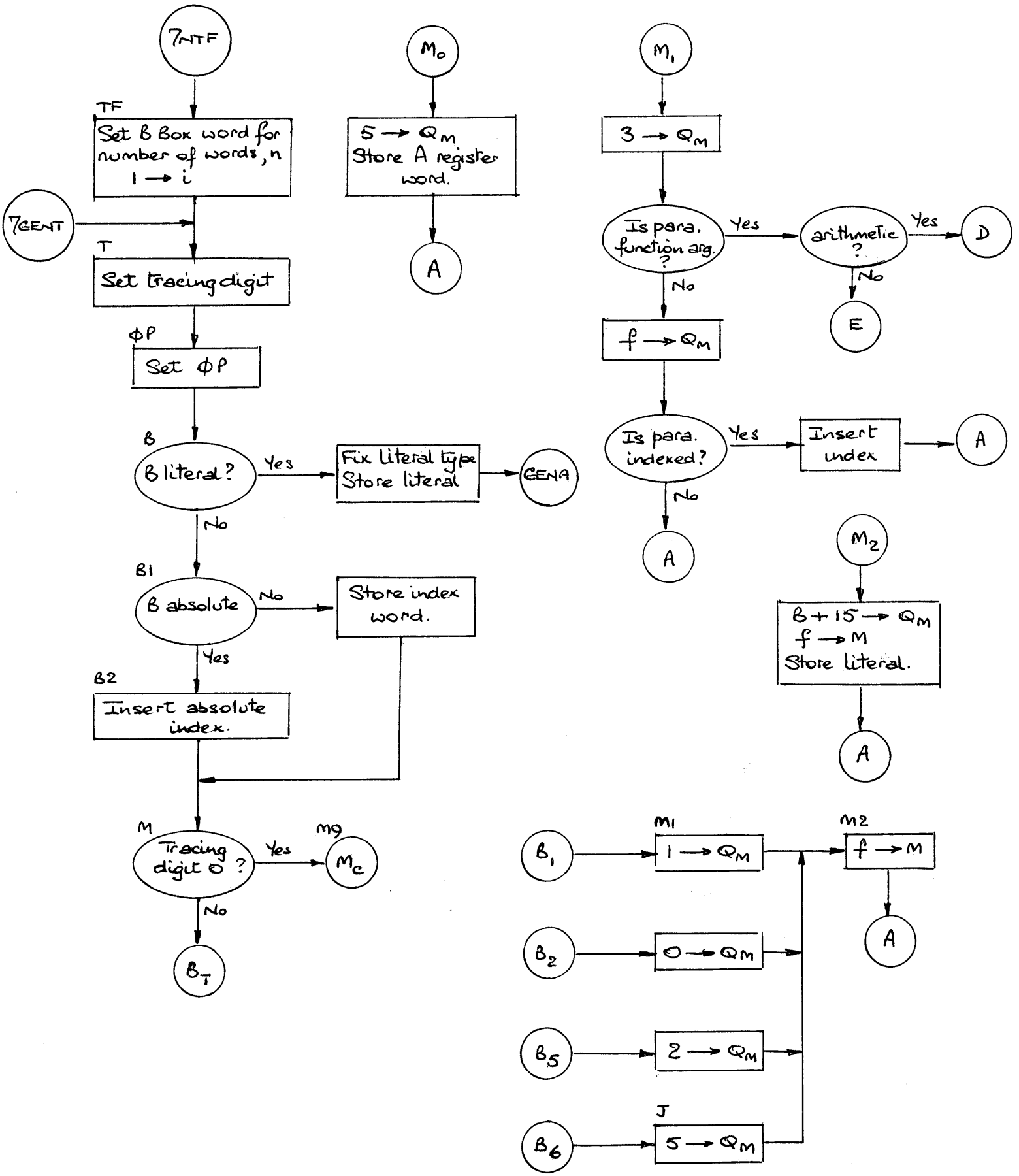

7b
1
i − 2 → i
2
i
< 0
7c
> 0
3
$c_i - 6$
= 0
4
$u_i$
≠
5
0 → $Q_i$
≠ 0
= 0
6
$c_{i+1} - 6$
= 0
7
$u_{i+1}$
≠ 0
8
0 → $Q_{i+1}$
≠ 0
= 0
9
$Q_i - Q_{i+1}$
< 0
10
$Q_{i+1} \rightarrow Q_c$
> 0
13
$Q_i \rightarrow Q_c$
i → k
i+1 → h
= 0
12
$Q_{i+1} + 1 \rightarrow Q_c$
11
i+1 → k
i → h
14
i → F ($S_h$)
i → $S_c$
15
$Q_h$
≠ 0
17
$P_h$ → F ($S_k$)
= 0
16
i → F ($S_k$)
19
i → $P_c$
20
$P_k \rightarrow P_c$
18
$Q_k$
≠ 0
= 0


7c
0 → i
0 → j
0 → k
$p_i$ → i
$c_{i+1}$ − 6
=0
$u_{i+1}$ = 2 or 4
Yes
0 → h
≠0
No
h + 1 → h
≠0
=0
$f_n - f_{i+1}$
0>
k − h
=0
i+1 → i
$c_i$ − 6
=0
$u_i$ = 2 or 4
Yes
0 → h
No
h + 1 → h
≠0
=0
$f_n - f_{i+1}$
≥0
k − h
=0
$f_i$ → $f_k$
k + 1 → k
Generate
$F_i$ → i
$c_j$ − 6
=0
i = f
No
Generate F
≠
Yes
i

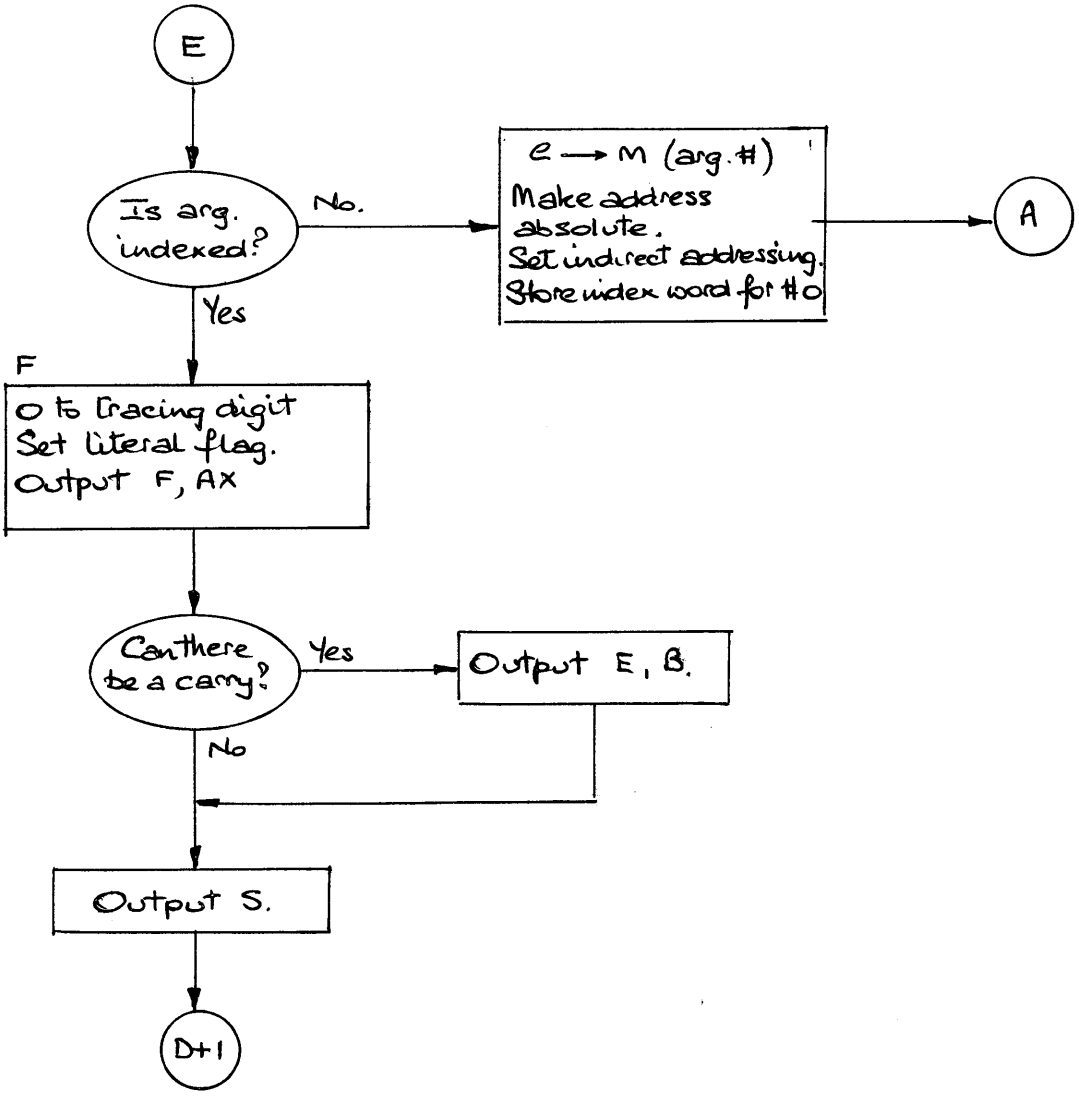
GENERATOR


(Tentry)
7GEN
Set i
7GEN1
(TRentry)
7GEN+1
Set i
1 → j
2
(i) = SK
Yes
Effective address of (i) → Pj0
i+1 → i
j+1 → j
No
3
0 → L
7GEN1
A0
7NSK
i+1 → i
7GEN1
1st. digit of OP(i) → T
A(i) → n
AT
A1
7NCT
Set transfer condition.
fP1 < 10
Yes
No
fP2 < 10
Yes
No
Fetch word containing fP2
Extract field2
Fetch word containing fP1
Extract field1
Fetch word containing fP1
Extract field1
Right adjust field1
Right adjust k (field2)
Fetch word containing fP2
Extract field2
Right adjust field2
Right adjust k (field1)
7NCT2
7NCT2
2
Compare fP1 to fP2
Condition met?
Yes
No
1
n → i
7GEN1
7NSK
A5
7NC
Fetch word containing fP1
Complement word.
Fetch word containing fP2
Insert field
Store word.
7NSK
AY
7NS
Adjust k to match field.
Fetch word containing fP1
Insert k in word
Store word.
7NSK
AL
7NM
Fetch word containing fP1
Fetch word containing fP2
Insert field.
Store word.
7NSK
B2
* A(i) → i
i
B0
7NT
n → i
7GEN1
B9
7NH
L-1 → L
RL → i
7GEN1
A9
End. digit of φP(i) → T
BT

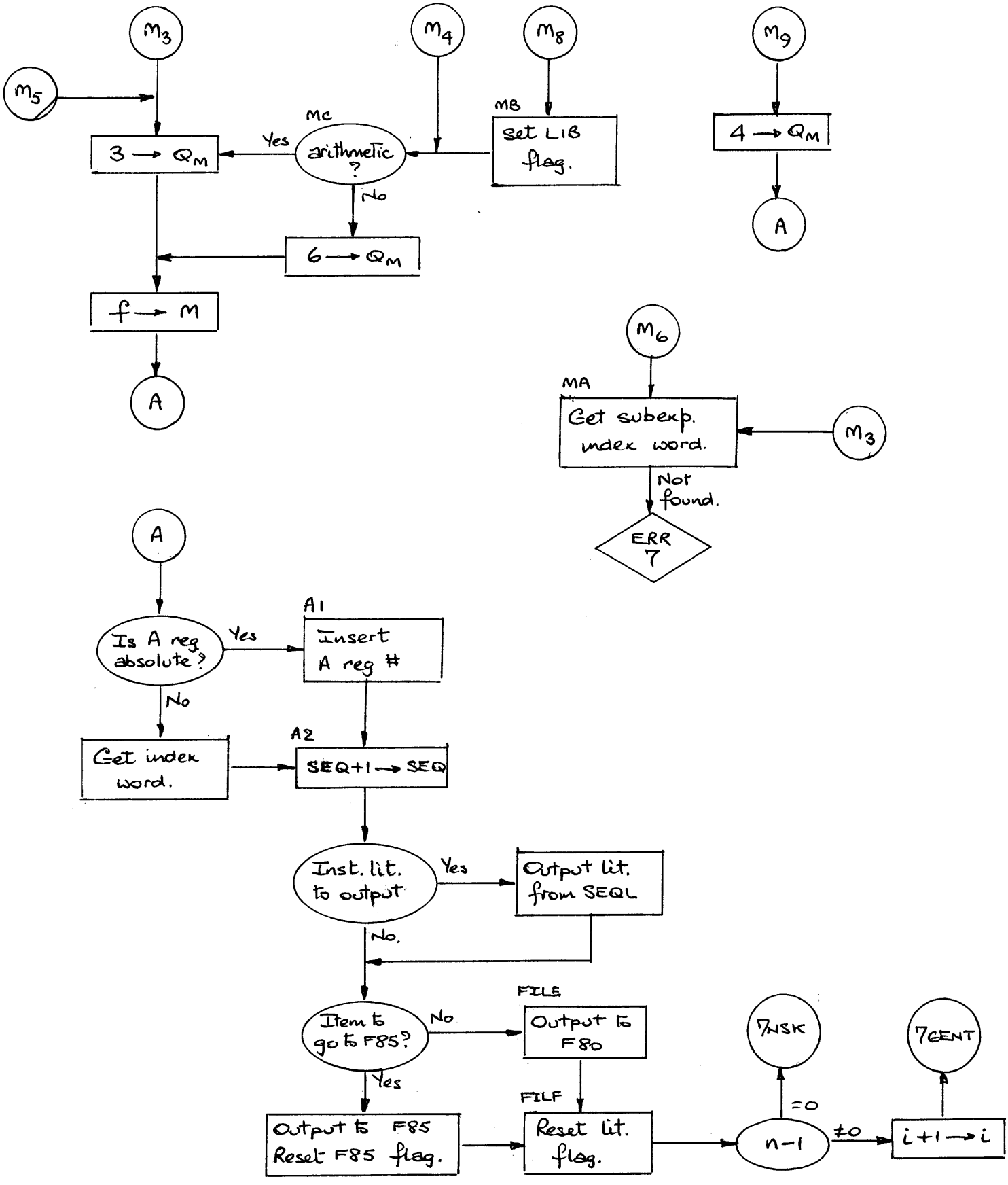
GENERATOR

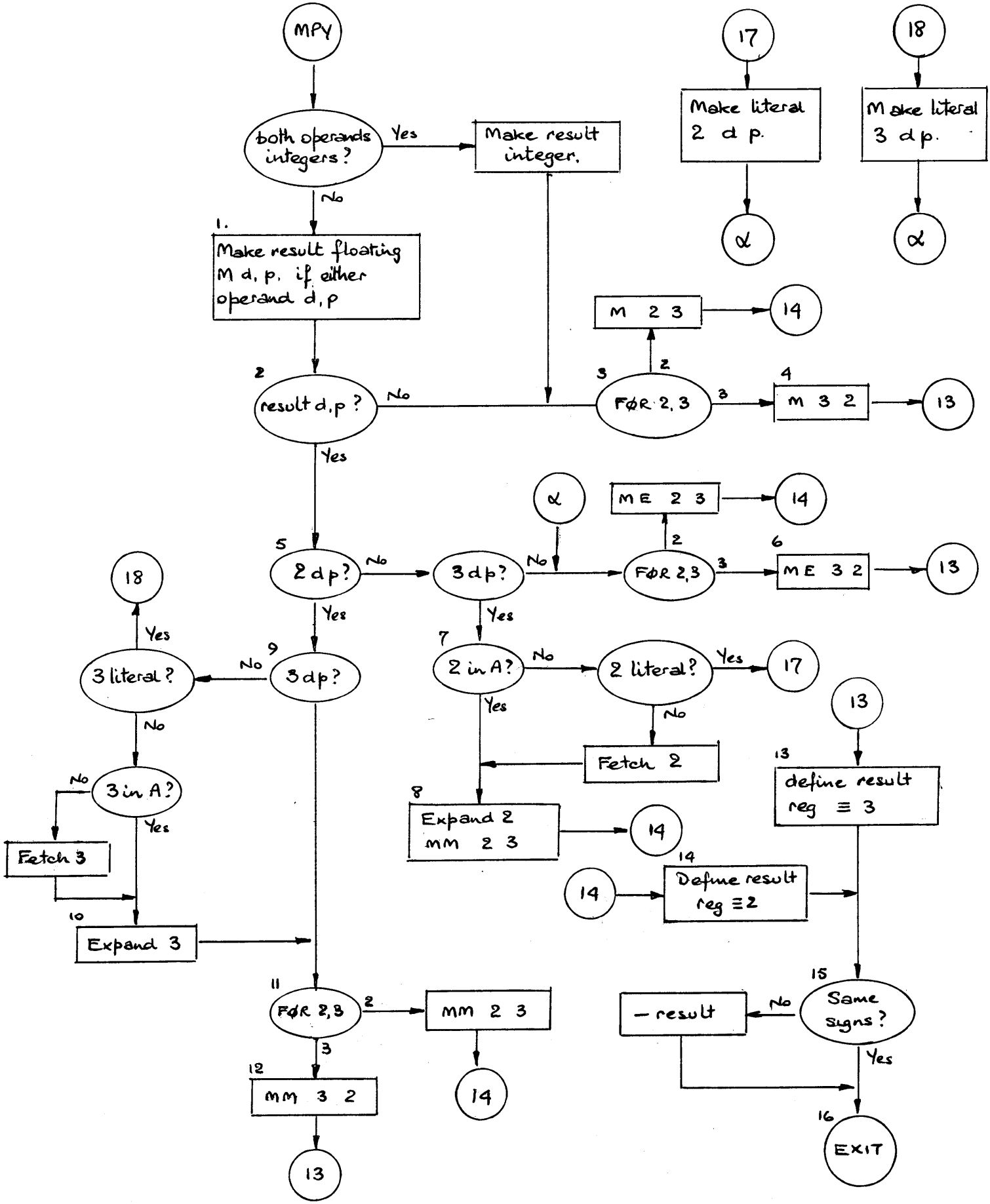

7NTF
TF
Set B Box word for number of words, n
1 → i
7GENT
T
Set tracing digit
ΦP
Set ΦP
B
B literal?
Yes
Fix literal type
Store literal
GENA
No
B1
B absolute
No
Store index word.
Yes
B2
Insert absolute index.
M
Tracing digit 0 ?
Yes
M9
Mc
No
BT
M0
5 → QM
Store A register word.
A
M1
3 → QM
Is para. function arg. ?
Yes
arithmetic ?
Yes
D
No
E
No
f → QM
Is para. indexed?
Yes
Insert index
A
No
A
M2
B+15 → QM
f → M
Store literal.
A
B1
M1
1 → QM
f → M
M2
A
B2
0 → QM
B5
2 → QM
B6
J
5 → QM

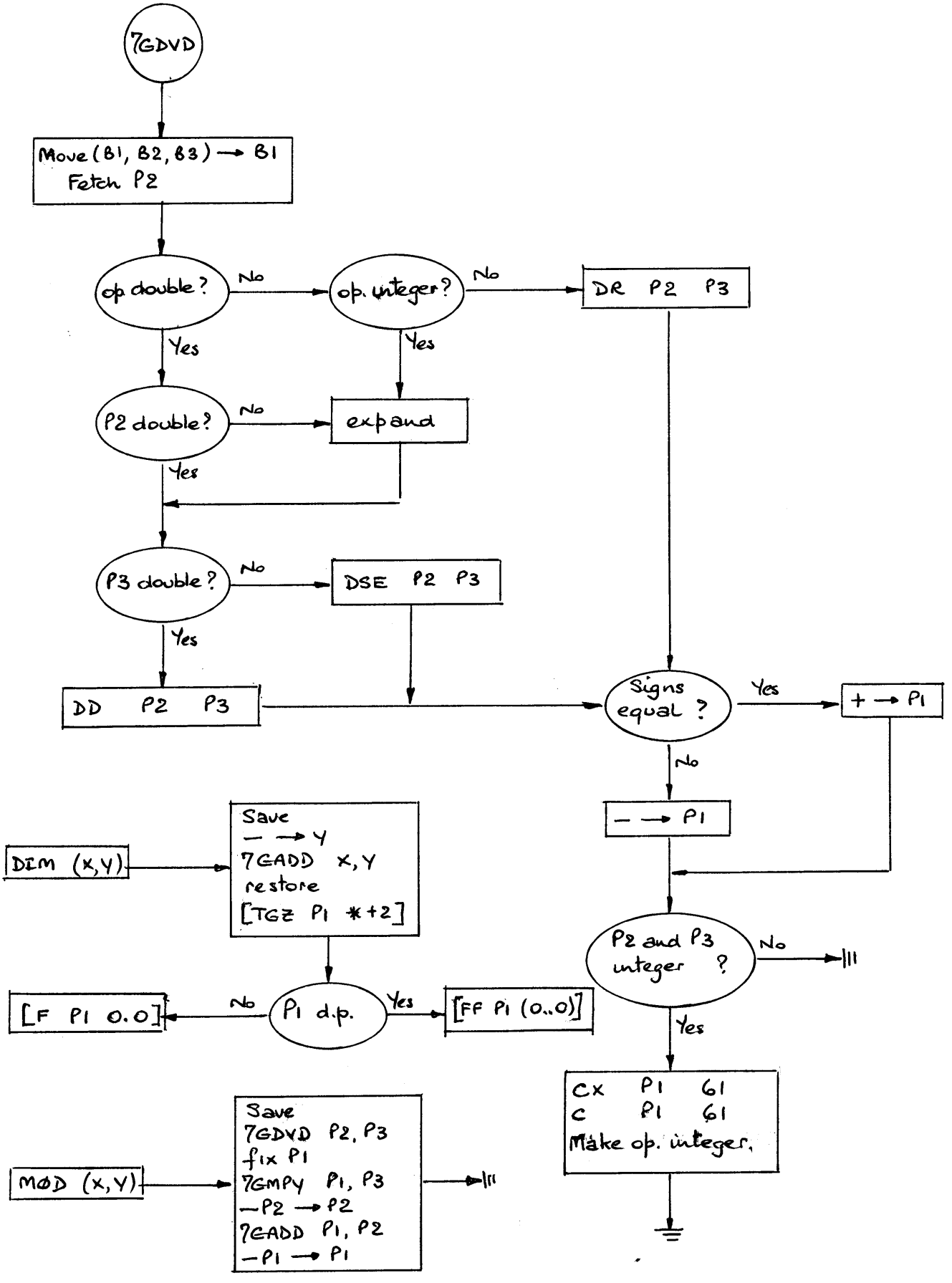
GENERATOR

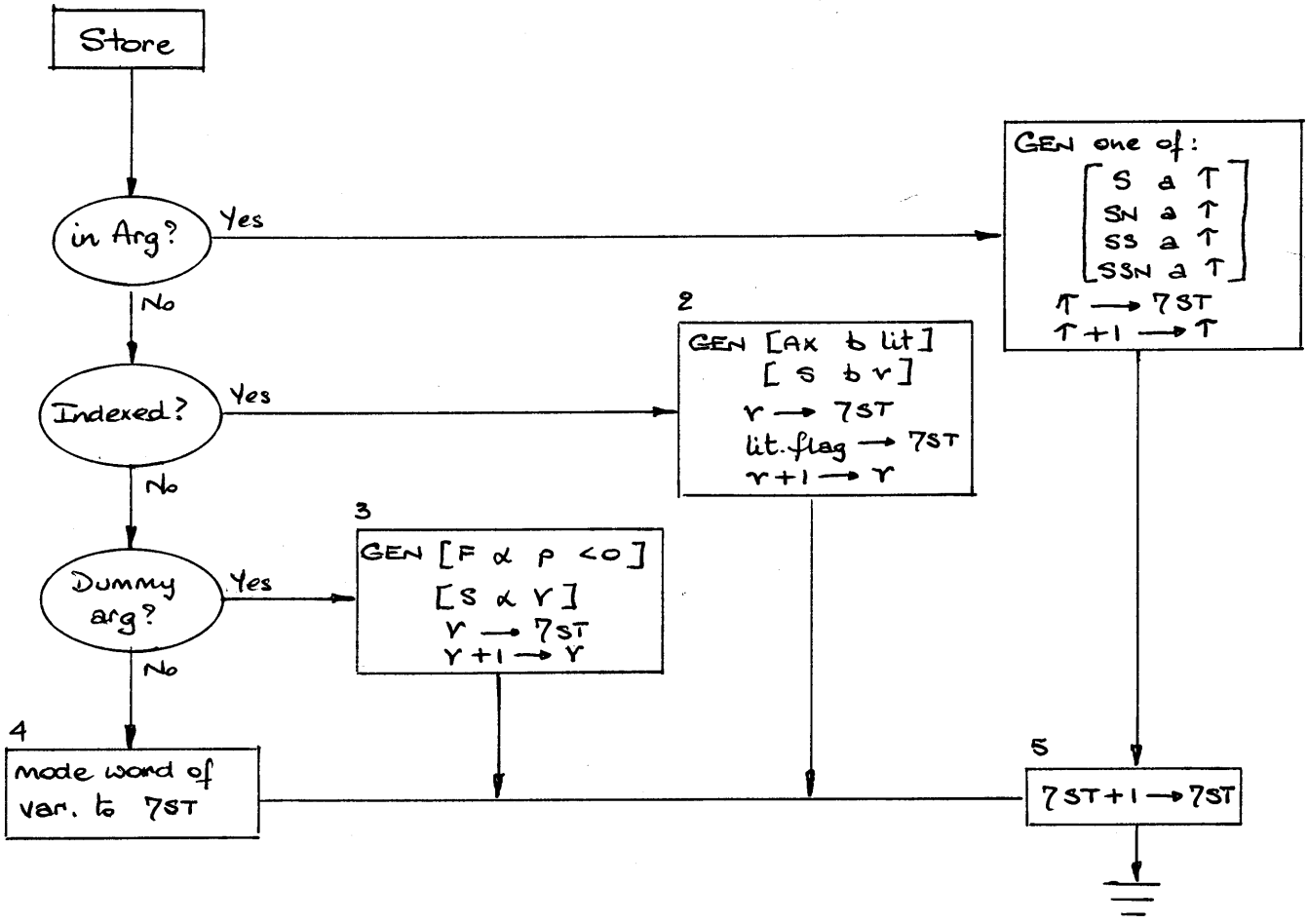

E
Is arg. indexed?
No.
e → M (arg. #)
Make address absolute.
Set indirect addressing.
Store index word for #0
A
Yes
F
0 to tracing digit
Set literal flag.
Output F, AX
Can there be a carry?
Yes
Output E, B.
No
Output S.
D+1

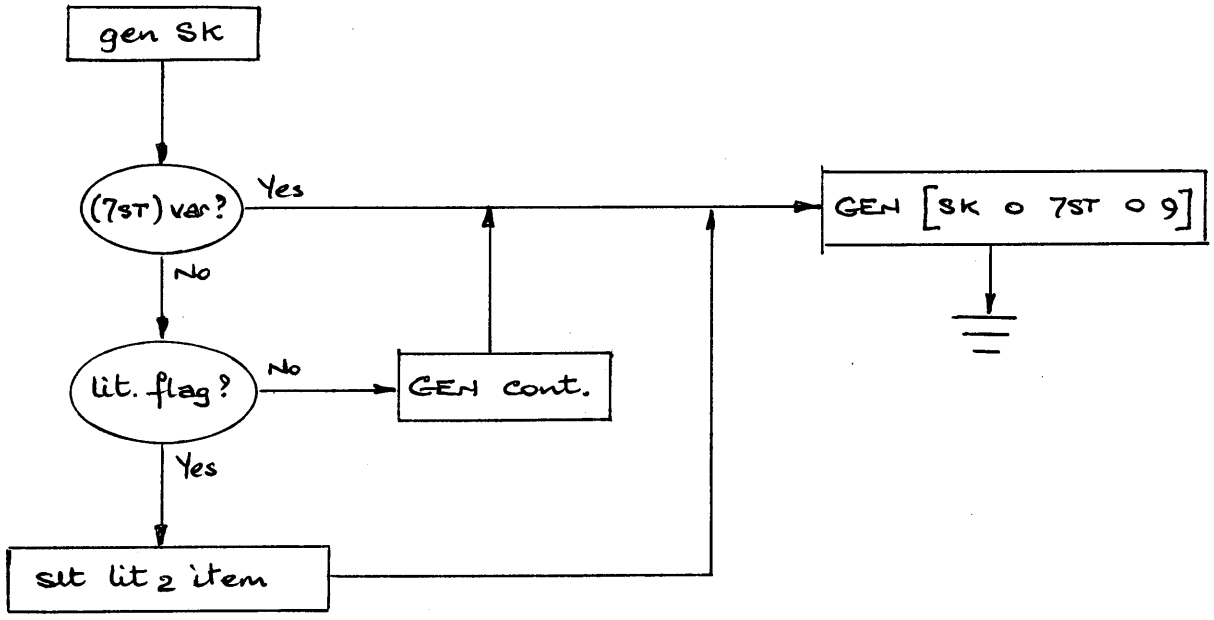
GENERATOR.

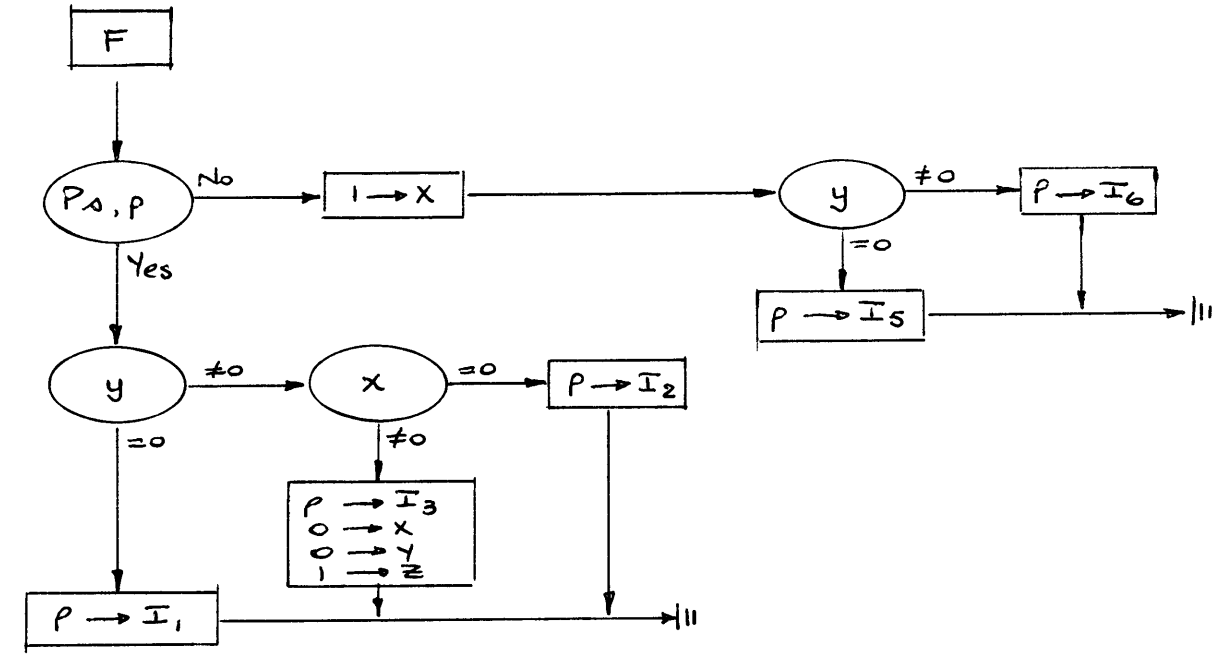

M3
M4
M8
M9
M5
MC
MB
Yes
arithmetic ?
No
3 → QM
set LIB flag.
4 → QM
6 → QM
A
f → M
A
M6
MA
Get subexp. index word.
M3
Not found.
ERR 7
A
A1
Is A reg. absolute?
Yes
Insert A reg #
No
A2
Get index word.
SEQ+1 → SEQ
Inst. lit. to output
Yes
Output lit. from SEQL
No.
FILE
Item to go to F85?
No
Output to F80
Yes
FILF
Output to F85 Reset F85 flag.
Reset lit. flag.
n-1
=0
≠0
i+1 → i
7NSK
7CENT

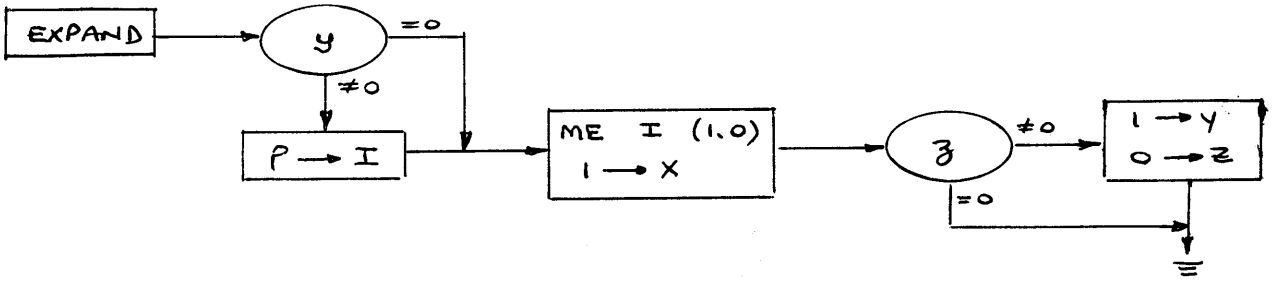

MPY
both operands integers?
Yes
Make result integer.
No
1.
Make result floating M d, p. if either operand d, p
2
result d,p ?
No
Yes
17
Make literal 2 d p.
18
Make literal 3 d p.
α
α
M 2 3
14
3
2
FØR 2, 3
3
4
M 3 2
13
α
ME 2 3
14
5
2 dp?
No
3 dp?
No
2
FØR 2,3
3
6
ME 3 2
13
Yes
Yes
18
Yes
9
No
3 literal?
3 dp?
7
2 in A?
No
2 literal?
Yes
17
No
Yes
No
No
3 in A?
Yes
Fetch 2
13
13
define result reg ≡ 3
8
Expand 2
MM 2 3
14
14
14
Define result reg ≡2
Fetch 3
10
Expand 3
11
FØR 2,3
2
MM 2 3
15
Same signs?
No
– result
Yes
3
12
MM 3 2
14
16
EXIT
13

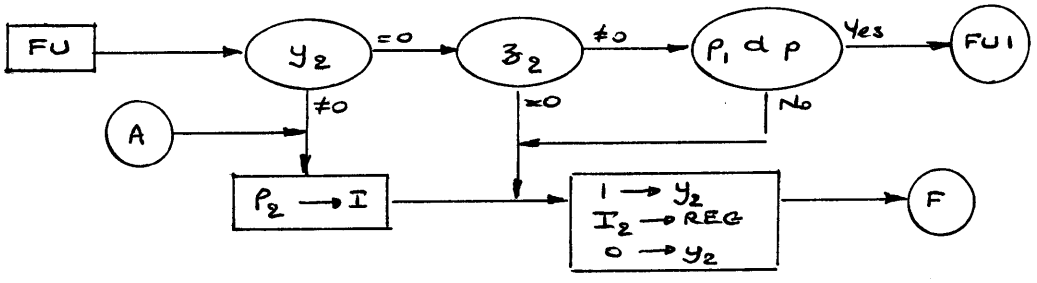

7GDVD
Move (B1, B2, B3) → B1
Fetch P2
op double?
No
op. integer?
No
DR P2 P3
Yes
Yes
P2 double?
No
expand
Yes
P3 double?
No
DSE P2 P3
Yes
DD P2 P3
Signs equal ?
Yes
+ → P1
No
— → P1
DIM (X,Y)
Save
— → Y
7GADD X, Y
restore
[TGZ P1 *+2]
P2 and P3 integer ?
No
Yes
[F P1 0.0]
No
P1 d.p.
Yes
[FF P1 (0..0)]
CX P1 61
C P1 61
Make op. integer.
MOD (X,Y)
Save
7GDVD P2, P3
fix P1
7GMPY P1, P3
—P2 → P2
7GADD P1, P2
—P1 → P1

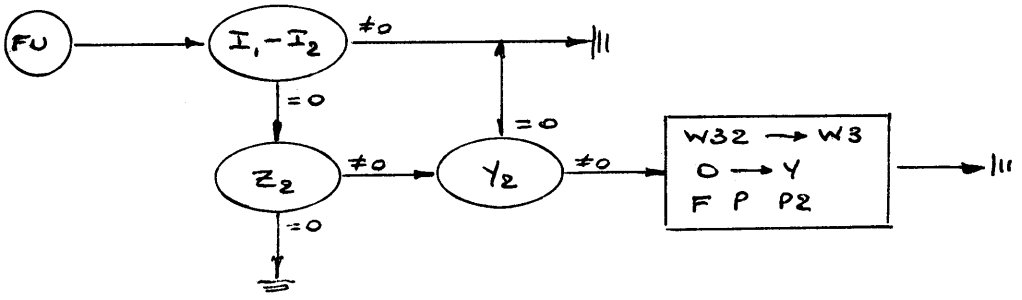

Store
in Arg?
Yes
No
GEN one of:
[S a T
SN a T
SS a T
SSN a T]
T → 7ST
T+1 → T
Indexed?
Yes
No
2
GEN [AK b lit]
[S b r]
r → 7ST
lit. flag → 7ST
r+1 → r
Dummy arg?
Yes
No
3
GEN [F α p <0]
[S α r]
r → 7ST
r+1 → r
4
mode word of var. to 7ST
5
7ST+1 → 7ST


gen SK
(7ST) var?
Yes
No
GEN [SK o 7ST o 9]
lit. flag?
No
Yes
GEN cont.
slt lit2 item


F
PA, P
No
1 → X
y
≠0
P → I6
=0
P → I5
Yes
y
≠0
x
=0
P → I2
=0
≠0
P → I3
0 → X
0 → Y
1 → Z
P → I1


EXPAND
y
=0
≠0
P → I
ME I (1,0)
1 → X
Z
≠0
1 → Y
0 → Z
=0


FU
y2
=0
z2
≠0
P1 d P
Yes
FU1
No
A
≠0
=0
P2 → I
1 → y2
I2 → REG
0 → y2
F


FU
I1 - I2
≠0
=0
=0
Z2
≠0
Y2
≠0
W32 → W3
0 → Y
F P P2

# PHASE VIII

Phase VIII is concerned with the assignment of physical fast registers to the instructions generated by Phase VII. In addition, movements to and from fast registers and memory cells used for temporary storage are interpolated into the generated program. The phase is divided into five subphases called VIIIa through VIIIe. The various functions of these subphases are described below.

## PHASE VIIIa - Backward Scan

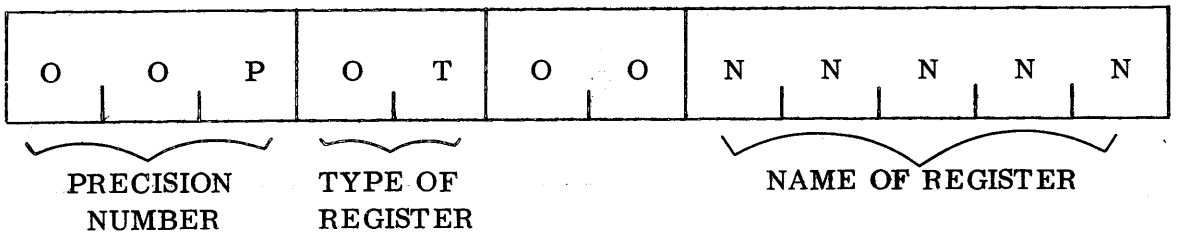
The output of Phase VII is in two files: File 8.0 and File 8.5. File 8.0 is generated in order on sequence number and contains the bulk of the generated code. File 8.5 is generated out of sort and contains relatively few items (to wit: extensible successors produce by T commands, library call items, and code used to initialize certain references to parameters in functions and subroutines). File 8.5 is sorted by the control program prior to the inception of Phase VIII.

Phase VIIIa reads Files 8.0 and 8.5 backwards and merges them into descending sequence. Library call items from 8.5 are detected by the merge and written directly to File 9.4 so that they do not enter into the remainder of Phase VIII processing.

The items from the merged file are now identified as representing instructions, continues, successors, instruction literal flags, or end of list flags. Successors and the two flag items are converted to their File 8.1 form and written out. The remainder of the Phase VIIIa processing is done on instruction and continue items.

Continue items serve to identify block boundries and as such initialize certain cells and tables which collect information about a block. Prior to this initialization, the information concerning the previous block is written in File 8.1. In particular, the tables (DR and DIS) containing the distances between usages of symbolic registers are emptied, the table (MAP) marking the preferential positions of indices (see VIIIb for a description of preferential positions) is cleared, and the cell containing the distance to the next TB (92) command is set to infinity (99999). In addition, the register SQ, which represents the position of each command in the block, is set to 99998.

Instructions have their symbolic registers processed. Each instruction contains a register reference word for its A field and perhaps (if the address mode is less than 10) also one for its B field. The register reference words have the form:




The precision number, P, identifies that portion of a symbolic register which is involved with this particular reference. (A symbolic register may comprise several physical fast registers.) The T field identifies the type of the register by:

| | | |
|---|---|---|
| T = | 0 | Absolute reference |
| | 1 | Arithmetic |
| | 2 | Index |
| | 3 | Common subexpression |
| | 4 | Preassigned reference |
| | 5 | Parameter reference |

The N field is a five-digit number which identifies within a given type the particular register involved. For T = 1, 2, or 3, N is a symbolic reference to be assigned by Phase VIII. For T = 0 or 4, N specifies directly which register is to be used. For T = 5, a memory cell is actually specified which is to be used to hold an expression that is a parameter to a function or subroutine.

Each instruction encountered is checked to see if it is one of the list TE, TG, TTE, or TTG. If so, certain anomalies in the value of P produced by Phase VII are corrected. Also, a check is made for the occurrence of a TB command so that steps may be taken to insure that registers #01 through #04 are free when the TB occurs.

For each register reference word, a subroutine (8AT) is entered which attaches to the instruction item the distances (recorded in the DIS table) to the next occurrence of the register and to the last occurrence. Also, the current value of SQ is attached and placed into the DIS table. Then SQ is decremented. With this information, Phase VIIIb can, during a forward scan, determine the relative undesirability of destroying the contents of a physical register by noting how soon it will be required in the computation.

PHASE VIIIb - Forward Scan

Phase VIIIb assigns a physical fast register for each symbolic fast register. The input is File 8.1, which, if read backwards, contains the executable part of the program in "execution order".

For each item in File 8.1, the suffix word is read and a branch is made on the identification to the proper routine to process that item. These individual routines read the remainder of the item.

Instructions:

The "B" reference is processed first if one is present. If the address mode is 05, then the "B" reference becomes the address of the instruction with no indexing. In this case, the "B" reference is tested to determine if it represents a single-precision common subexpression which has been previously put into temporary storage. If so, the address of the instruction is replaced by the corresponding temporary storage reference.

If the address mode is not 05 and is less than 10, then the proper assignment is made by the assign subroutine.

The instruction is checked to determine if it is an unindexed fetch command which immediately follows a store command which referred to the same symbolic register. If so, a flag ($REGSYM_0$) is set for use by the assign subroutine.

Some special instructions identified by operation codes 44, 49, and 54 are called forced-assign instructions. They refer to single, double, and quadruple precision registers respectively. Their use is to force VIIIb to make a particular fast register assignment for a given symbol rather than searching for an optimal one.

In this manner, for example, the result of the evaluation of an arithmetic statement function is forced to appear in register #01. If a forced assignment instruction is detected, the Pi switch is reset and the dictionary reference field of the instruction item is recorded.

The assign subroutine is entered with the parameters describing the "A" reference. Then the File 90 form of the instruction is constructed.

If the item immediately prior to this instruction was an instruction literal item, the implied continue item is written in File 90 and the instruction is written to File 94 once or twice with any non-zero sequence numbers specified by the instruction literal.

If the GOOD flag was set non-zero by the assign routine or the Pi switch is reset, or if for a F or FF instruction the "A" assignment equaled the "B" assignment, then the instruction is not output and control returns for the next File 8.1 item.

The instruction assigned is checked to determine if it is a S command; if not, it is written to File 9.0 and control returns. If so, and it is not indexed, its address is saved to compare with the next fetch command as described above. Also, checks are made to determine if the instruction represents an index definition item or a beginning of list item (address mode equal 40 and 50 respectively).

In the former case, an index definition item is formed and written in File 8.4 (see VIIIe description). The inverse level number is taken from the item and stored in LEVEL. Now if the dictionary reference field of the instruction item is not zero, it represents the sequence number of a continue to which the defining command transfers (either a BIT or an A-TLZ pair). This return point for a loop then also represents a point where the index is defined, so a second File 8.4 definition item is produced using the sequence number. In addition, it is

required that the index be available in the same register at the beginning of the loop as it was at the BIT so that, if the second definition is reconverted to a store command by VIIIe, then a meaningful quantity will be stored for the first entrance into the loop. Thus, a File 8.3 entry condition item (File 8.3 continue item) is required. Since the serial number (see discussion of treatment of continue items) of the reference continue is not known, a special item identification of 2 is required so that VIIId can determine the position of that continue in the object code by means of the sequence number alone.

In the latter case, the "A" reference is saved in LOCK which inhibits the assign routine from using that particular register until released. This is necessitated for the List= Expression items which contain explicit loops since, in this case, continue items will appear within the code generated for the list, and it is normally assumed that arithmetic quantities (in this case the expression) are not preserved across a continue.

In either case, control returns immediately for a new item so that the instruction does not appear in File 9.0.

Continues:

The continue item serves to force the output of information collected since the last continue and also to initialize the areas in which information will be collected starting with this continue. The commands required to save and restore any fast registers beyond #04 which are used by an arithmetic statement function are generated.

In particular, the code first reads the remainder of the continue item and checks to see if it does not introduce a new block (the identification is negative); if so, the File 9.0 form of the continue is produced and written out. Control returns immediately for a new item.

If the Tau Switch is reset, the previous continue introduced an arithmetic statement function. Subroutine X1 is entered to save and restore registers. Tau is then set.

The continue flag is then examined. The possible values are zero, one, and two, which respectively indicate a normal (programmer referenced statement name), erasable (beginning of SAL block), and arithmetic function continue.

If an arithmetic function is detected, the AFRST table is cleared and Tau reset. AFRST will record registers used.

For arithmetic function and normal continues, the File 9.0 item is written and the inverse level (if non-zero) is saved in LEVEL.

Now for normal and erasable continues, the table of available arithmetic temporary storage, ATSAT, cells is reset and the Epsilon switch tested. If set, this is the first continue encountered. The ENCT, EXCT, and HIST tables are cleared and Epsilon reset.

If reset, the File 8.2 continue item formed by the last continue is written out, the NI counter advanced, and a File 8.3 entry condition item is written for each non-zero entry in ENCT. Again ENCT, EXCT, and HIST are cleared.

In any case, the next File 8.2 continue item is formed and saved, and, if this continue is erasable, register #00 through #04 are marked as having been used.

Control returns for a new item.

Successors:

Successor items serve to detail the flow of control from one block to another. They are processed as follows:

> The successor item is read in, the NI counter advanced, and a File 8.2 successor item is formed and written out. Then for each non-zero entry in EXCT, a File 8.3 exit condition item is written. Control is returned for a new item.

Index Map:

The index map contains a mark for each physical register preferred for symbolic index quantities. These marks are used to initialize the register commitment table, RCT, which aids the assign subroutine to select the optimal assignment for a given symbolic register.

The index map is read in and its digits are stored one to the word in RCT. Also, the storage history table, SHT, is marked as empty, and the register usage table, RUT, is cleared.

Instruction Literal:

The instruction literal item signals that the next instruction is to be used as a literal. The item is read in and saved and the IL switch is set on.

TB Distance:

The TB distance item specifies the distance to the next TB command. This information is used to determine if, were a given symbolic register assigned in the range #00 through #04, it would have to be moved before its last usage due to an interviewing subroutine call.

The TB distance item is read in and its value saved.

End of List:

The end of list item resets the LOCK and LIST cells. The use of these quantities is explained in the paragraphs entitled "instructions".

The Assign Subroutine:

The assign subroutine determines which physical fast register to attach to a given symbolic register. The parameters given the subroutine are the register reference word, the position and number of the distances recorded by VIIIa and the current SQ for this reference. These parameters are first listed to determine if the reference is absolute, preassigned, or to a locked register. If so, the appropriate action is taken and the subroutine exited immediately. If not, the main body of the routine is entered.

The assign subroutine selects a physical register by examining each of the registers as a possible assignment. Several numbers are calculated for each possible assignment and are compared with the best assignment thus far found. If better, this assignment is substituted for the "best so far" and the process continues. The criteria used, in order of significance, are:

1. A penalty count representing roughly the number of additional instructions introduced into the code by this assignment.

2. The inverse sum of distances to next usages of any quantities already is the register.

3. The number of "index committed" registers used by the assignment.

4. The number of "arithmetic committed" registers used by the assignment.

5. The number of previously unused registers used by the assignment.

6. The size of the group of free registers in which the assignment appears.

7. The number of the physical register assigned.

Two of these criteria vectors are compared element by element until an inequality is found; the assignment with the smaller element is considered best. The effect of a possible assignment is kept in a set of tables called the action tables. Actually, two sets are maintained--one for the current assignment and one for the previous best assignment. These tables are called T0 through T9 and contain the following information:

T0: The criteria vector described above.

T1: Any fetches which must be inserted into the program.

T2: Any stores which must be inserted into the program.

T3:<br>T4:<br>T5: New histories which must be inserted into the register usage table (see below).

T6: Entries in the register usage table which are to be converted to secondary histories (see below).

T7: Entries in the register usage table which are to be deleted.

T8: Not used.

T9: Used to initialize the other tables.

The register usage table, RUT, is a matrix containing seven entries for each available fast register. Entries in the RUT are divided into two classes: primary and secondary. A primary entry is one which represents a quantity residing in that particular register. A secondary entry is one which represents a future usage of a register. For example, consider the code:

```
F   α   X
ME  α   Y
```

When assigning $\alpha$ for the F command, an entry is made in two successive rows of the RUT. The first represents the register $\alpha$ containing an actual quantity and is primary. The second represents the register $\alpha + 1$ containing the second half of the product x * y. In this manner, assignments may be made in the light of the commitment of a register to a certain quantity even though that quantity does not yet reside in the register.

Attached to the RUT are two other matrices called DIST and ULTD which contain the distance and ultimate distance respectively associated with the RUT entry. DIST is the SQ of the next references to the quantity and ULTD is the SQ of the last such reference. A fundamental rule of formation of the RUT is that it contains no conflicts. Two entries, $E_1$ and $E_2$, conflict under the following conditions:

1. Both $E_1$ and $E_2$ primary.
   Always a conflict.

2. $E_1$ primary and $E_2$ secondary.
   A conflict provided that ULTD ($E_1$) > DIST ($E_2$).

3. Both $E_1$ and $E_2$ secondary.
   A conflict provided that DIST ($B_1$) < DIST ($E_2$) < $ULTDCE_1$ or
   DIST ($E_2$) < DIST ($E_1$) < ULTD ($E_2$).

In brief, a conflict exists when a register is simultaneously committed to two quantities, either now or in the future.

The heart of the assign routine consists of three nested loops: the I, J, and K loops. The variable I specifies which of the physical registers is under consideration as the assignment for the first segment of the symbolic register being assigned; the variable J specifies which segment of the symbolic register is being treated; and the variable K, which of the seven entries in the RUT is currently being tested for conflicts. Note that the J loop is usually trivial, since symbolic registers are usually single precision.

When an entry in the RUT is examined for conflict with the symbolic register being assigned, appropriate information is filed in the action tables under the assumption that this is the physical register to be assigned. For example, if the entry contains a common subexpression which has not been previously sent to temporary storage and the current register being assigned is a secondary arithmetic quantity, then an entry is made in T2 to specify that the common subexpression be stored and an entry in T6 to specify the primary RUT entry be made secondary. If considering the RUT entry as secondary still produces a conflict, an additional entry is made in T7 to specify that the RUT entry be erased completely. If the common rule expression were a double-precision register where other part was secondary, then that secondary usage is meaningless if not associated with a near-by primary.

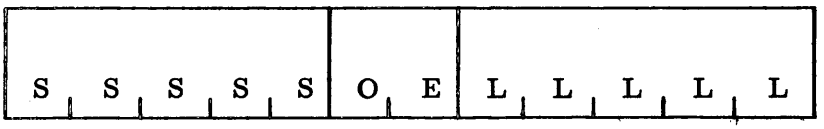
This example should serve to depict the sort of processing that occurs within the loops of the assign routine.

After all possible assignments have been considered, the action tables representing the optimal assignment are processed and the actions represented there are carried out; i.e., the required instructions are introduced into the object code and the RUT is updated. In addition, any empty positions in the entry condition table are updated. The exit condition table is always updated.

It should be noted that if it is specified that an index quantity be fetched, and the corresponding ENCT position is blank, the fetch is not actually produced. Phase VIIId will insure that the index quantity is present when the code is executed. If not blank, an index fetch request item is sent to File 8.4, and Phase VIIIe produces the actual fetch command.

PHASE VIIIc - Build Flow Tables

The duty of Phase VIIIc is to prepare a "flow-chart" of the problem being compiled. This "flow-chart" consists of two tables called UV and P. The P table consists of one-word entries of the following form:




An entry is made in the P table for each entry point (continue item) and each transfer (successor item) in the object code. Two possible orderings of the continues and sucessors are discernable: Entry Order and Exit Order. In Entry Order, each continue is followed by those successors which reference it. In Exit Order, each continue is followed by those successors which leave the block initiated by the continue.

The P table is formed in Entry Order. The Exit Order is implied by the L field. The L field contains the position, relative to the beginning of the P table, of the next entry in Exit Order. Thus, the P table may be processed in either order--stepping directly through or following the chain as specified by L. The S field is the major sequence number of the continue or successor item which produced the entry. The E field is the extensibility flag copied from the successor item. It is zero for continues.

The UV table is merely a list of those positions in the P table which contain continue entries. An additional entry is made in the UV table which references the first all in the P table not containing information.

An additional table called T is used to associate Exit Order with Entry Order. If an item is in position i in Exit Order and position j in Entry Order, then the M portion of $T_i$ contains the value j. The T table also contains in position j the inverse level (see Phase V description) of the $j^{th}$ item in Entry Order.

The input to Phase VIIIc is File 8.2. This file is arranged in Entry Order by sorting on the first word and read into memory in its entirety. For each item read, the serial field representing Exit Order is stored into the T table, the item rearranged so that it may be sorted to Exit Order, and the position in the item in Entry Order is attached. After the end of File 8.2 is reached, it is sorted to Exit Order and each item receives the value of i from the one following it. This forms the L field described above. The file is then returned to Entry Order and the P and UV tables are filed. All is now ready for Phase VIIId.

PHASE VIIId - Index Analysis

Phase VIIId insures that index quantities needed at the beginning of a block are actually present. When required, index fetch request items are manufactured and sent to File 8.4 for subsequent processing by Phase VIIIe. Two sources of information are combined to determine the fetches required: the "flow-chart" constructed by Phase VIIIc and the Entry and Exit condition items developed by Phase VIIIb and recovered in File 8.3.

File 8.3 is sorted first on physical register and within physical register into Entry Order. Each physical is treated separately, and the fetches required for it throughout the entire program are determined before progressing to the next register. Information is read from File 8.3 in groups on the physical register. Groups not containing indexes are ignored. It is stored into a precleared table in the position corresponding to its entry order.

The serial number (NI counter) is used to reference the T table and thus determine the proper positions. Since the serial is not known for special continued items, their sequence number is looked up in that part of the P table which corresponds to entries in the UV table. If the position so located is empty, the item is simply stored there. If it contains an arithmetic quantity, the special continue supercedes it. If, however, another index is found, that index must necessarily be fetched. A file 8.4 item is produced before the special continue is stored.

After a group has been obtained, an iterative process is performed. The iteration is continued until a maximum number of passes have occurred (as specified by CRNOOP in the communication region) or until no further information can be obtained. Then a final pass occurs which produces the actual fetch requests on the basis of the information obtained.

The iteration process determines whether or not an index quantity can be guaranteed to exist in the register when a block is entered. Such a guarantee is possible if all successors referring to a given continue are identical. If so, that continue is marked as containing the index. If not, and all the successors which reference a given continue contain the same index insofar as they contain anything, then that continue is unknown and no updating occurs. If, however, two successors contain different quantities, then the continue is marked as ambiguous. Whenever information is gained by either discovering than an index does indeed exist at a block entry or that the register is ambiguous, then that information is passed on to these successors which are themselves unknown until a successor is found. This "passing on" of information is accomplished by processing the items in Exit Order. Thus, the next pass may well yield information about other continues.

One refinement to the above process has been built in. If the successors entering a block disagree only by virtue of some of them being unknown, and those which are unknown are exits from the same block that they enter; i.e., represent simple loops, then it may be assumed that the common known value is then available at the continue.

When the iteration terminates, each known continue is compared with the successors referring it. If they agree, no fetch is required. If none agree or if some do, and any of those which do not are non-extensible, then a File 8.4 fetch request is produced which will cause the desired index to be fetched at the continue. If all of those successors which disagree are extensible, then the File 8.4 fetch request is produced to fetch the value at the successor. This scheme effects an excellent compromise between time and space efficiency in the object code produced.

PHASE VIIIe - Produce F and S Commands

Phase VIIIe combines the index definition items produced by Phase VIIIb and the index fetch items produced by both Phase VIIIb and Phase VIIId to produce fetch and store commands. The input is from File 8.4 and the output is to File 9.4. File 8.4 is previously sorted on three characters (six digits) so that all references to a given symbolic index register are brought together with the fetch request items preceding the definition items.

Phase VIIIe reads a group on six digits into the FETCH area. If the data obtained represent definition items, they are discarded immediately, since no fetch requests are present for that particular symbolic index register. Whenever a group of fetch requests is found, the corresponding group of index definition items is read into the STORE area. The two words of each of the items in STORE are **reversed** so that the sequence number of the definition point is on the left and the entire group is sorted into ascending sequence on the sequence number.

Now for each item in FETCH, the following operations occur:

1. The STORE area is scanned from the beginning until an entry is found whose sequence number is greater than that of the fetch request. Then the previous definition is flagged (by setting the three high-order digits of the symbolic index name to zero).

2. The level number of the definition so marked is compared to that of the fetch request. If they are not equal, the remainder of the STORE area is scanned until that definition item is found with the greatest sequence number such that the level number is equal to that of the fetch request. If such an item exists, it also is flagged as above.

3. A F command is formed from the fetch request item and written to File 9.4.

After all the entries in FETCH have been processed, the STORE area is once again scanned. For each entry which has been flagged, an S command is generated and written to File 9.4. Control then reverts to the beginning and the next symbolic index register is processed.

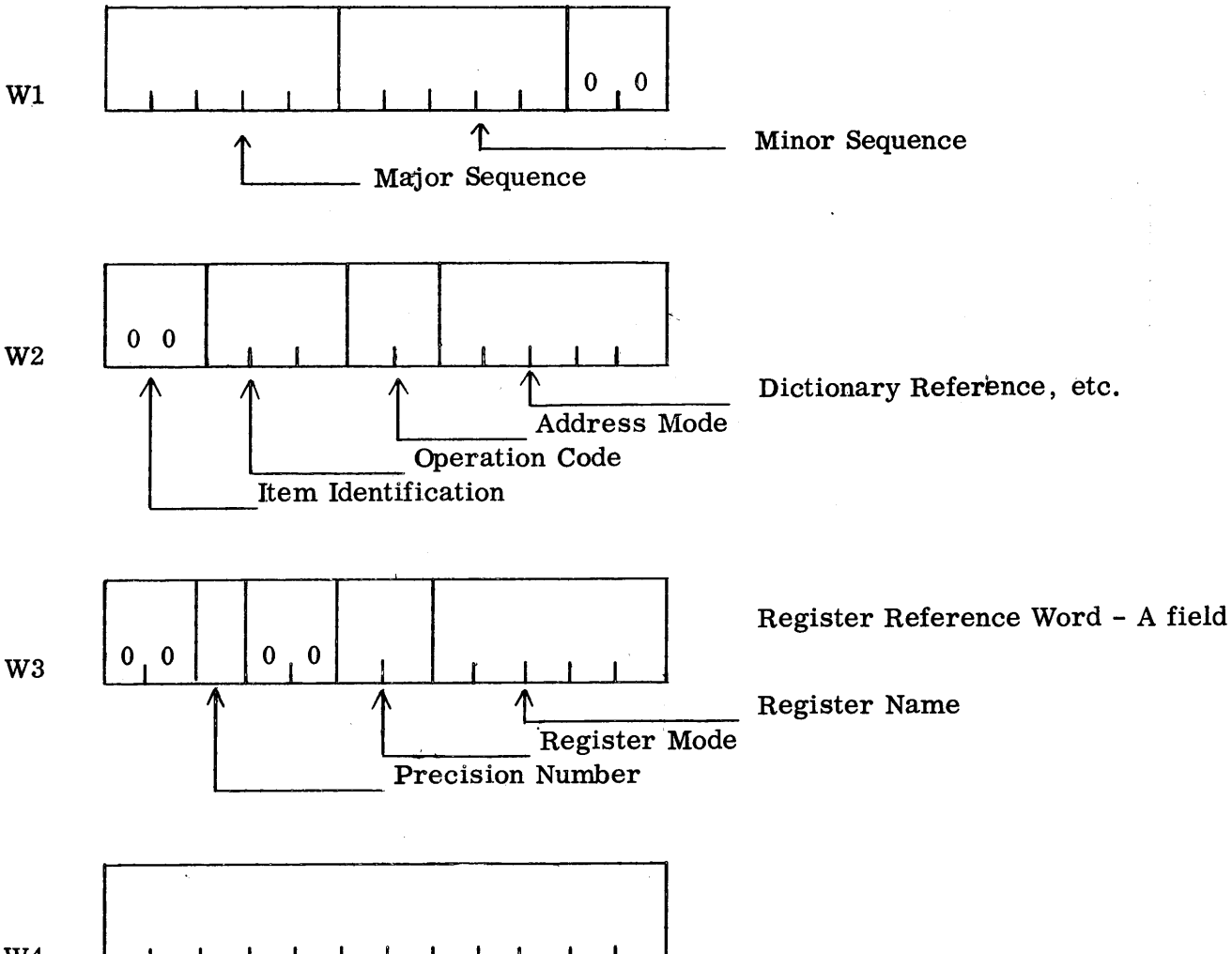
The F and S commands generated are given the sequence numbers supplied with the File 8.4 items. The A register is likewise obtained. The M field is marked as temporary storage. The particular temporary storage call is determined by a counter which is advanced by one for each group of fetch requests processed.

The algorithm described here guarantees that only those definitions are converted to S commands which are absolutely required for each F command to have a meaningful operand.

# PHASE VIII FILE FORMATS

## File 8.0 Items

Instruction item




If the address mode is less than 10, the W4 is the register reference word for the B field and has the same format as W3. If the address mode is 10 or greater, W4 contains the specific literal referenced and is formated appropriately for that literal.

The Address Modes are

00 Absolute Address

01 Self relative forward (HERE +)

02 Self relative backward (HERE -)

03 Symbolic

04 Temporary storage reference

05 Fast register reference

06 Symbol with same symbol as marker

07 Symbol with special marker

10
11
12
13
14
15 } Literals - See Phase IX description
16
17
18
19
20
21

40 Register definition mode

50 Beginning of List mode

The Register Modes are

00 Absolute register reference

01 Index register

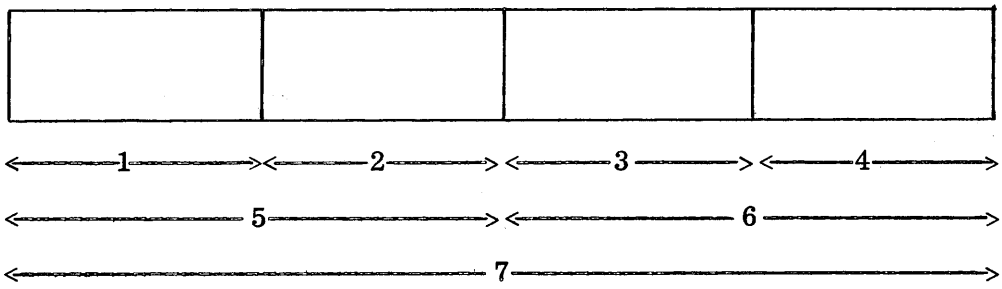
02 Arithmetic register

03 Common subexpression

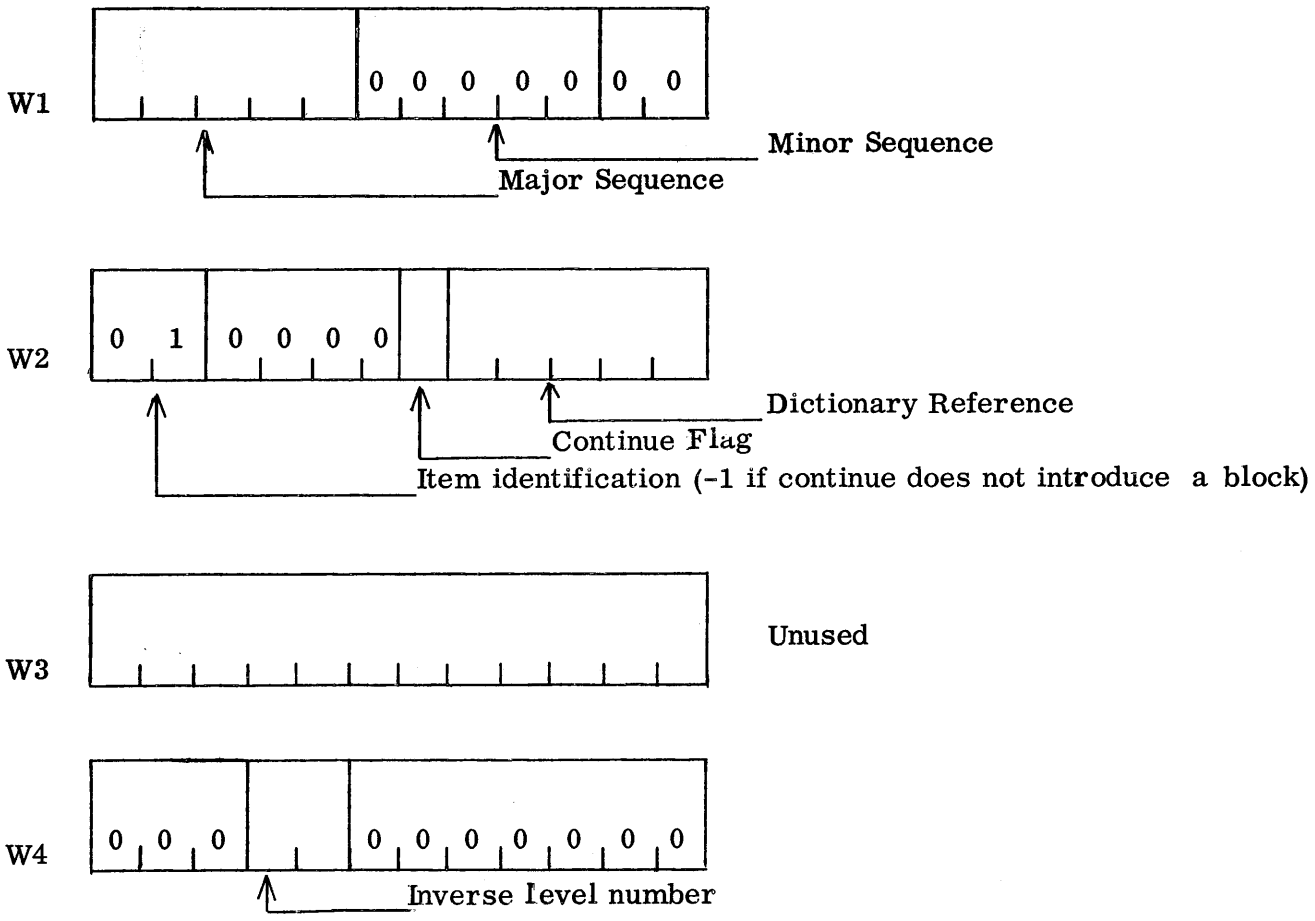
04 Pre-assigned reference (with #)

05 Parameter storage reference

The precision number describes which of the segments of a symbolic fast register is active for this particular reference. The four possible segments are arranged as follows:




Continue Item

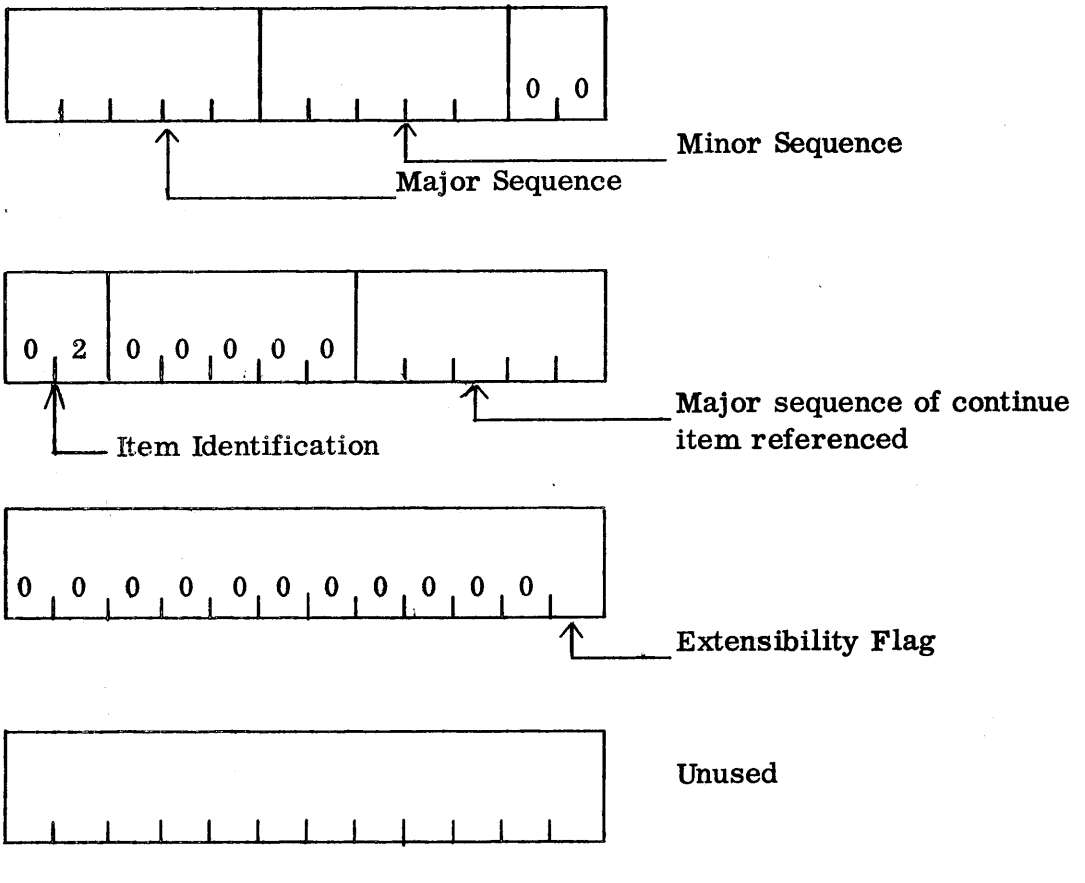



The Continue flags are 0 - Normal entry point

1 - Introduces Authentic Statement Function

2 - Introduces SAL block

Successor Item

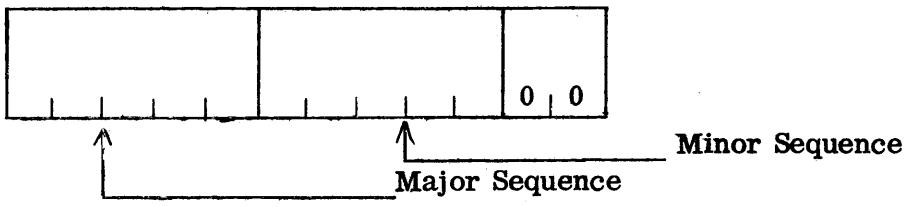



The Extensibility flags are - 0 - Non-extensible
1 - Extensible

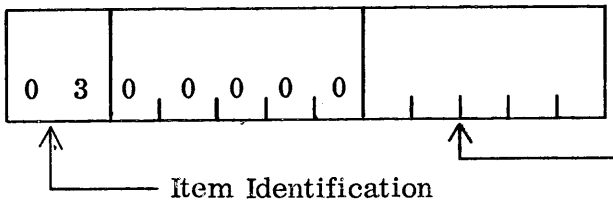
A successor is extensible implies that Phase VIII may insert instructions at the successor without disturbing the registers used in the code that immediately follows and without invalidating any self-relative references.

Instruction Literal Item

W1

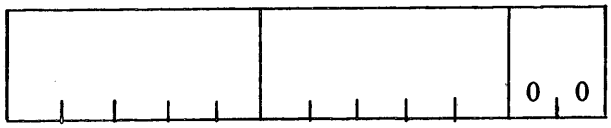


W2



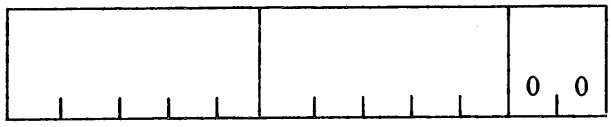

Dictionary reference of inserted continue

W3



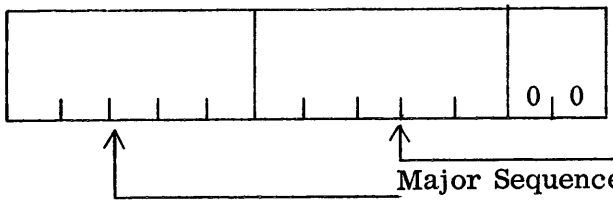

Sequence numbers of first usage as a literal (if any).

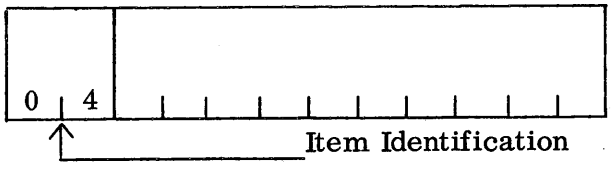
W4

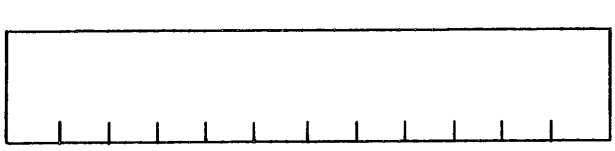



Sequence number of second usage as a literal (if any).

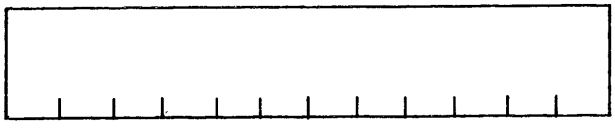
End of List Item



Unused



Unused

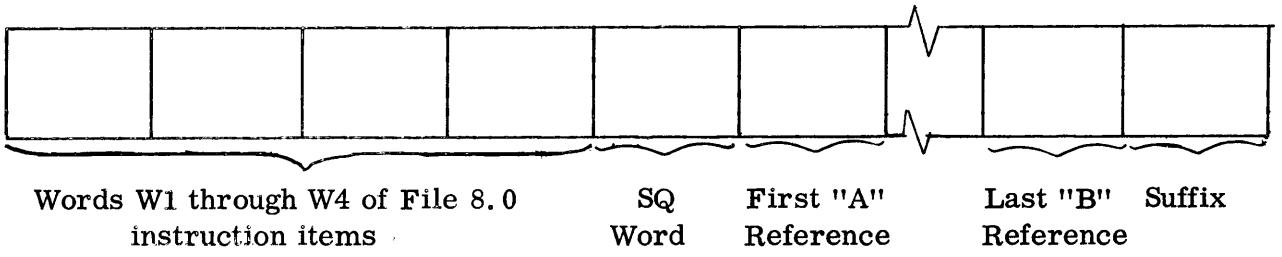
File 8.5 Items

File 8.5 items have precisely the same format as file 8.0 items with the exception that library call items may occur in file 8.5. A library call item has "-0" for its item identification. As library call items are only passed through Phase VIII without processing, no detailed item format is given here.

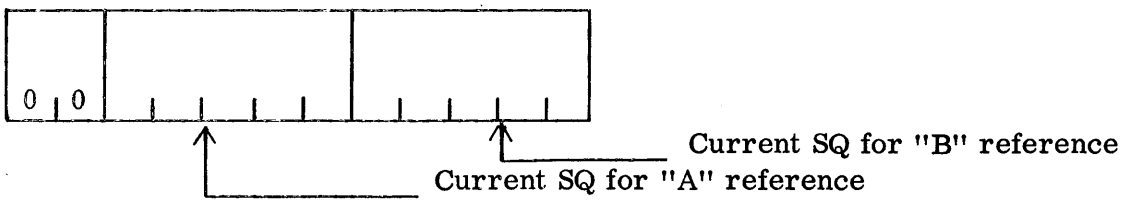
File 8.1 Items

File 8.1 is essentially a variable length item file. The file routine is coerced into treating variable length items by describing the file as consisting of one-word items and using the 17-word-read and n-word-write options. Its contents are essentially those of the merged input from files 8.0 and 8.5 with information added as developed by Phase VIIIa. Each item produced by Phase VIIIa is suffixed (not prefixed as file 8.1 is read backwards) by a word containing the item identification and a count of the number of words (not including this one) in the item. This suffix allows Phase VIIIb to read the remainder of the item in one operation.

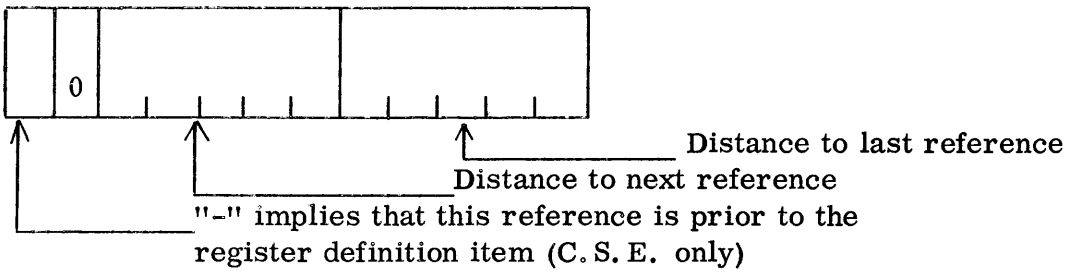
Instruction Item




The SQ word is

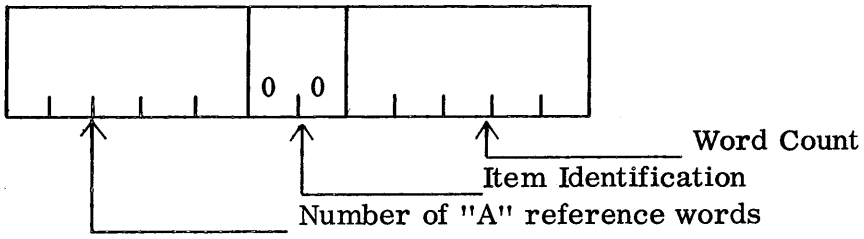



The "A" and "B" reference words are

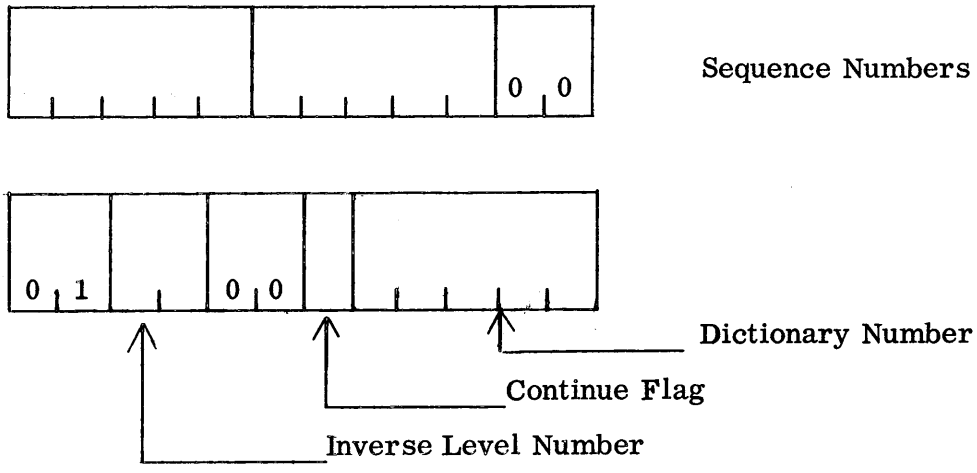


The total item may contain up to four "A" reference words and four "B" reference words.

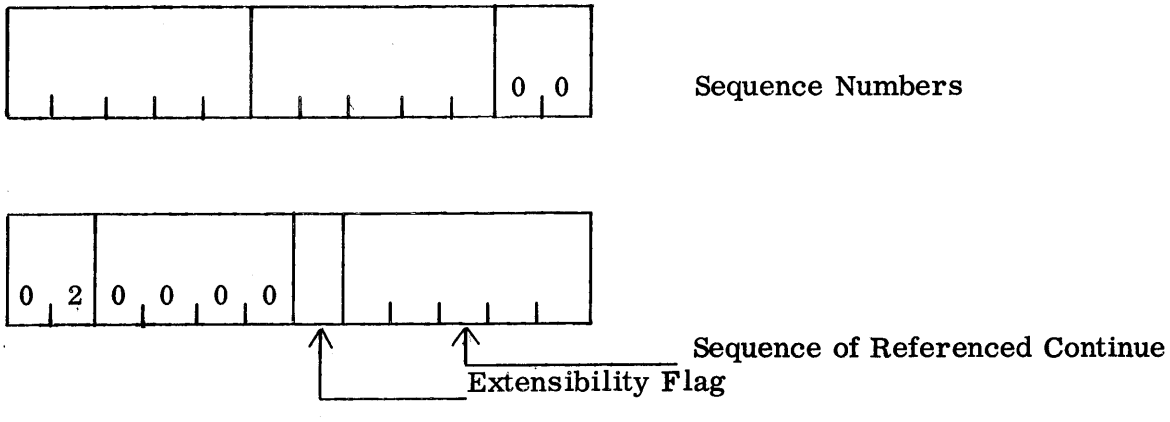
The suffix is

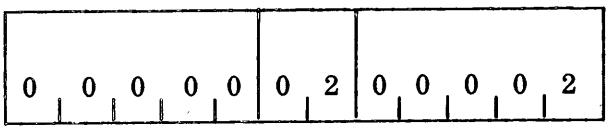



Continue item



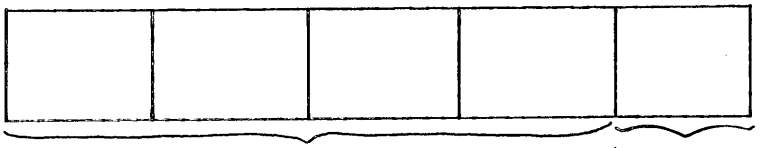

Successor item






Suffix

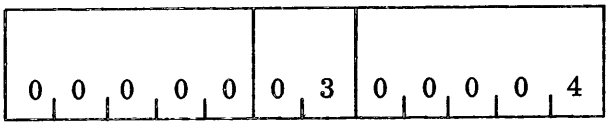
Instruction literal item



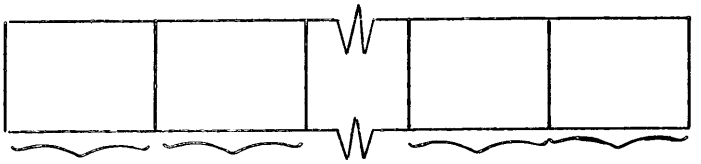
Words W1 through W4 of
File 8.0 instruction literal

Suffix

The suffix is

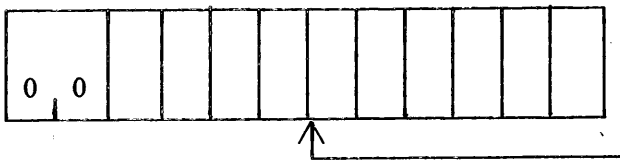



Index map item



Index Map Words Suffix

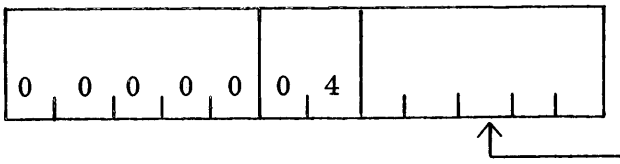
Each index map word records the usage of ten fast registers and is of the form




Preferred register flags

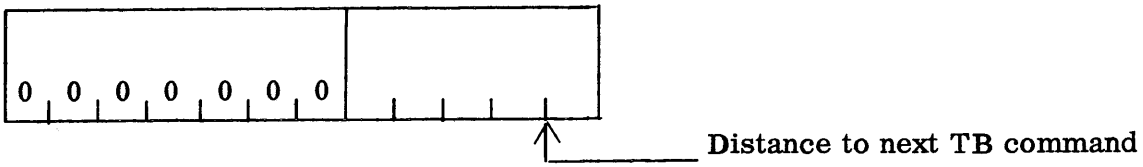
0 - not preferred
1 - preferred

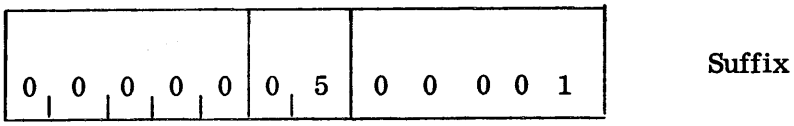
The suffix is




Word count

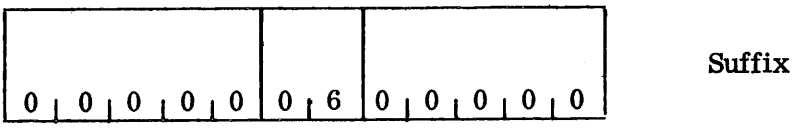
TB distance item



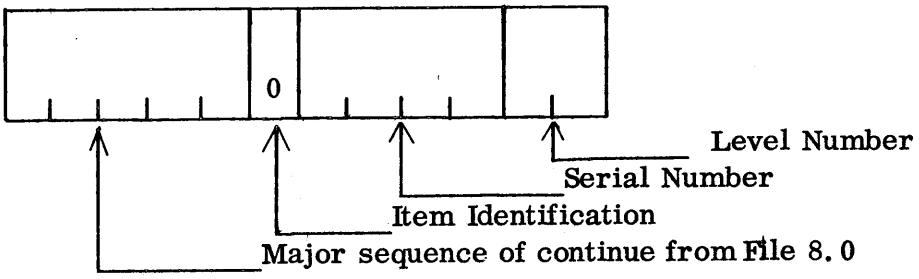

End of List item

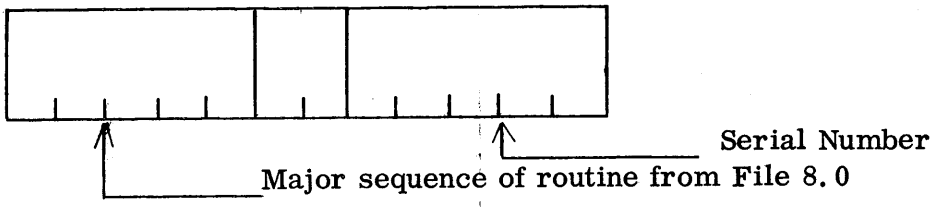
The End of List item is merely a flag and carries information only by its presence. Hence, only the suffix is required.



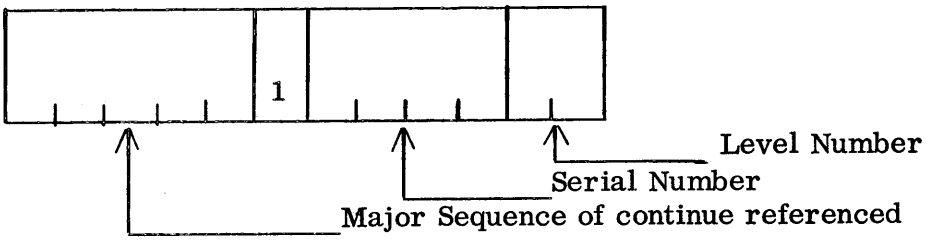

File 8.2 Items

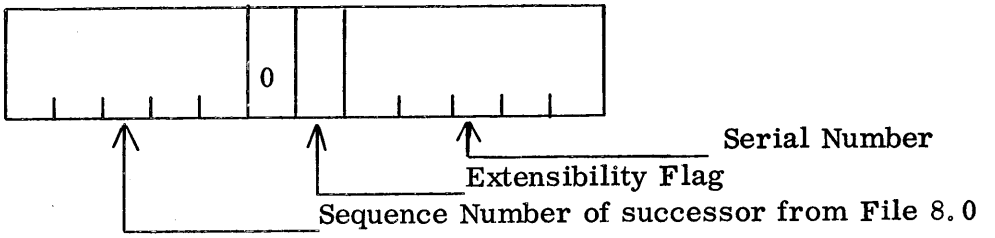
Continue item



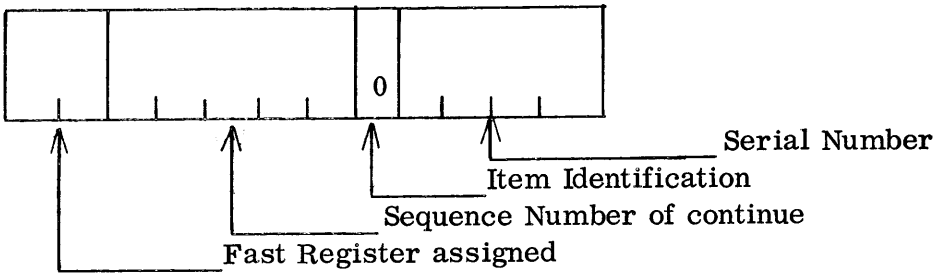
Successor item

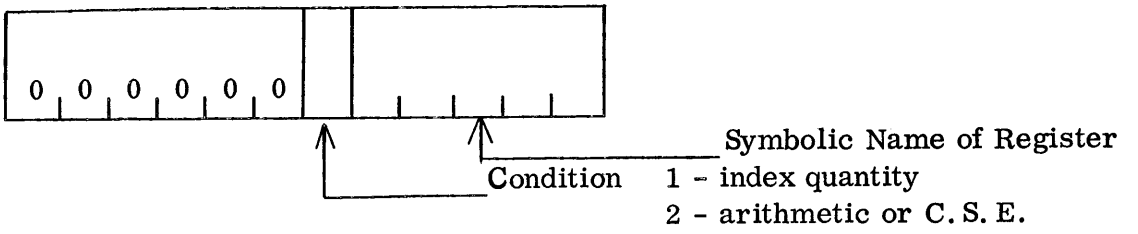



File 8.3 Items

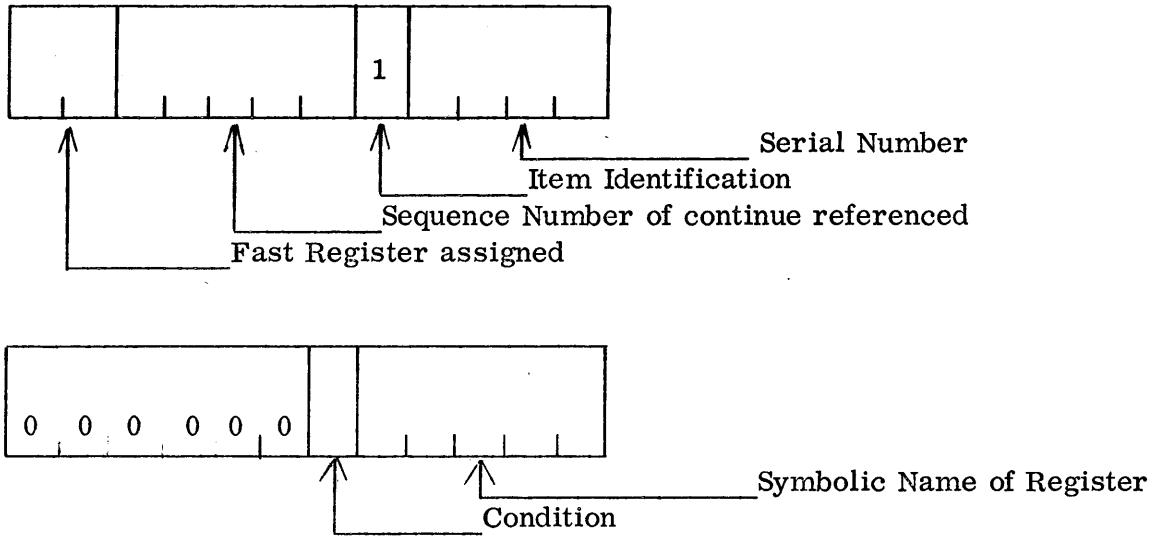
Entry Condition item



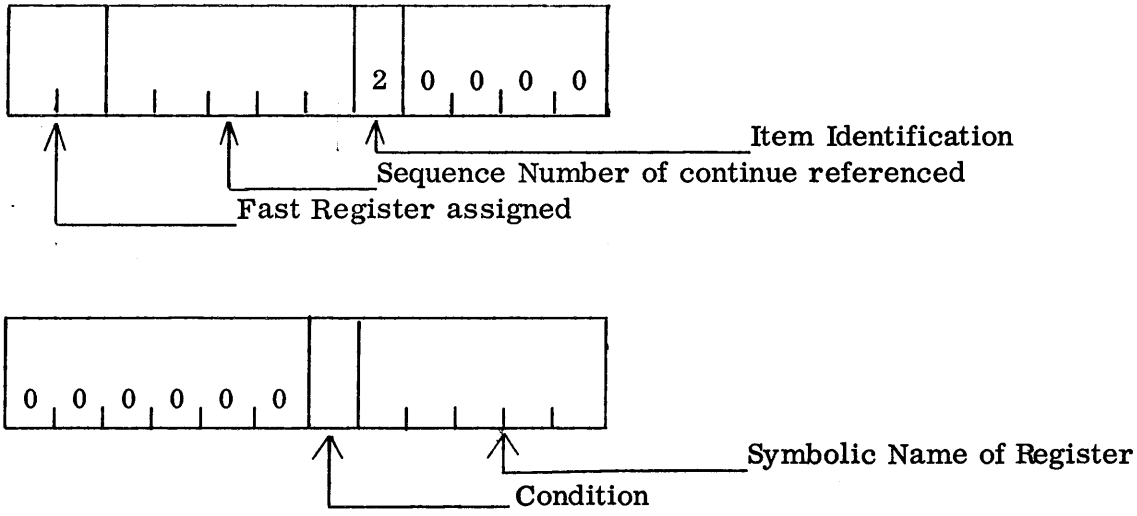




Note: for condition 2 the symbolic name is not required and is always made zero.

Exit Condition item

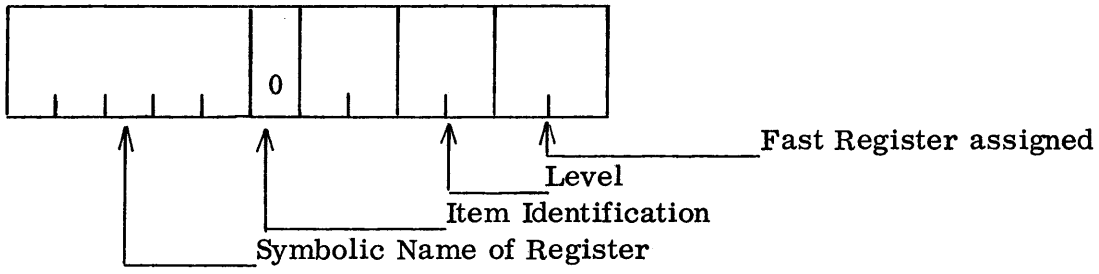



Special Continue item




File 8.4 items

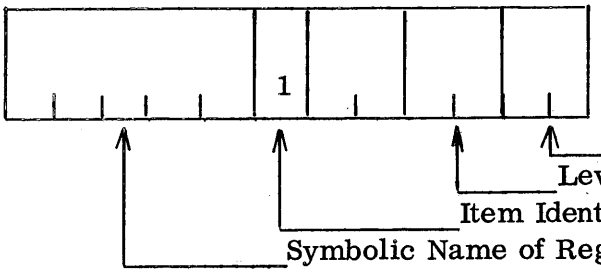
Fetch Request item



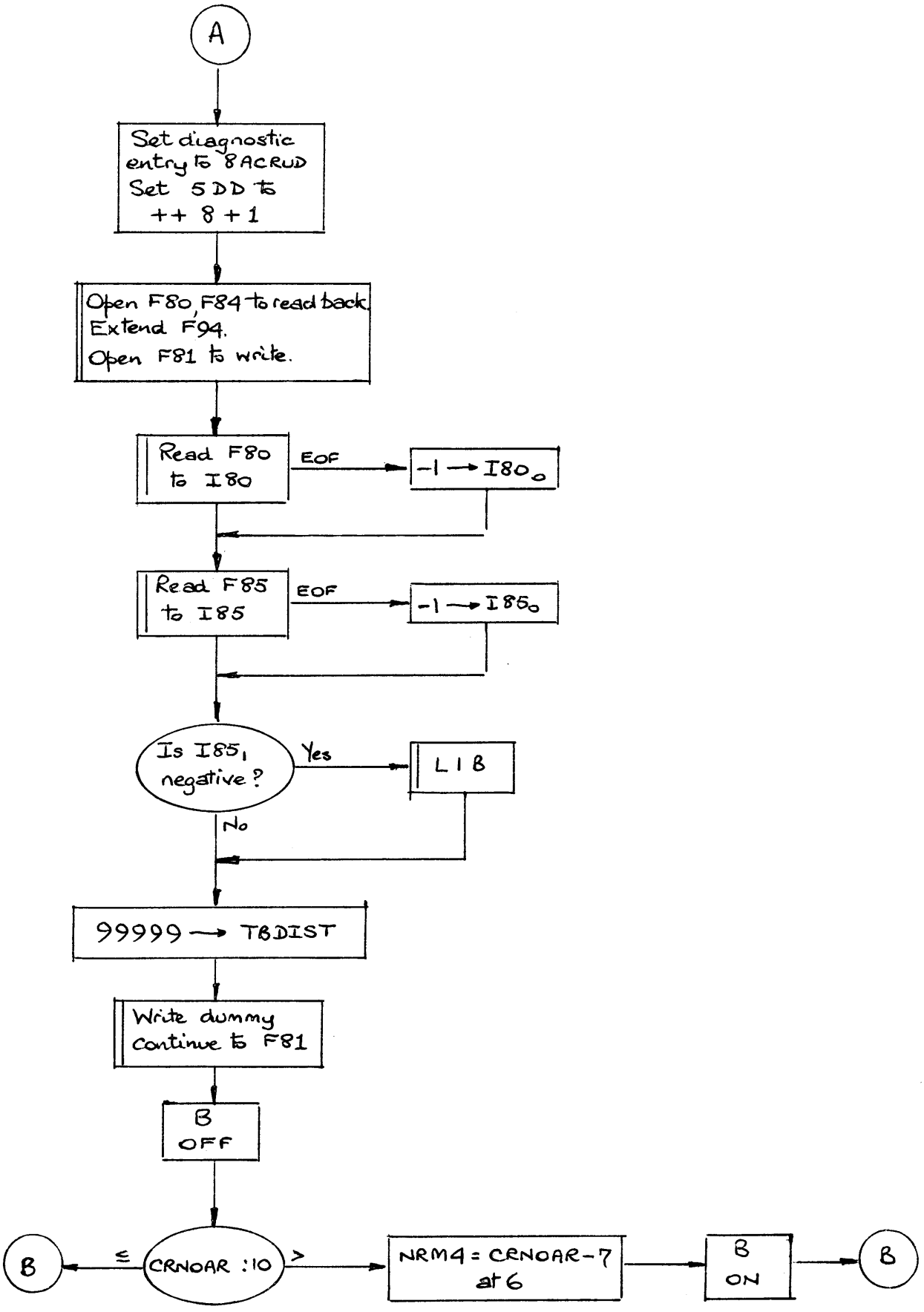


Sequence Numbers

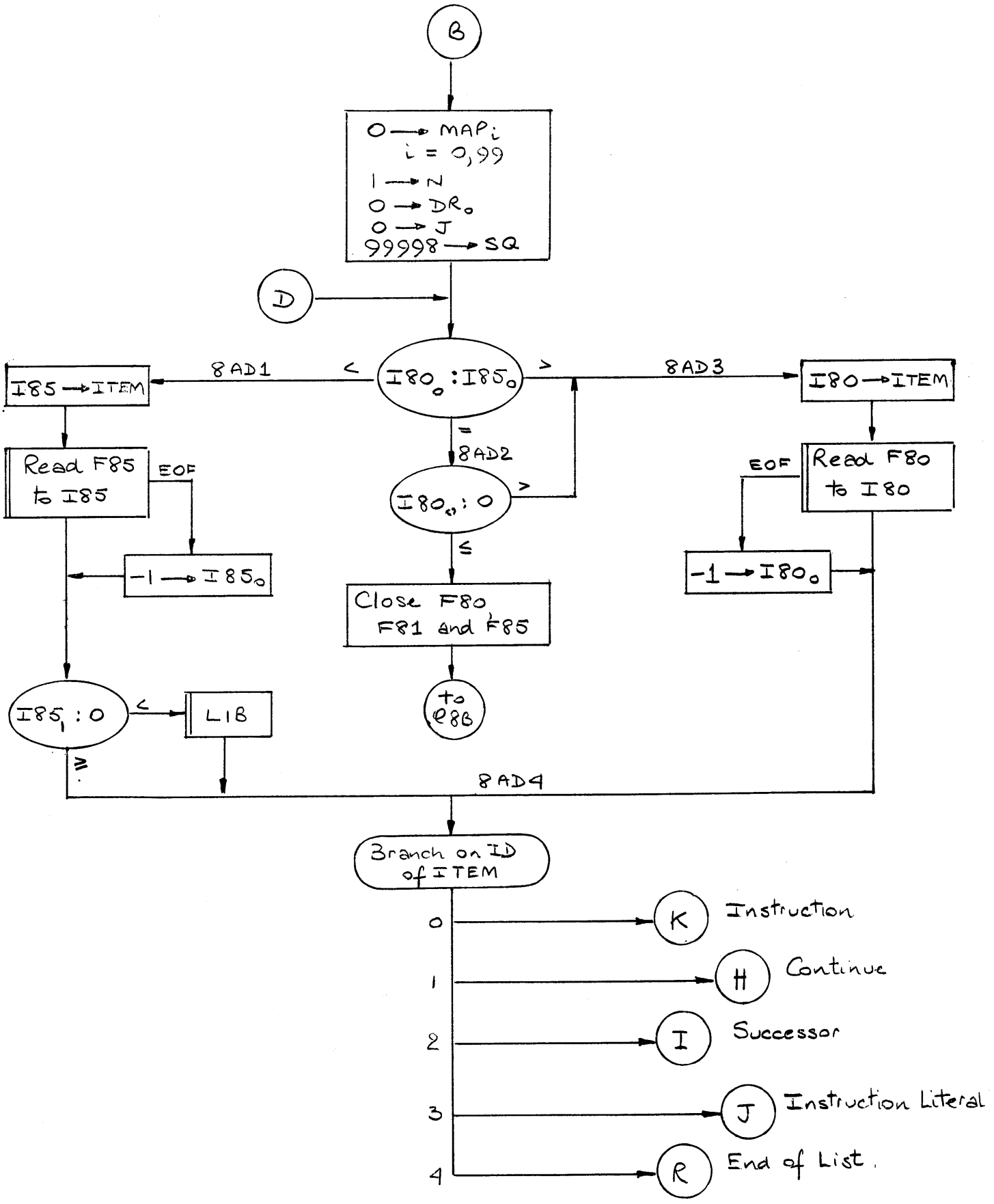
Index Definition item



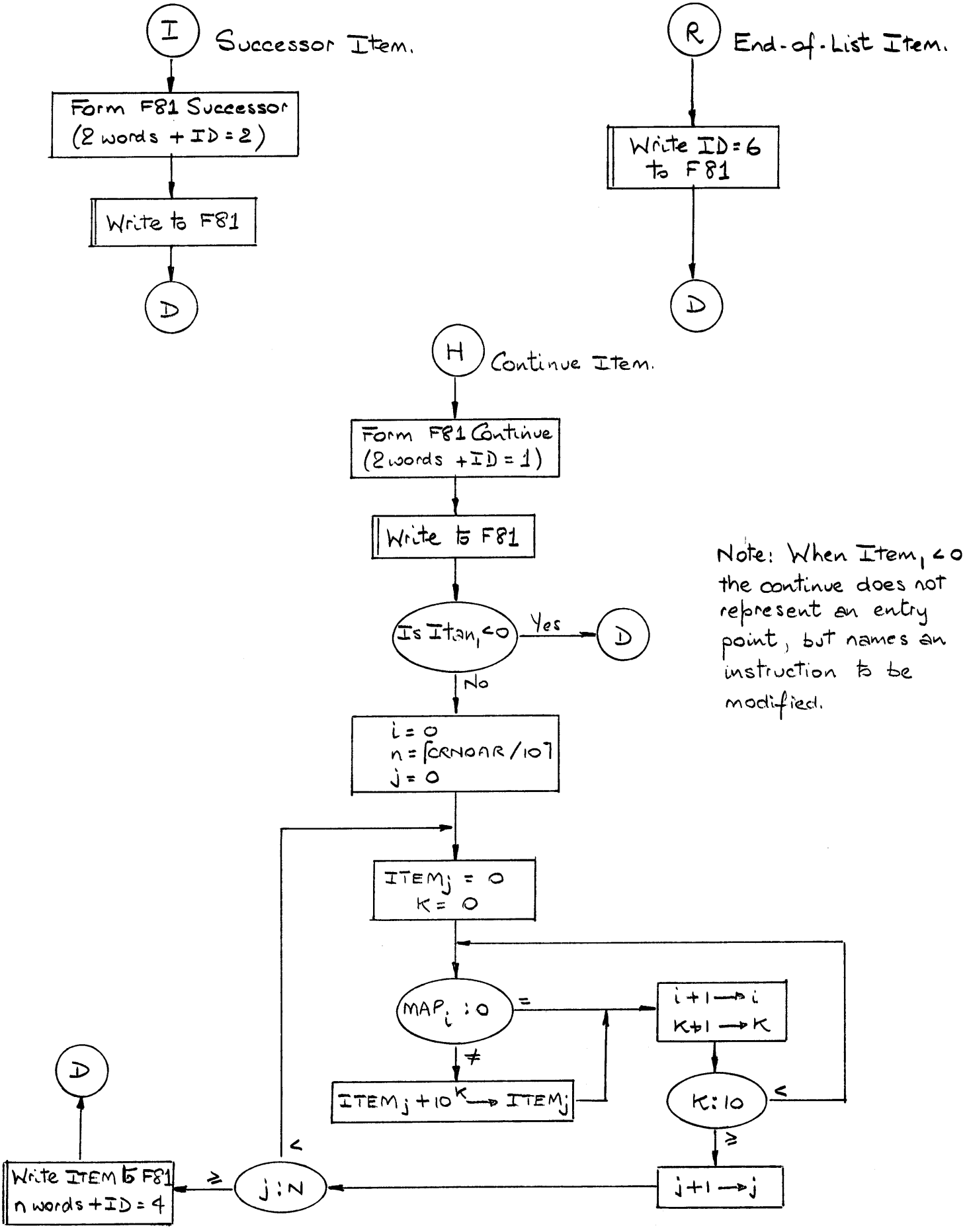
BACKWARD SCAN



BACKWARD SCAN.

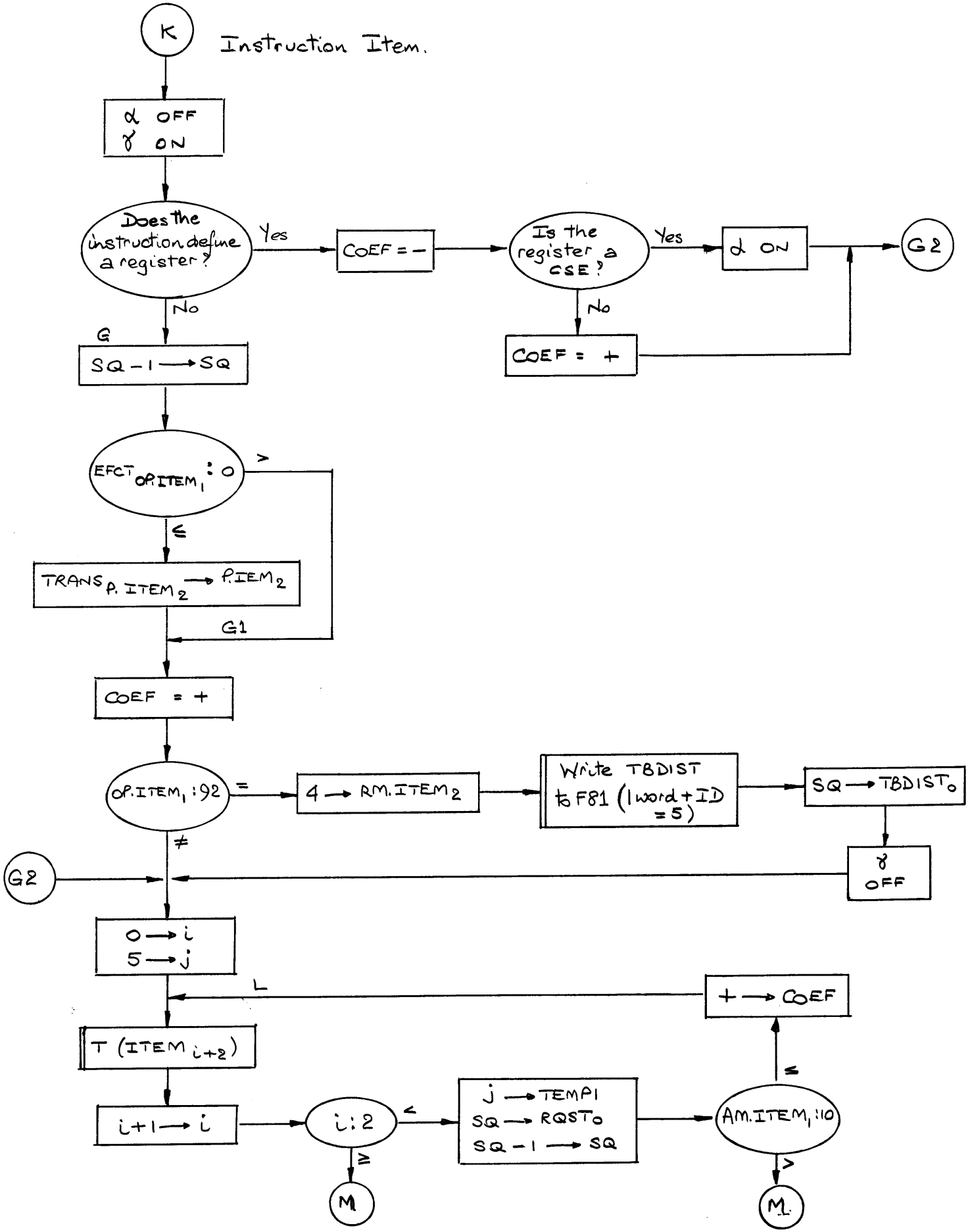


# BACKWARD SCAN.

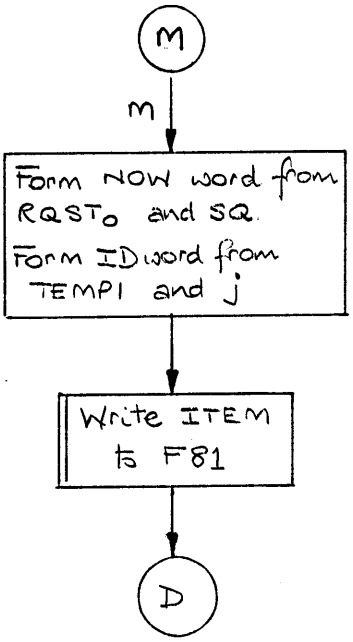


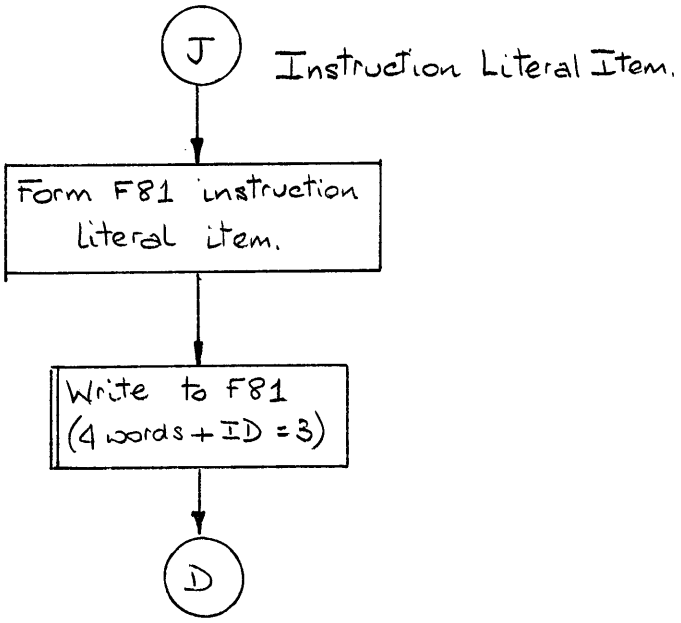

Note: When Item1 < 0 the continue does not represent an entry point, but names an instruction to be modified.

BACKWARD SCAN.

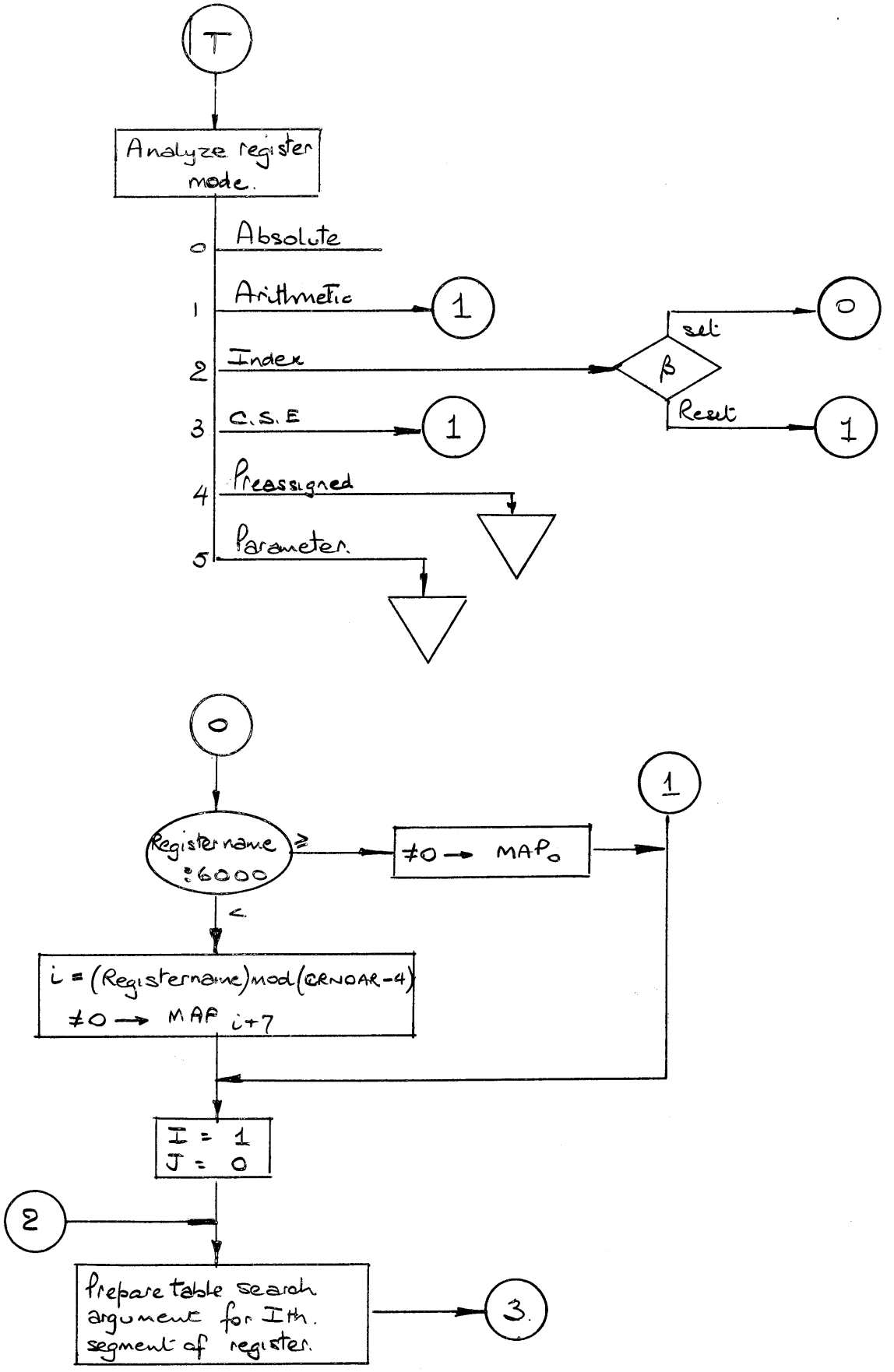


BACKWARD SCAN

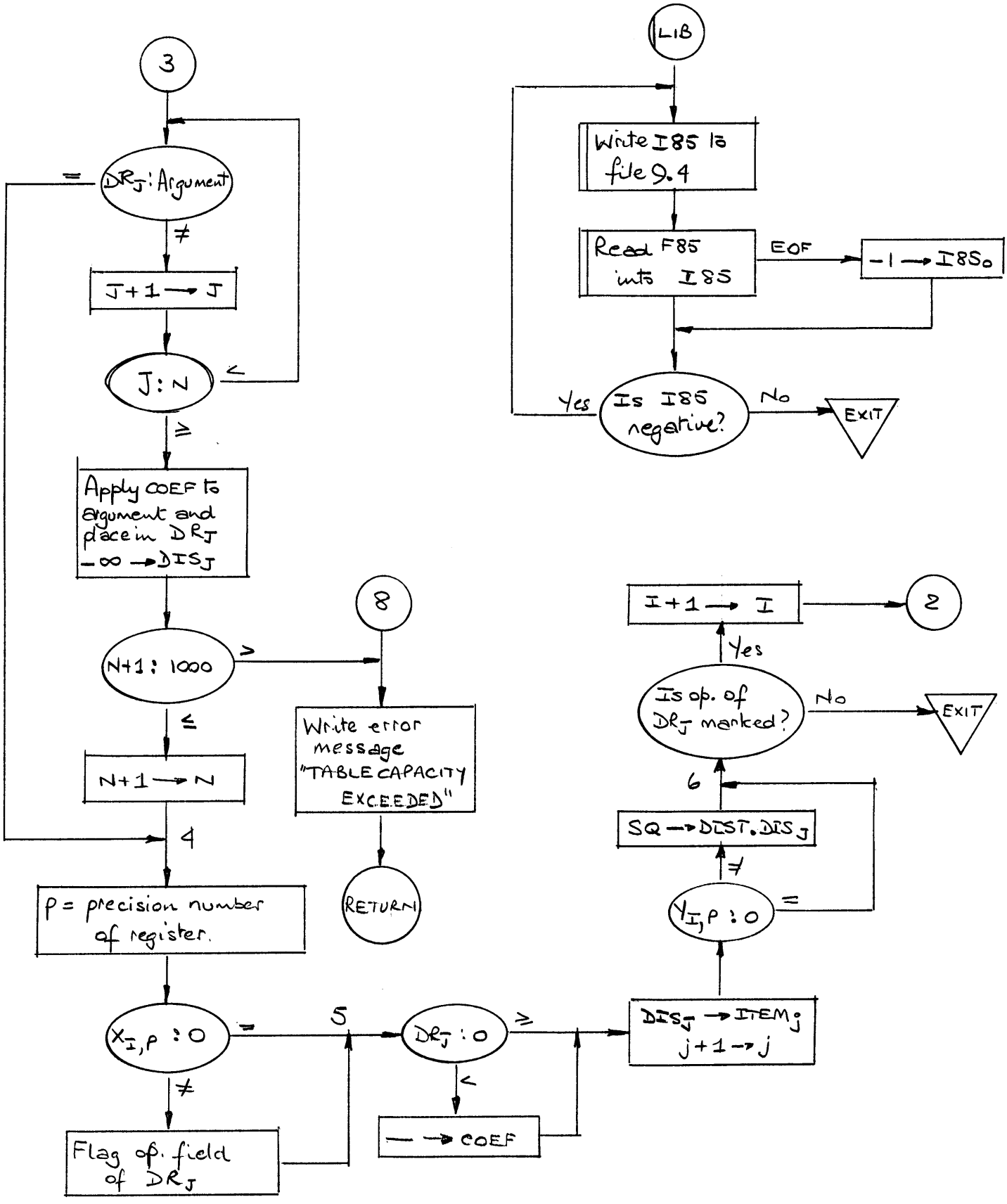


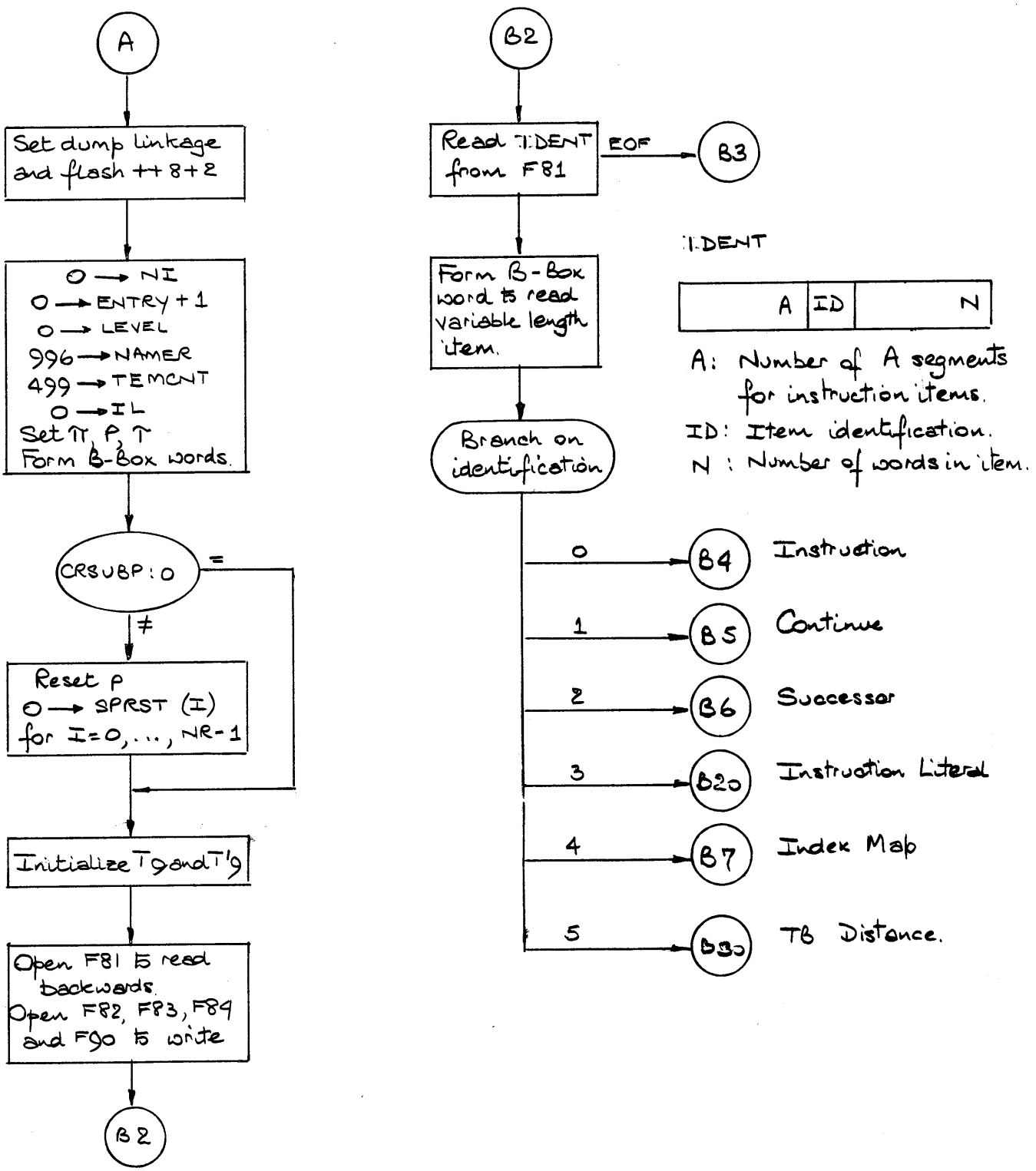
# BACKWARD SCAN.



BACKWARD SCAN.

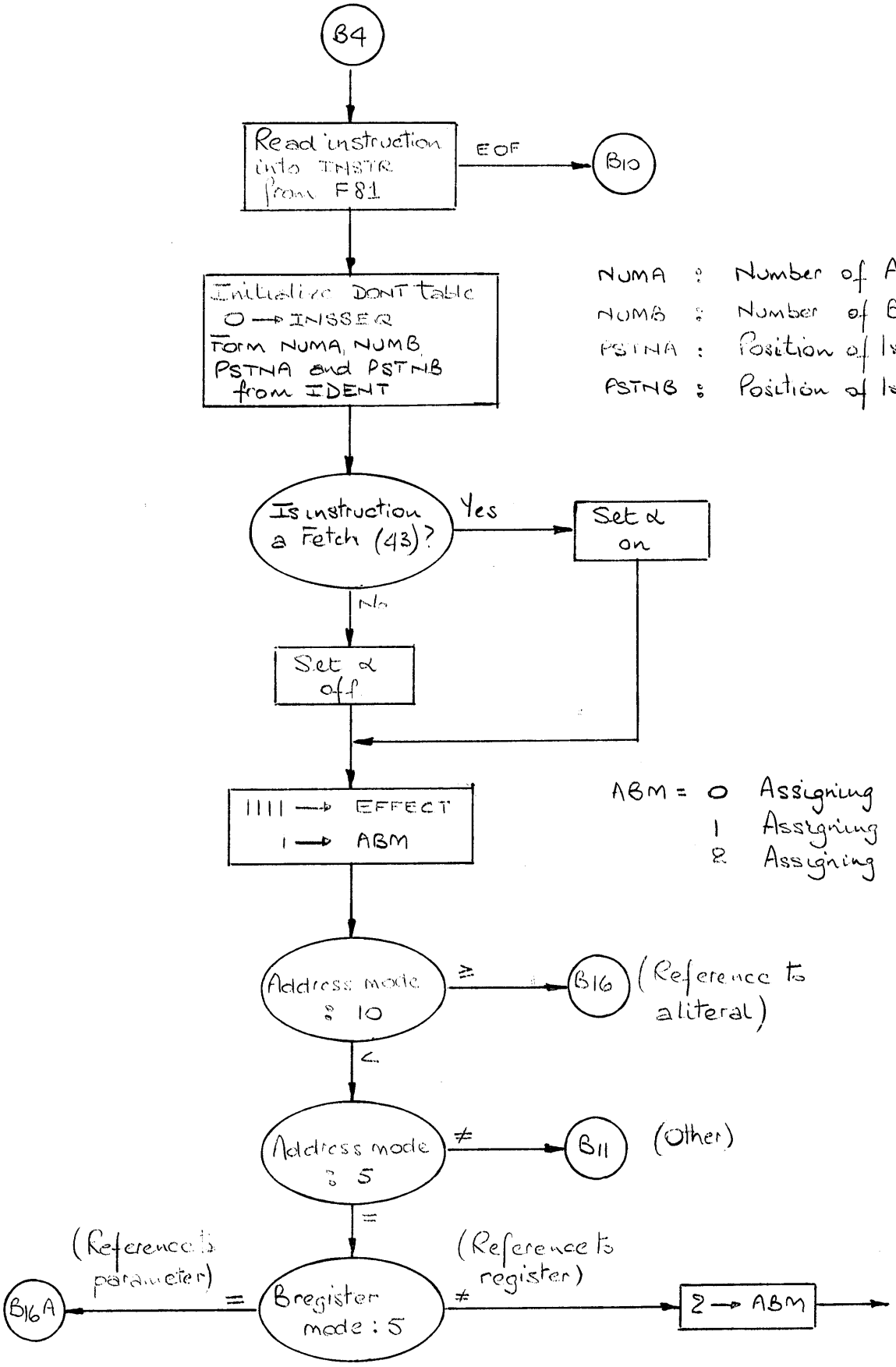


FORWARD SCAN.

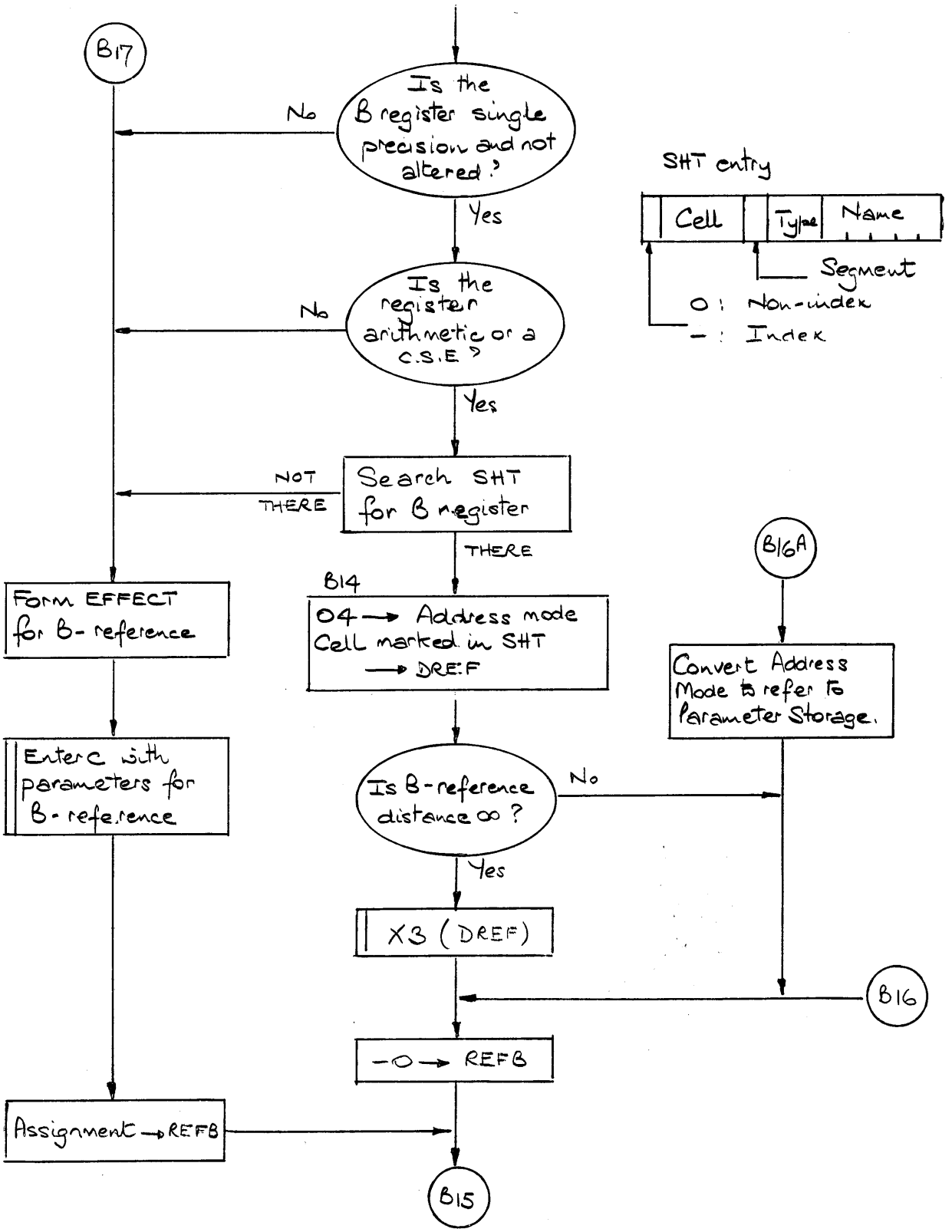


# LARC SCIENTIFIC COMPILER Phase VIII B

## FORWARD SCAN.

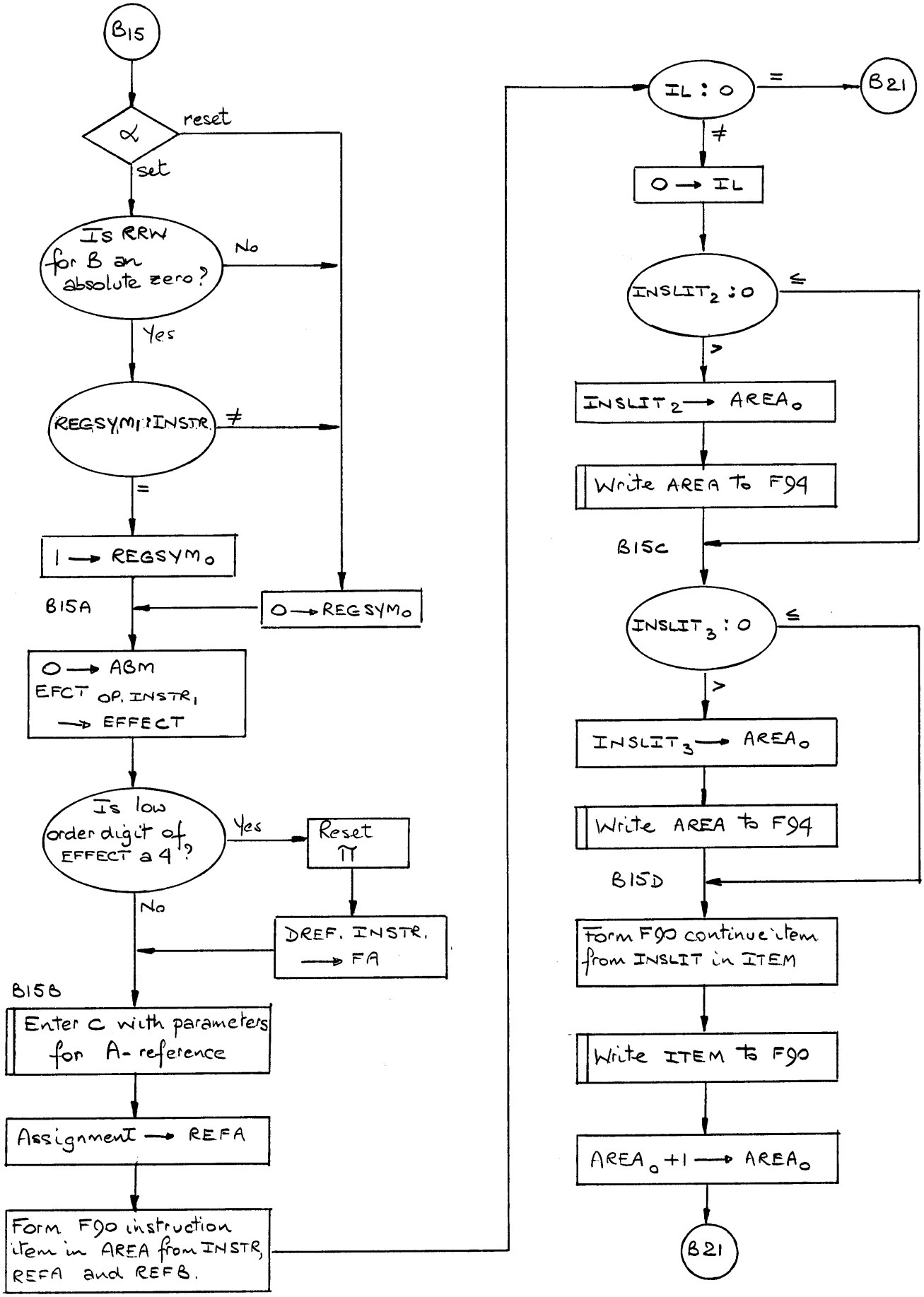


FORWARD SCAN.

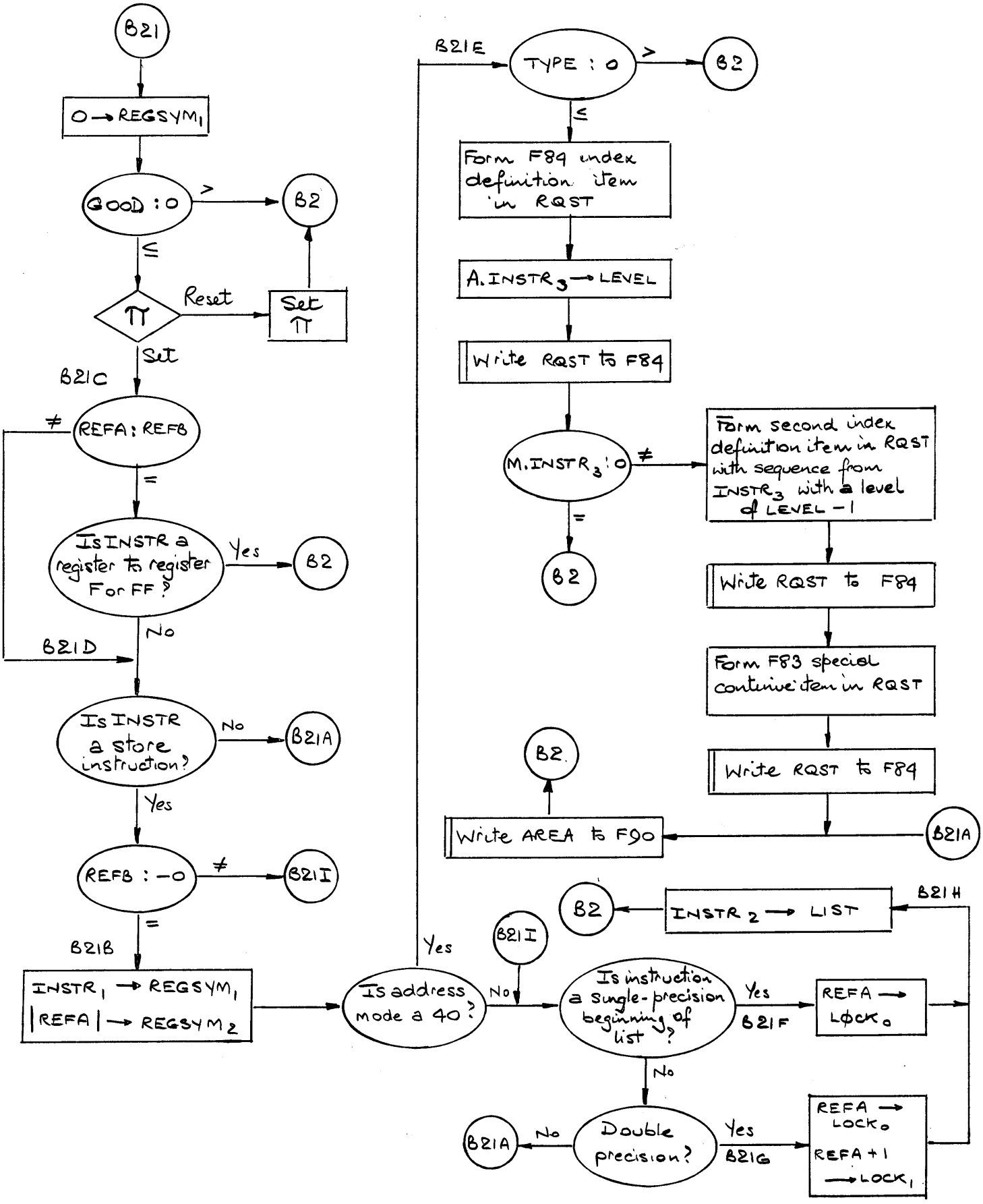


# LARC SCIENTIFIC COMPILER Phase VIII B

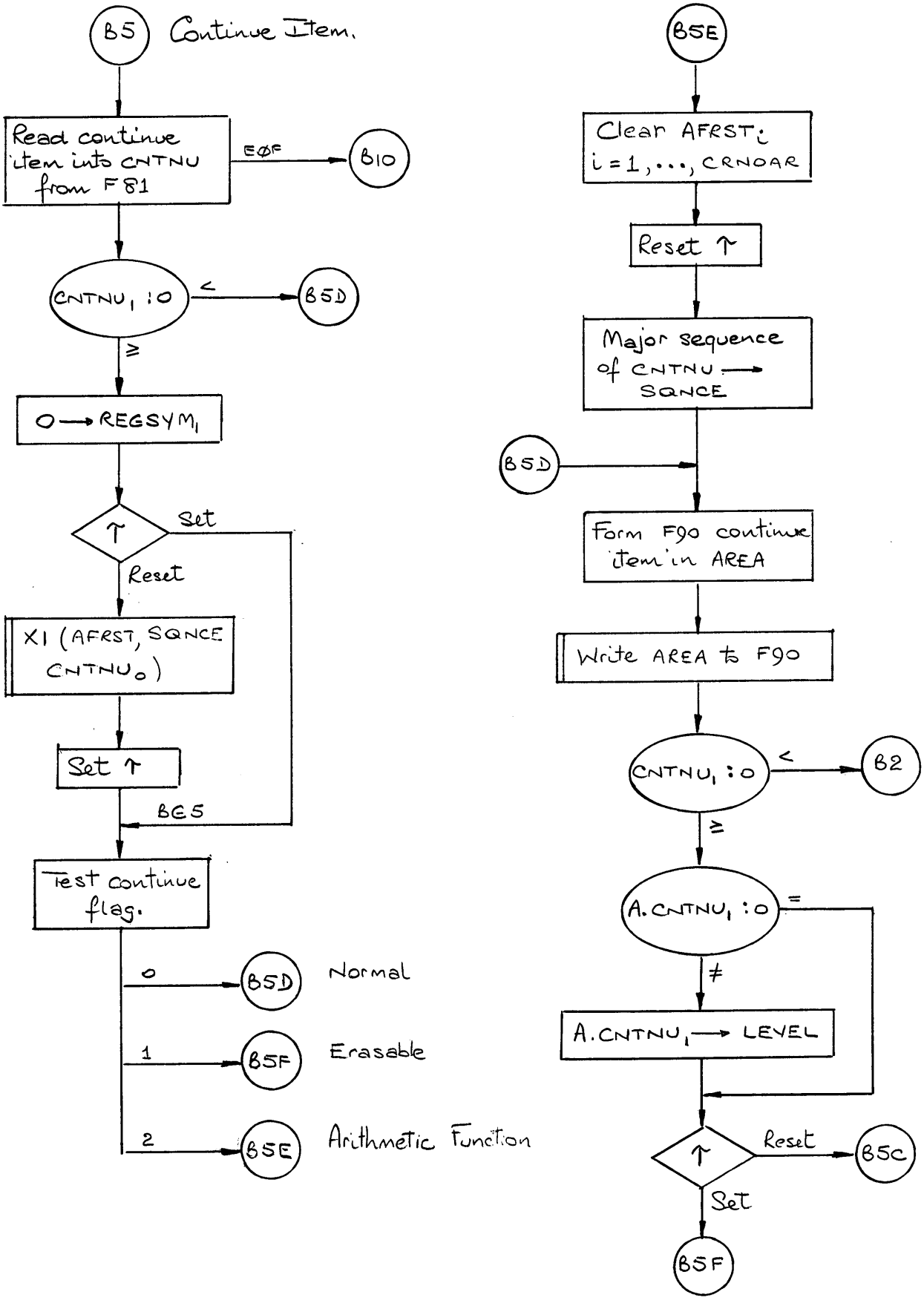
## FORWARD SCAN



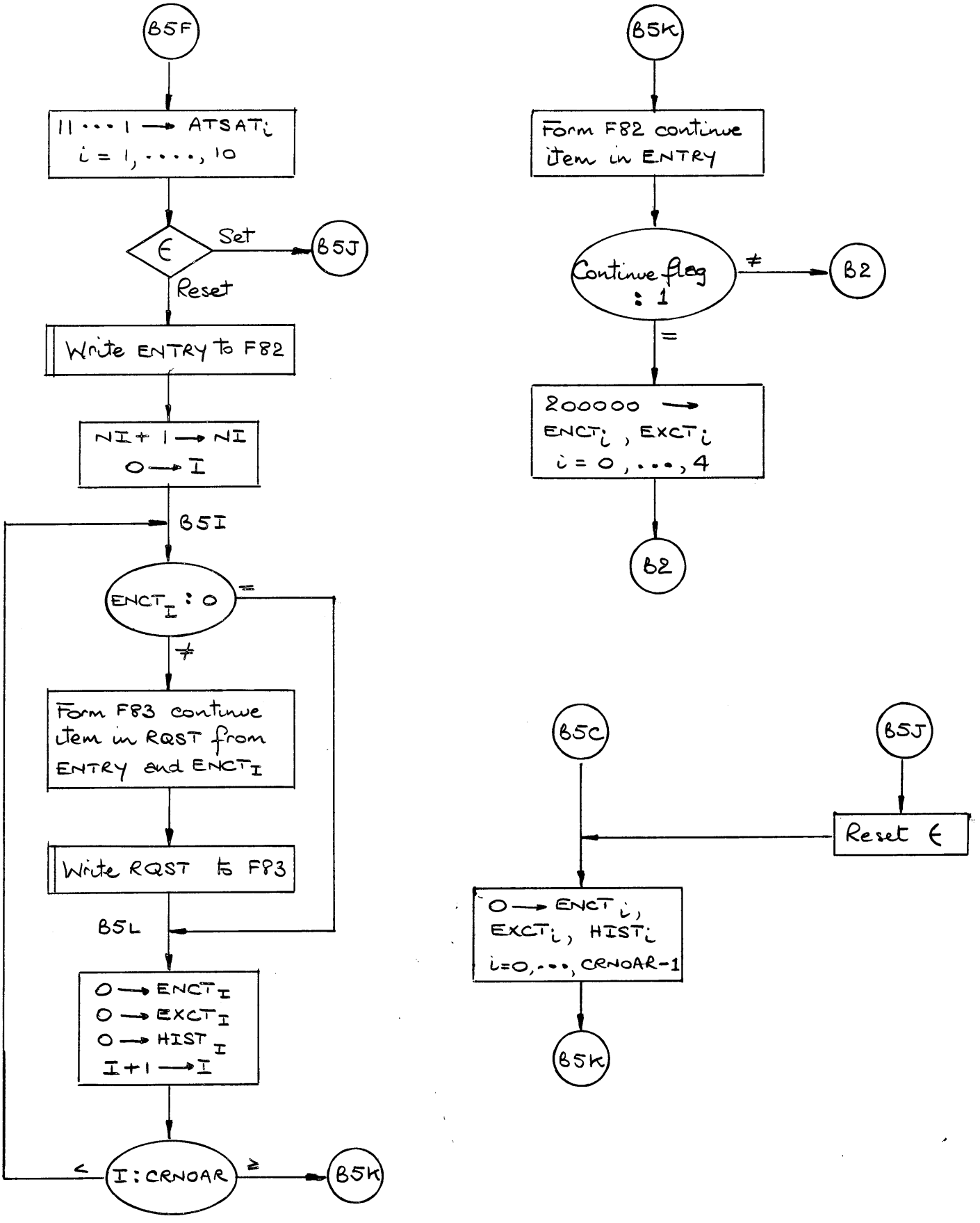
FORWARD SCAN.



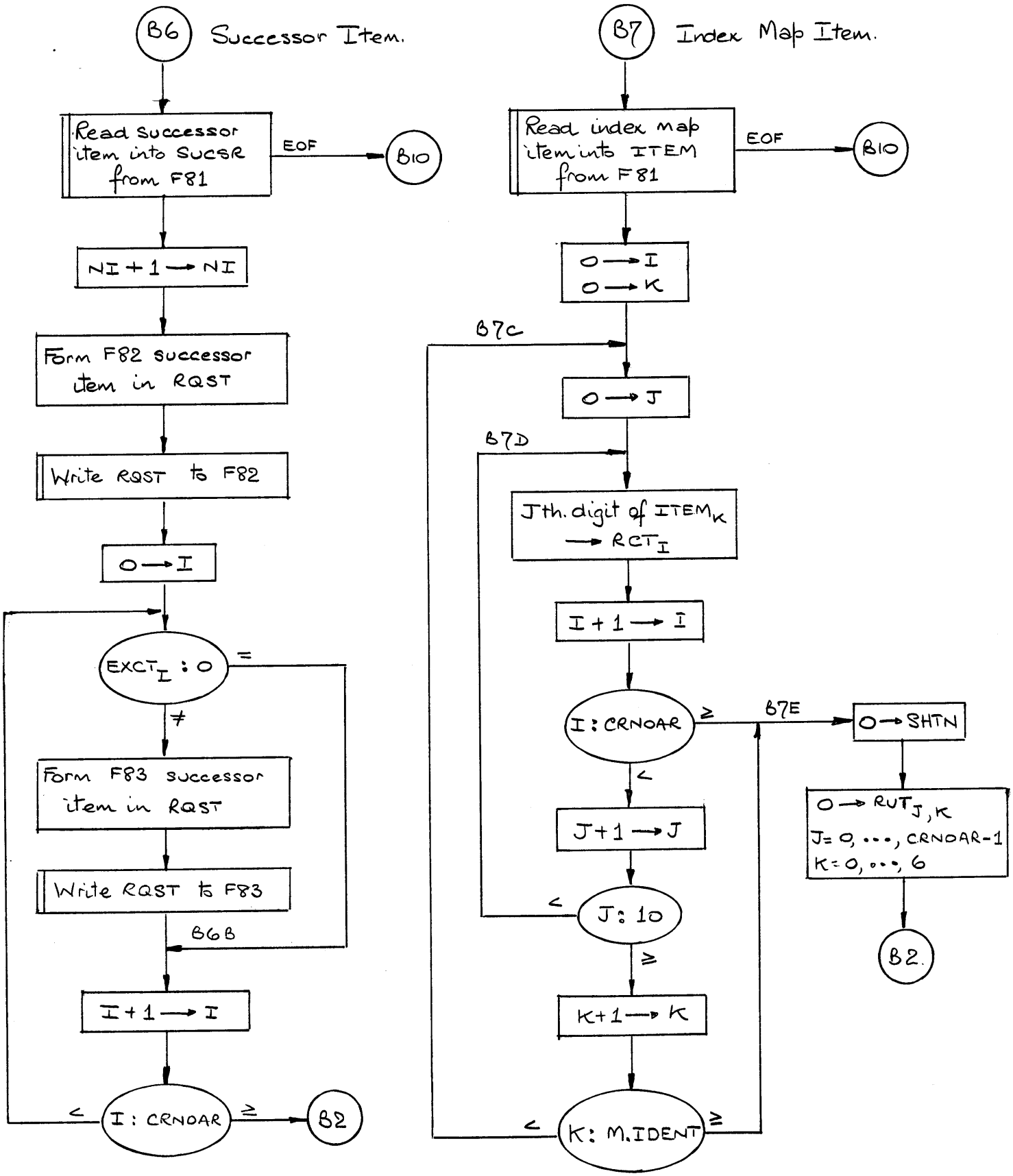
FORWARD SCAN



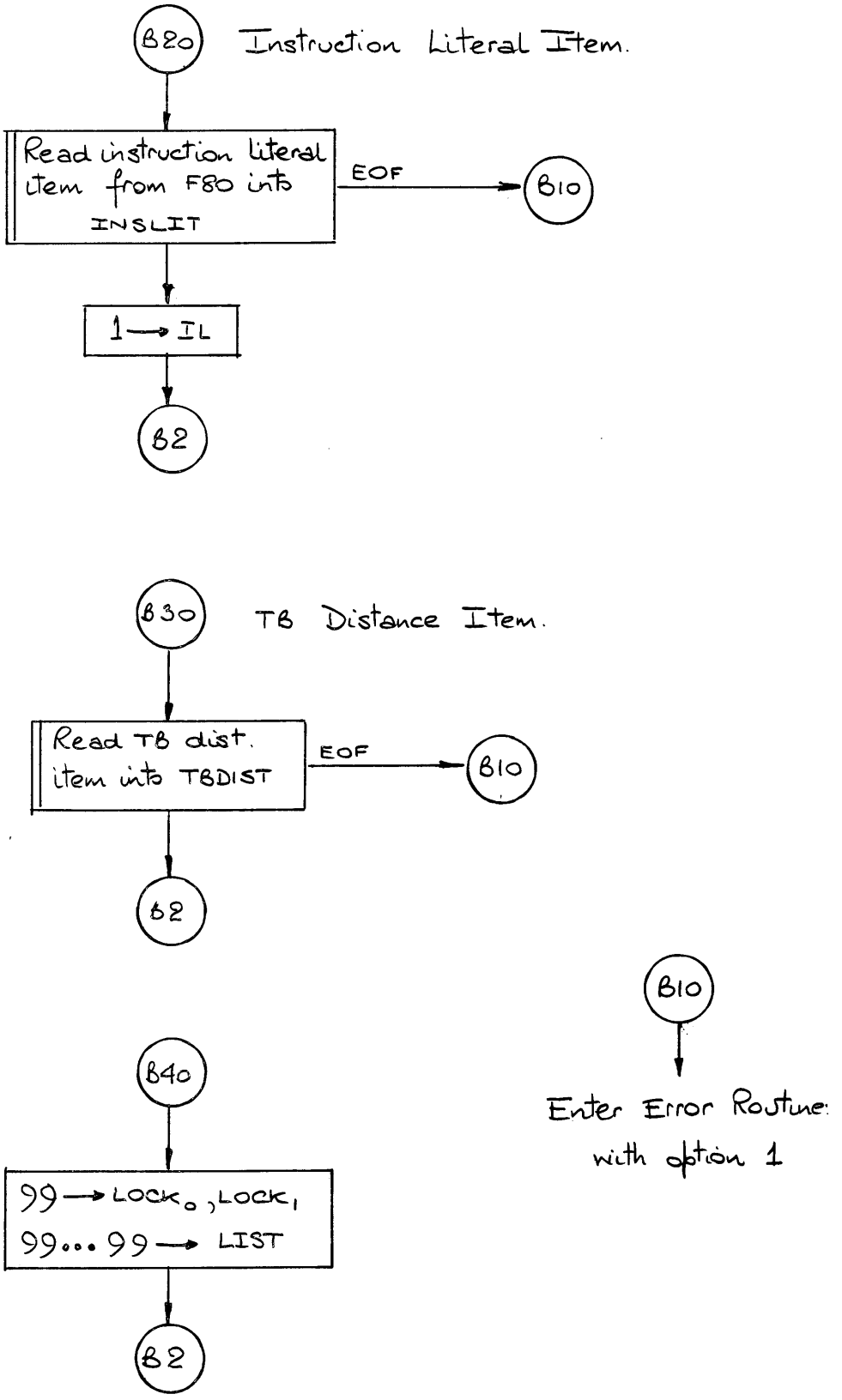
FORWARD SCAN.



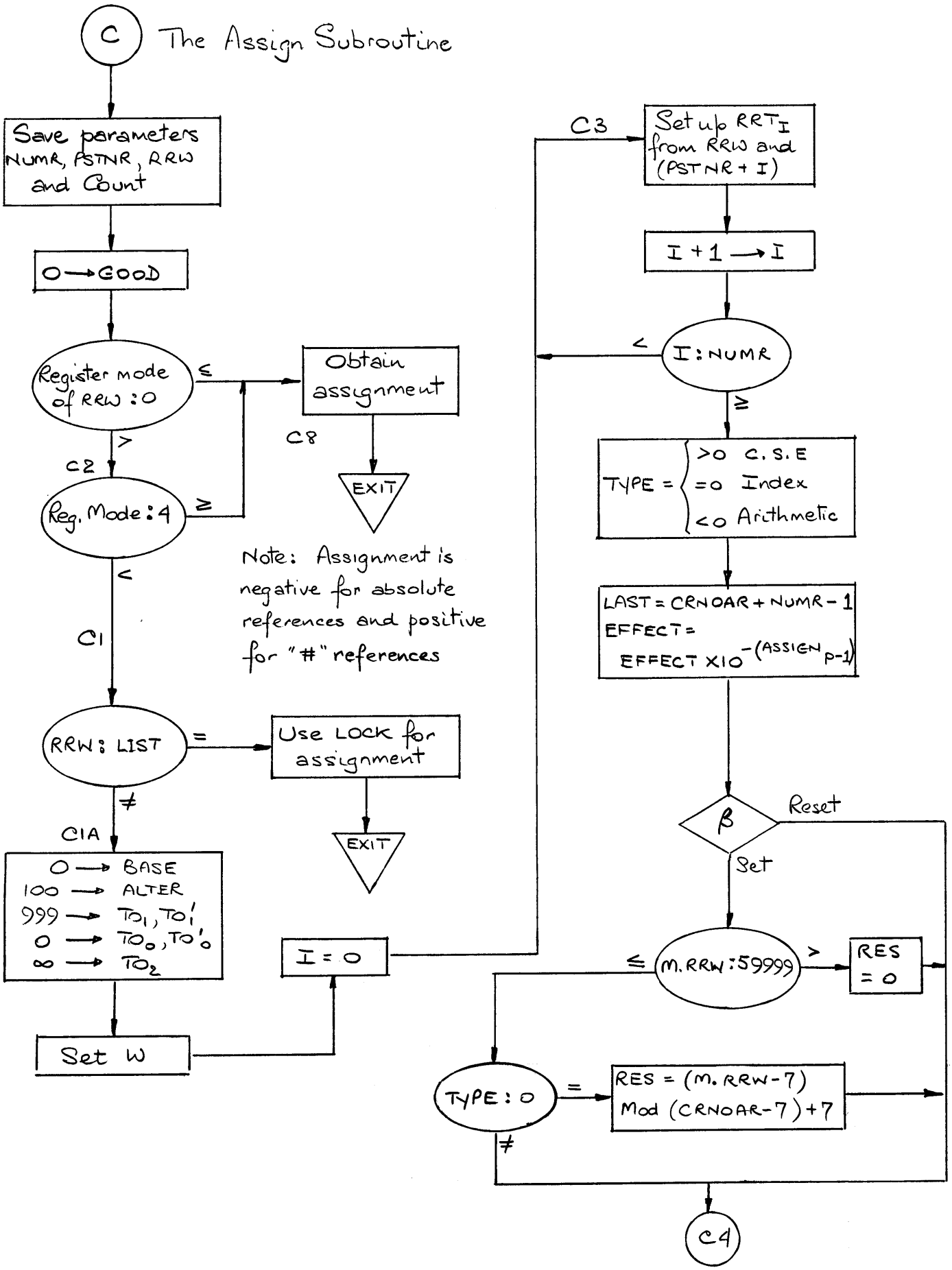
FORWARD SCAN.



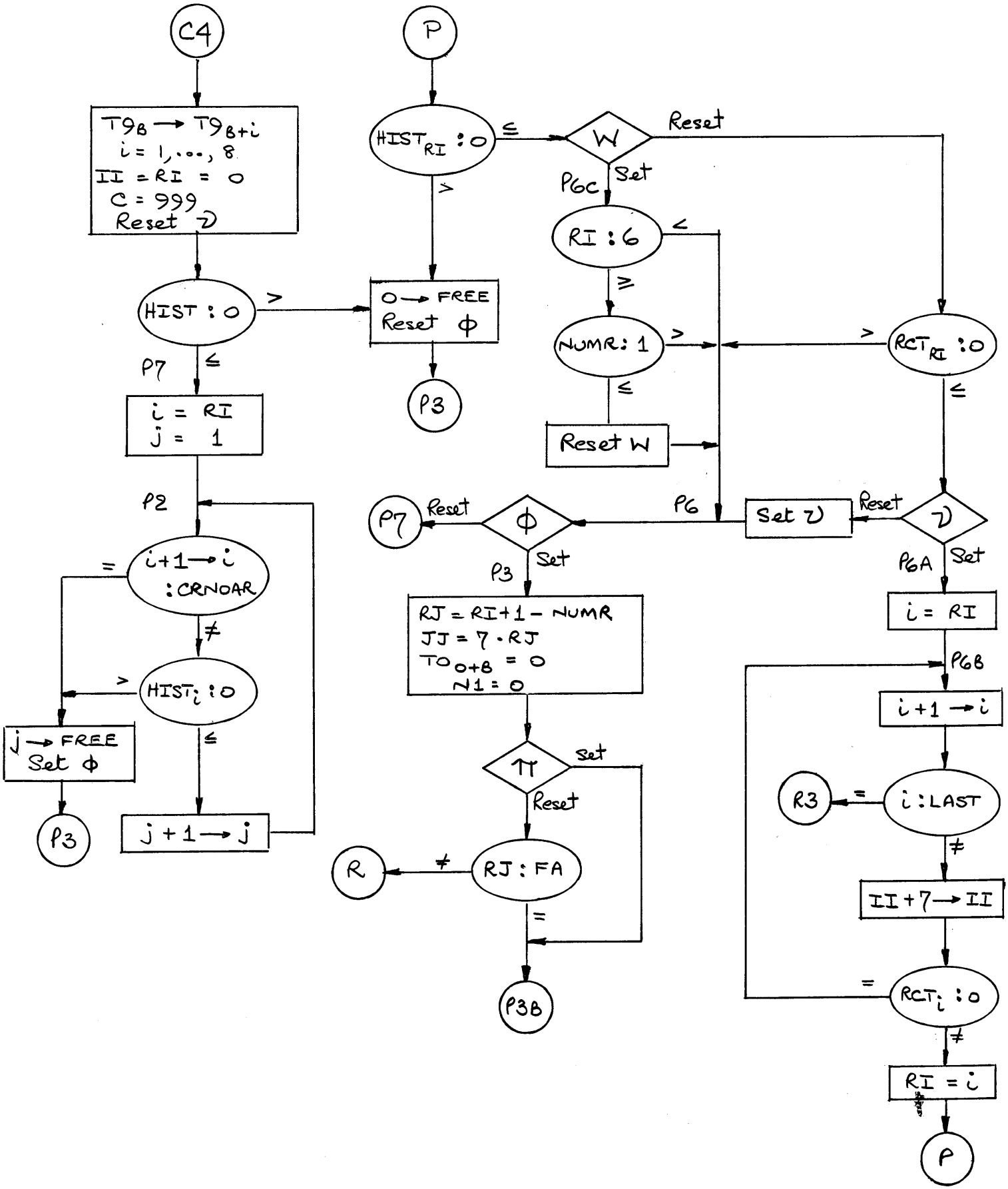
FORWARD SCAN.



FORWARD SCAN

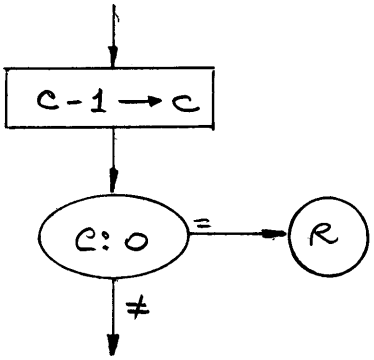


FORWARD SCAN

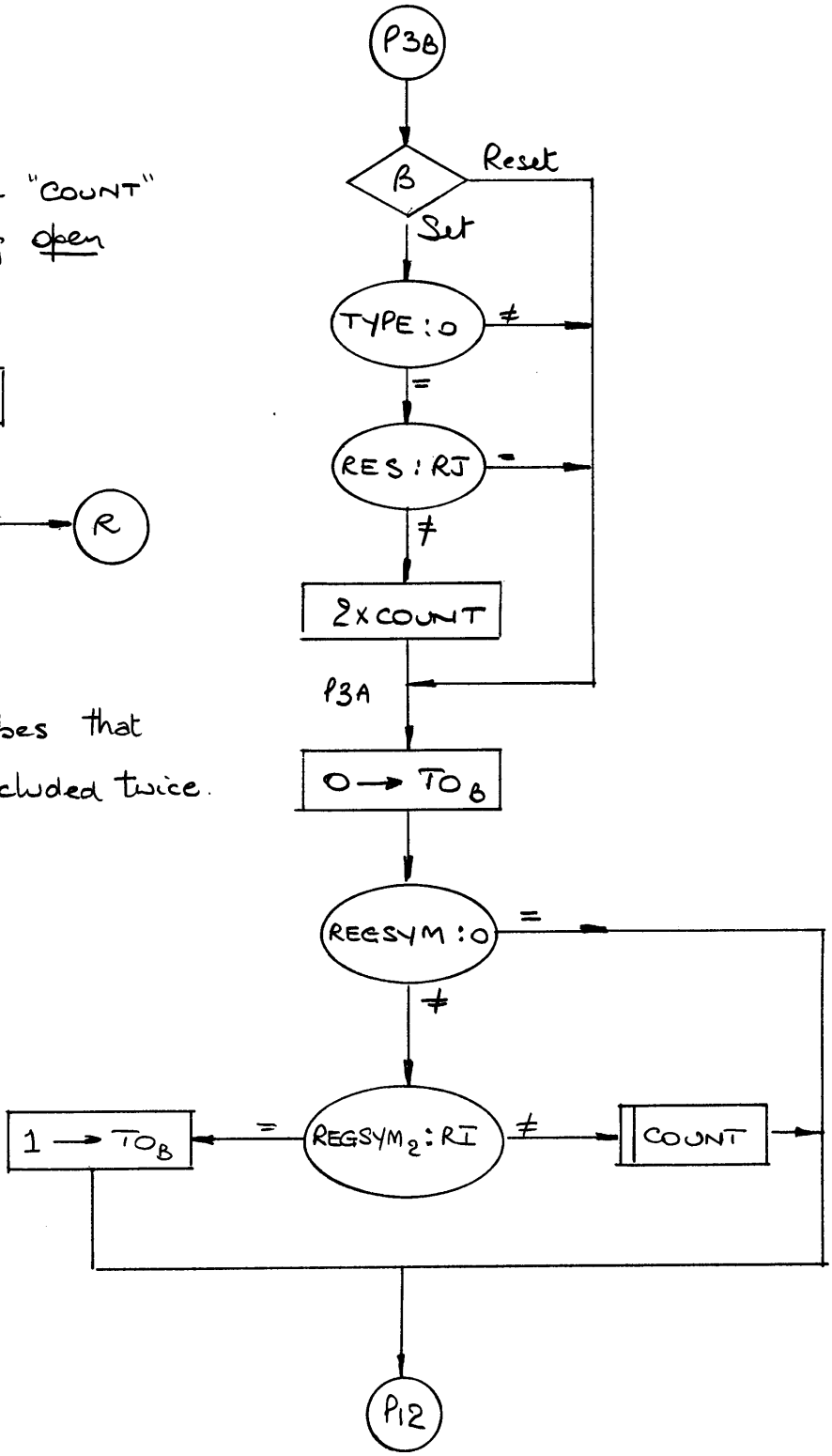


FORWARD SCAN.

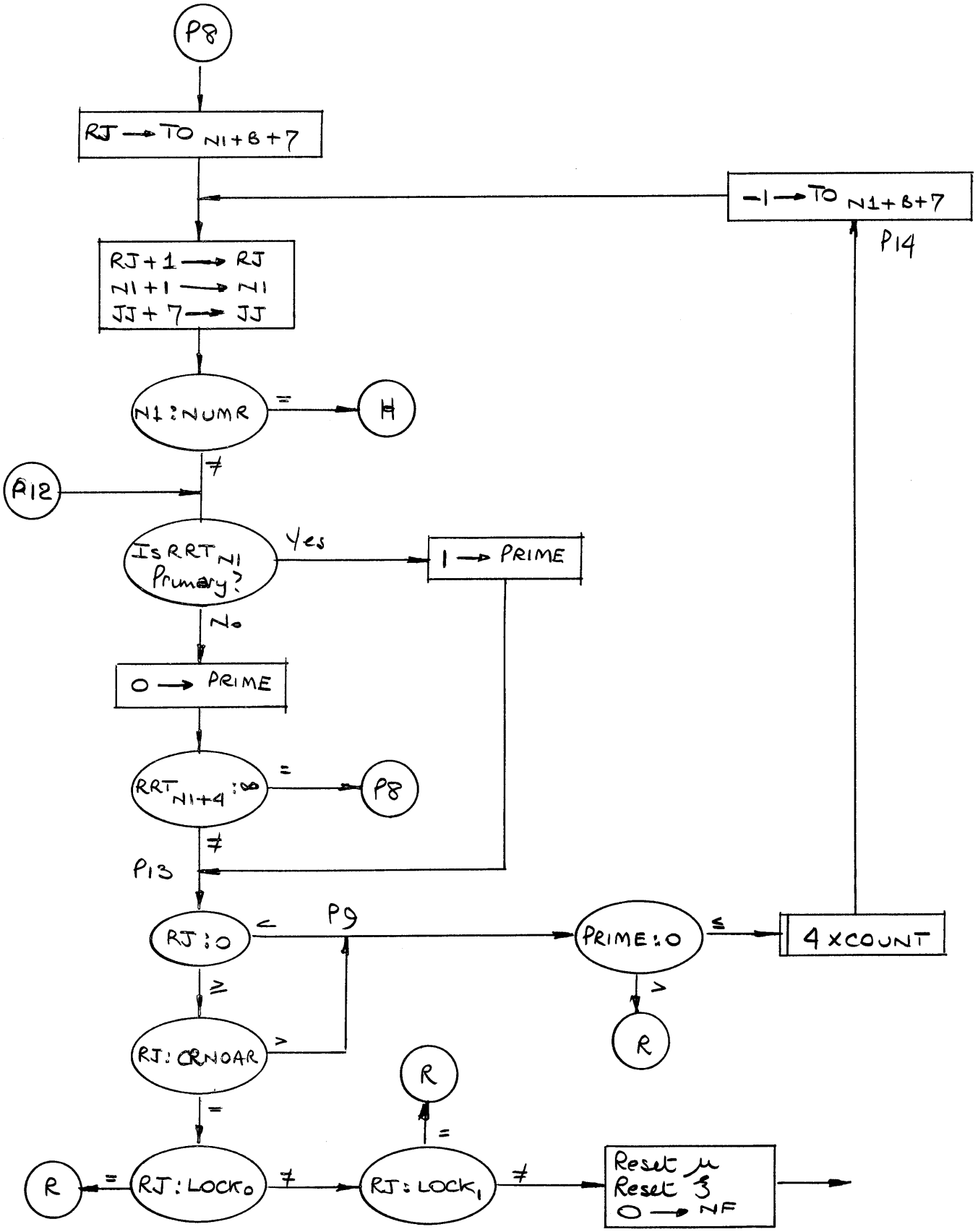
Note: The operation "COUNT" specifies the following open subroutine.



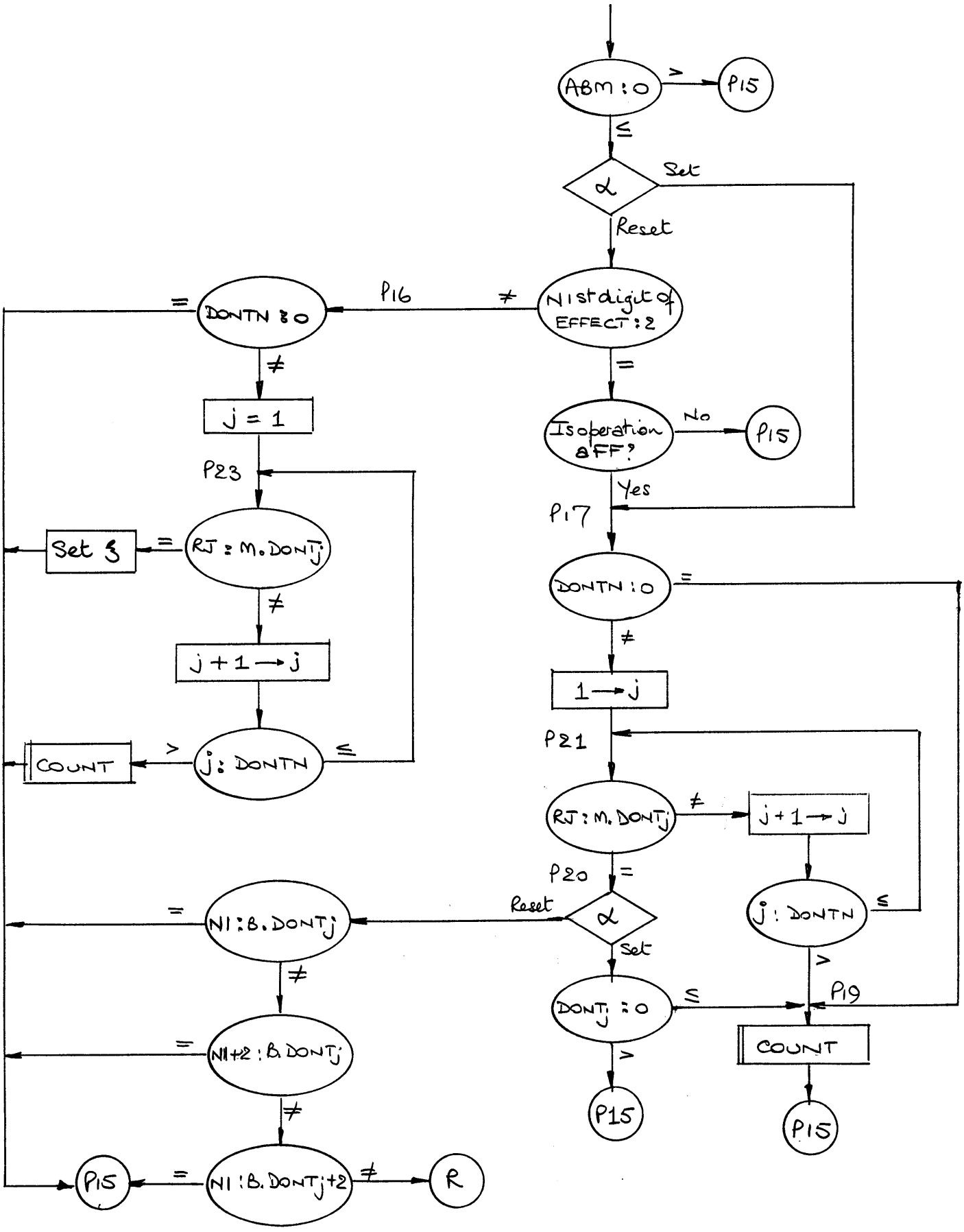

"2 x COUNT" prescribes that the subroutine be included twice.



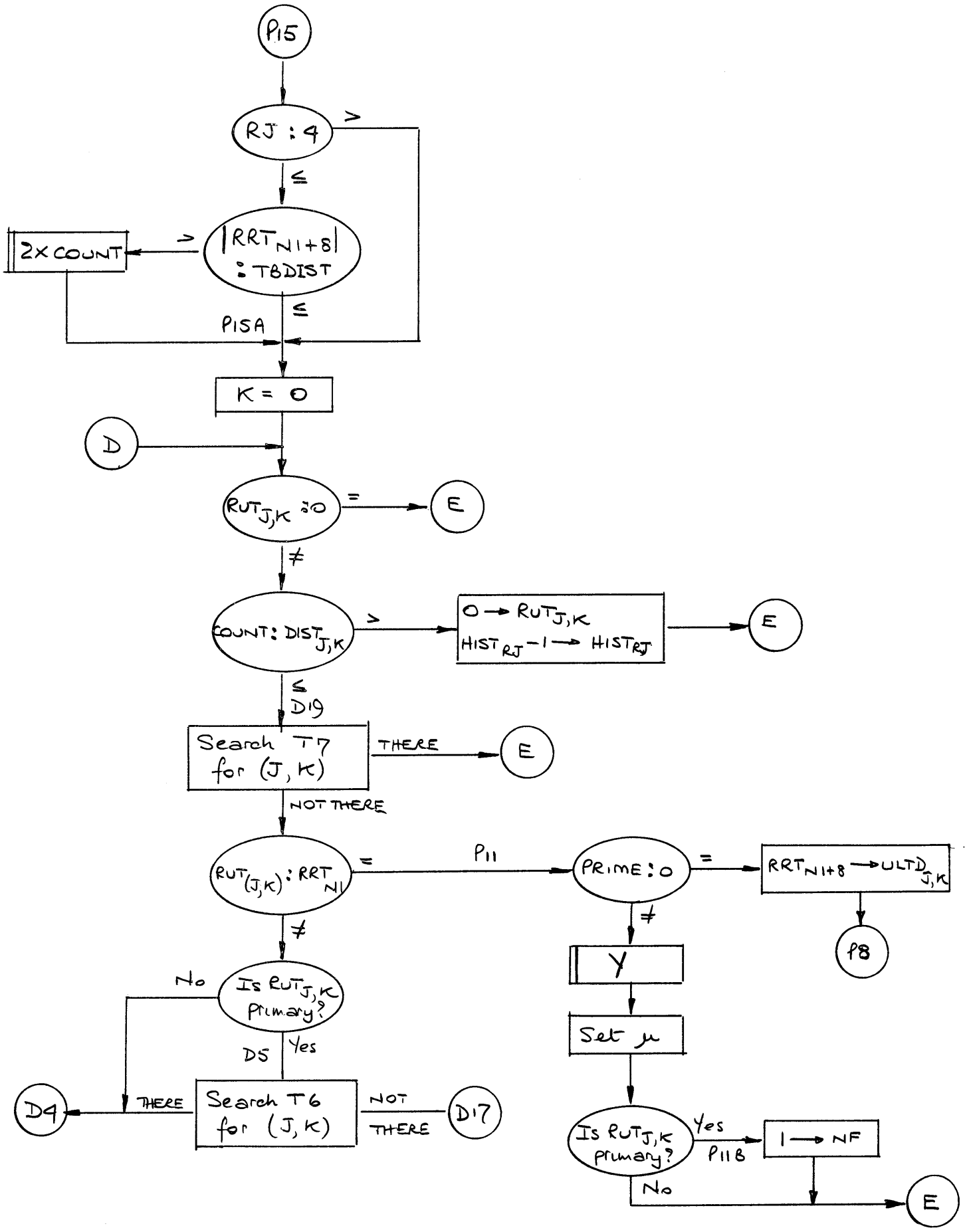
FORWARD SCAN.



FORWARD SCAN

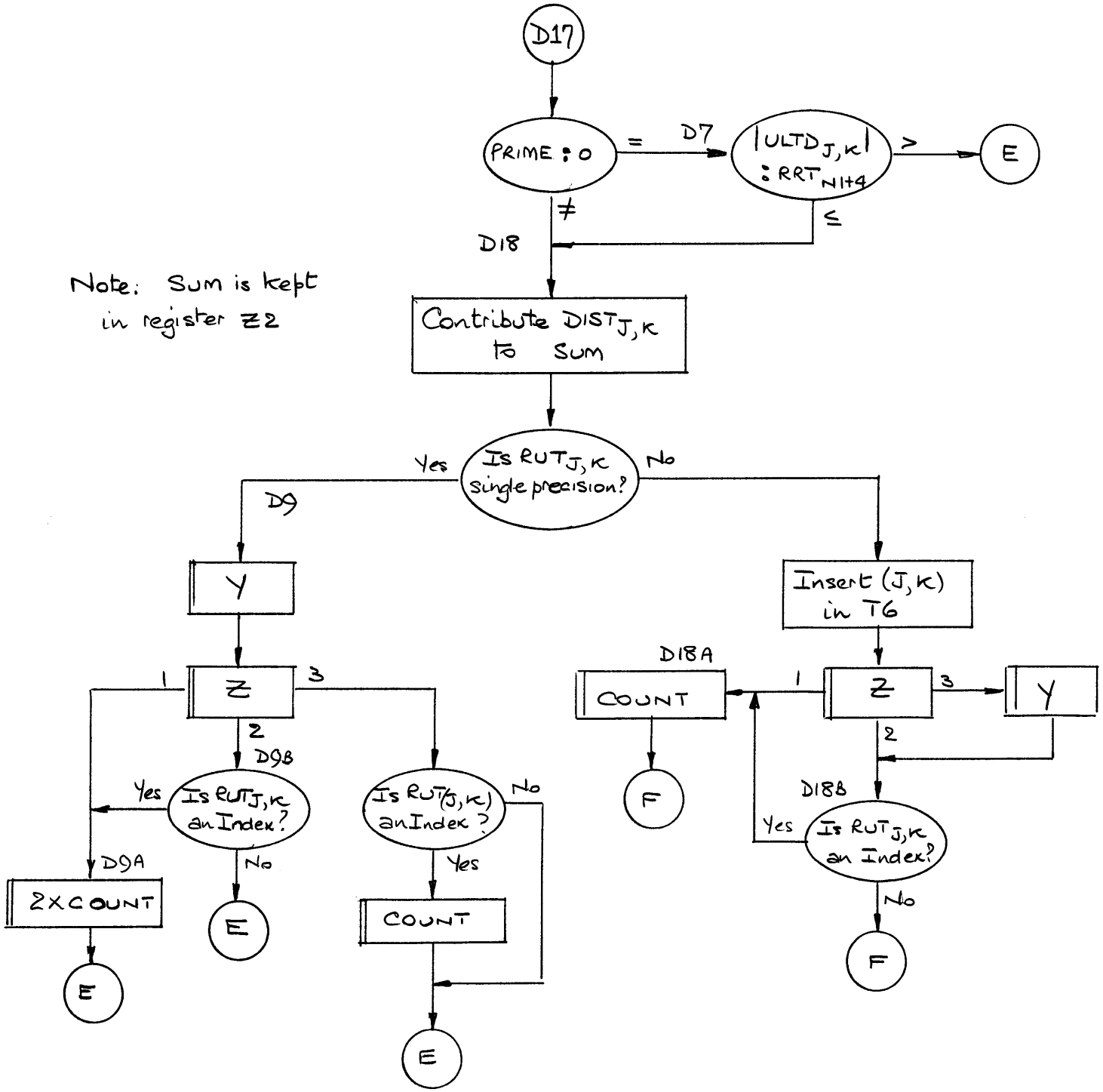


FORWARD SCAN.



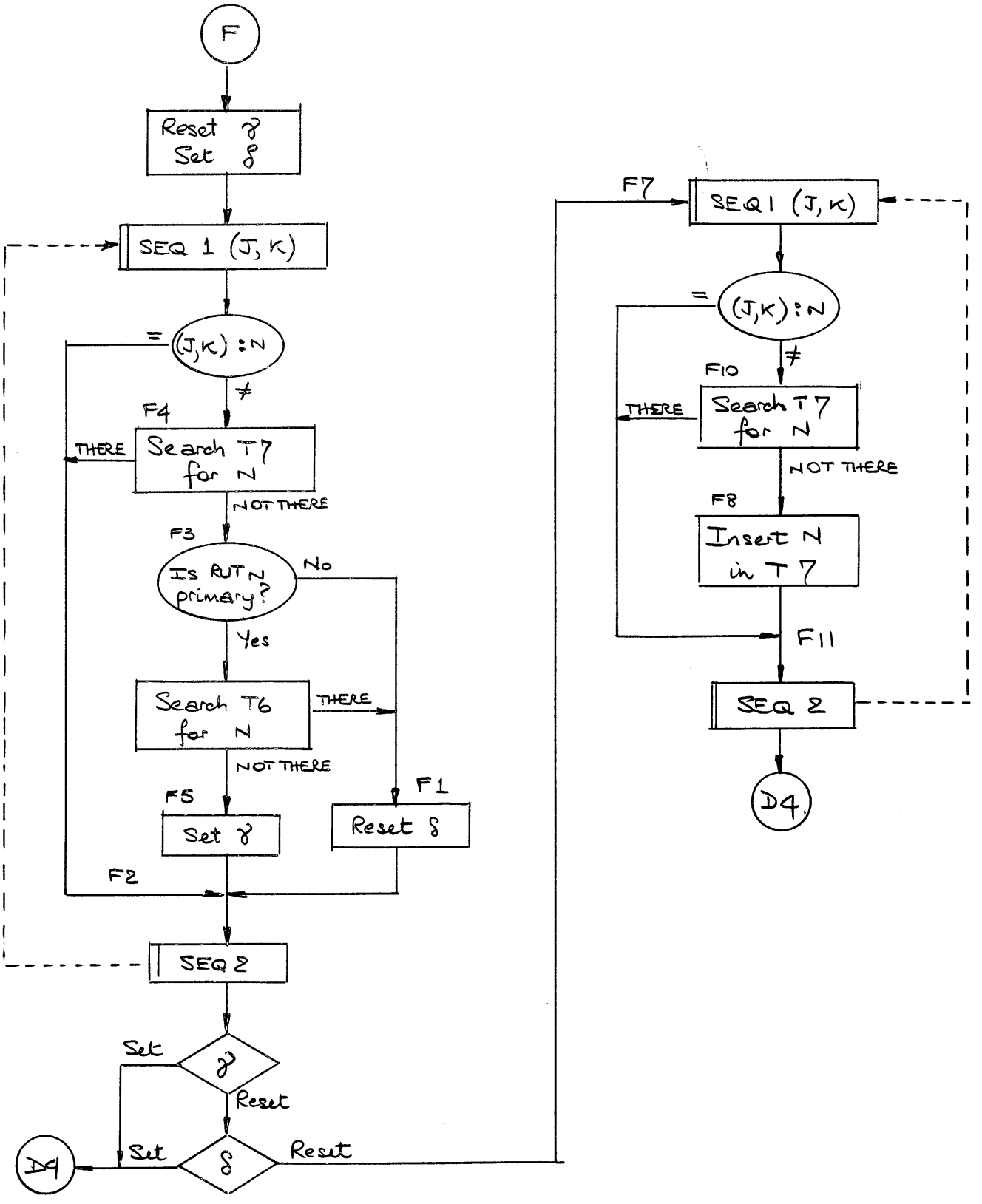
# LARC SCIENTIFIC COMPILER Phase VIII B

## FORWARD SCAN



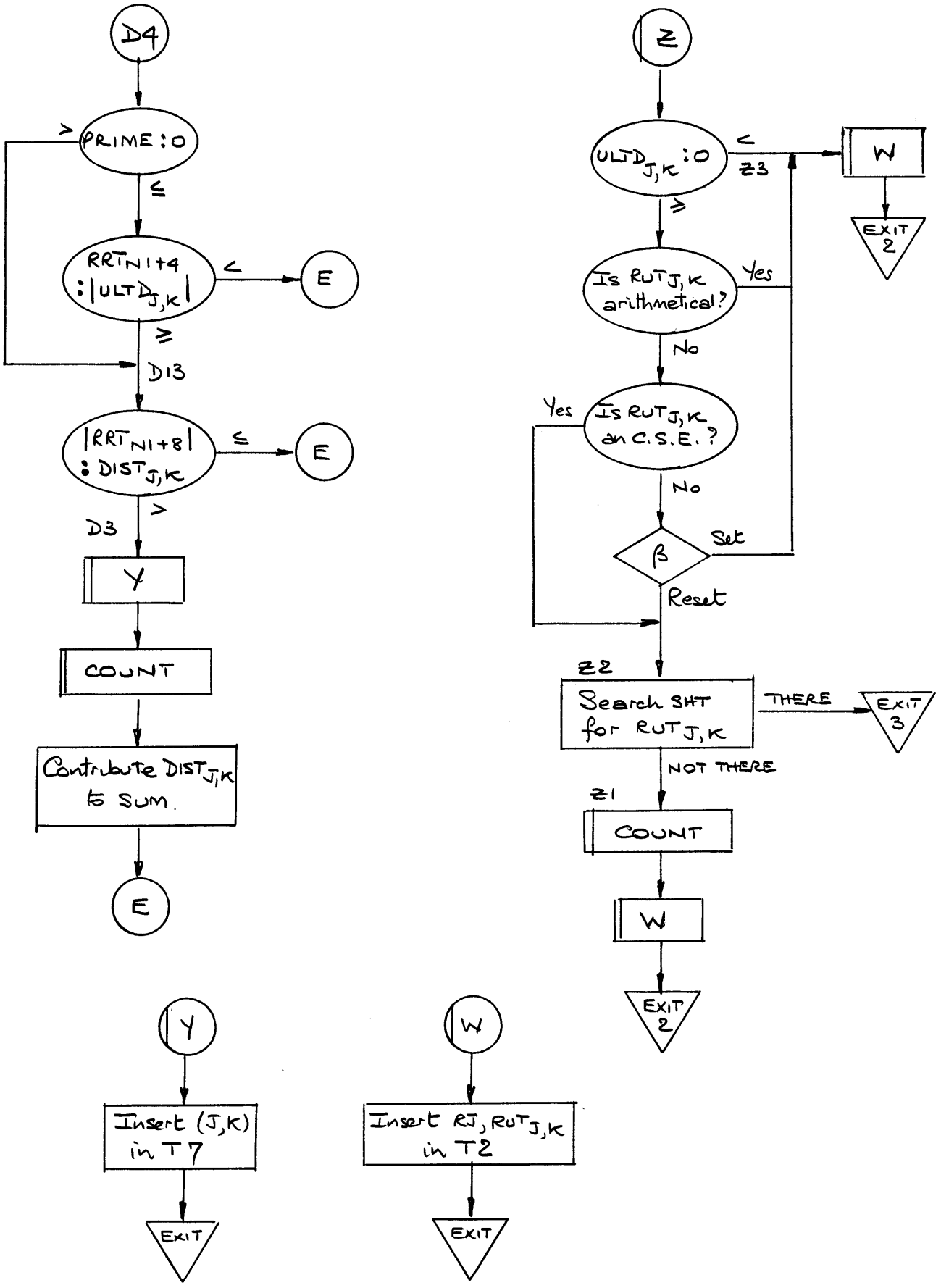
# LARC SCIENTIFIC COMPILER Phase VIII B

## FORWARD SCAN.



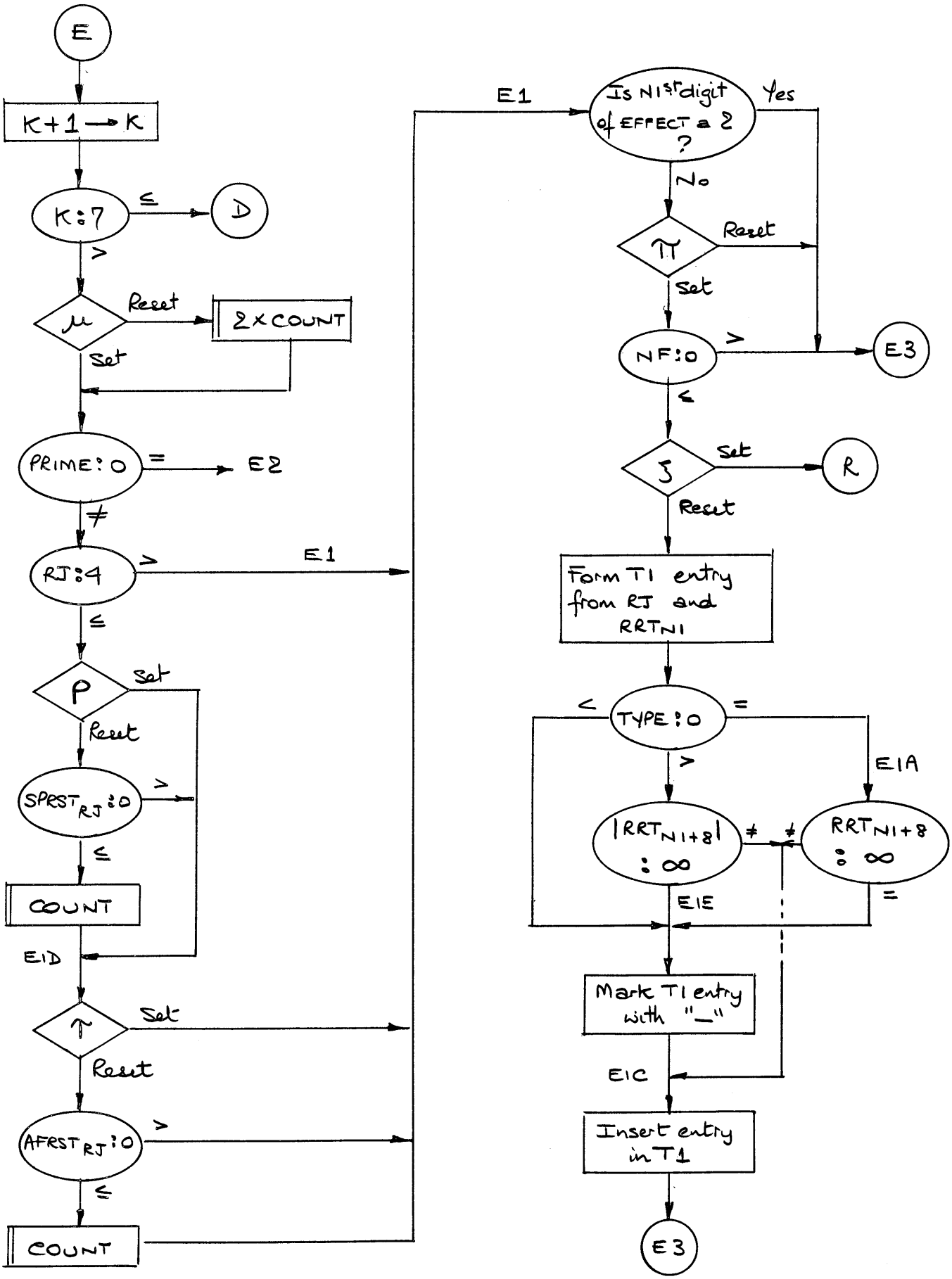
# LARC SCIENTIFIC COMPILER Phase VIII B

## FORWARD SCAN.

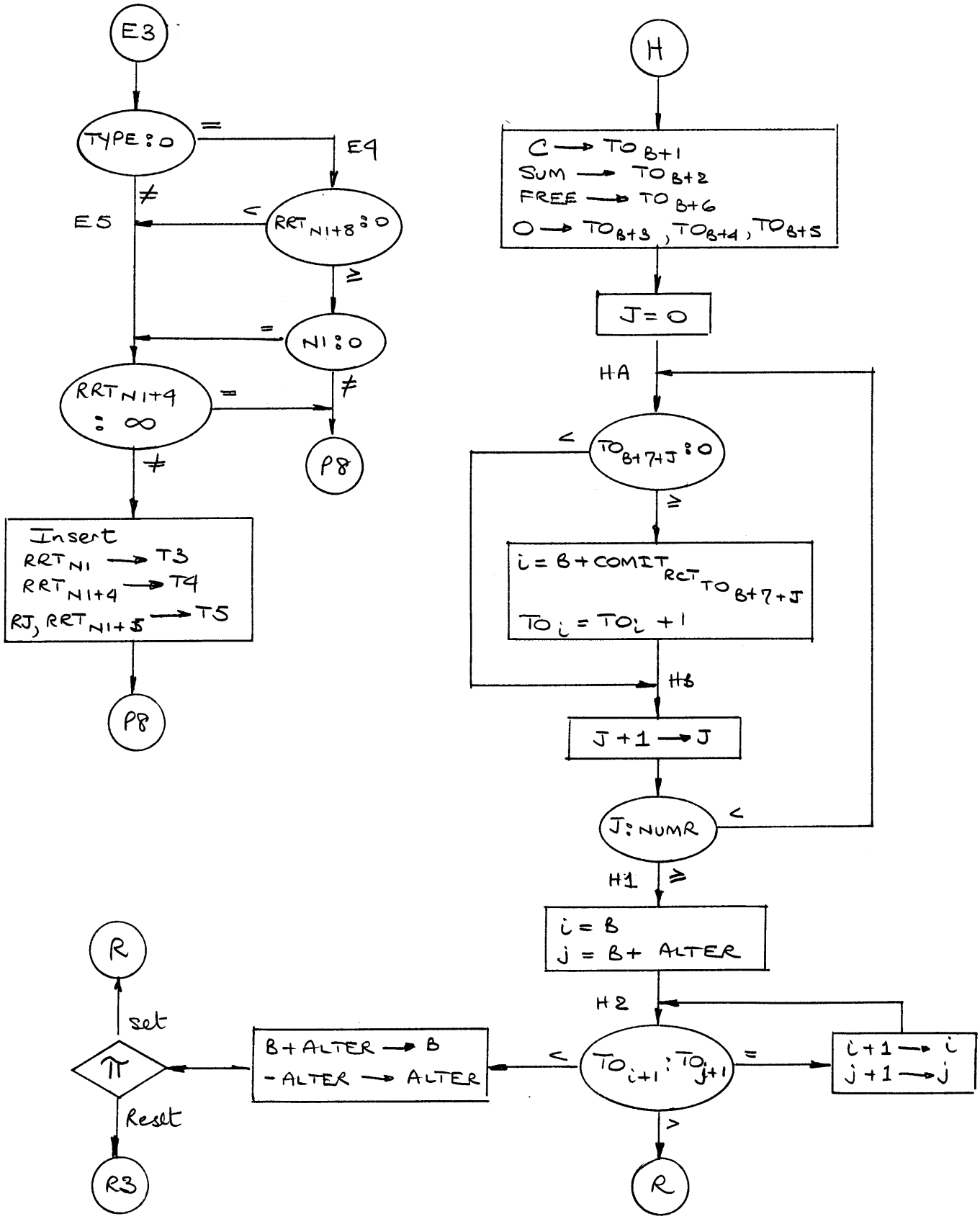


# LARC SCIENTIFIC COMPILER Phase VIIIB

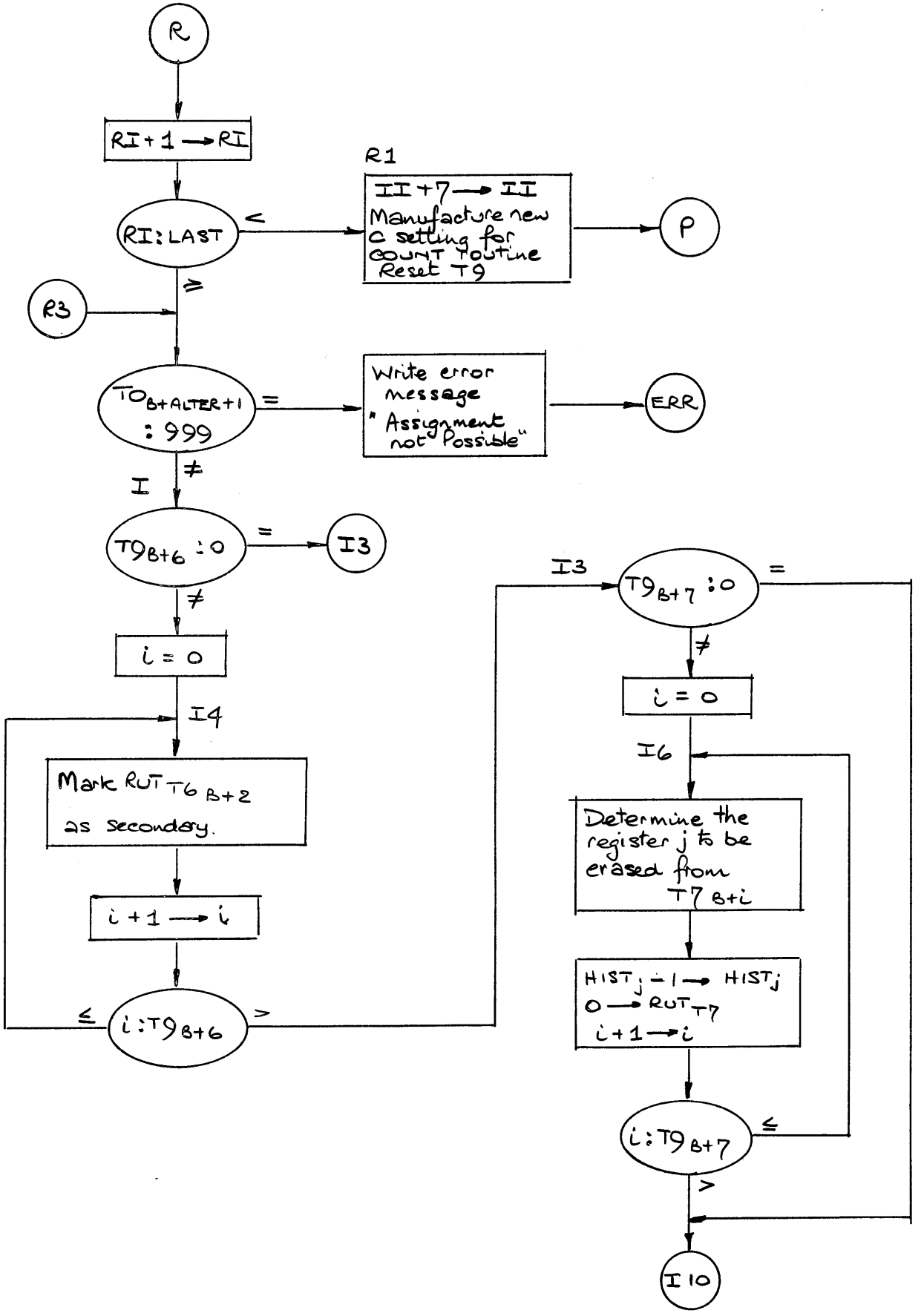
## FORWARD SCAN



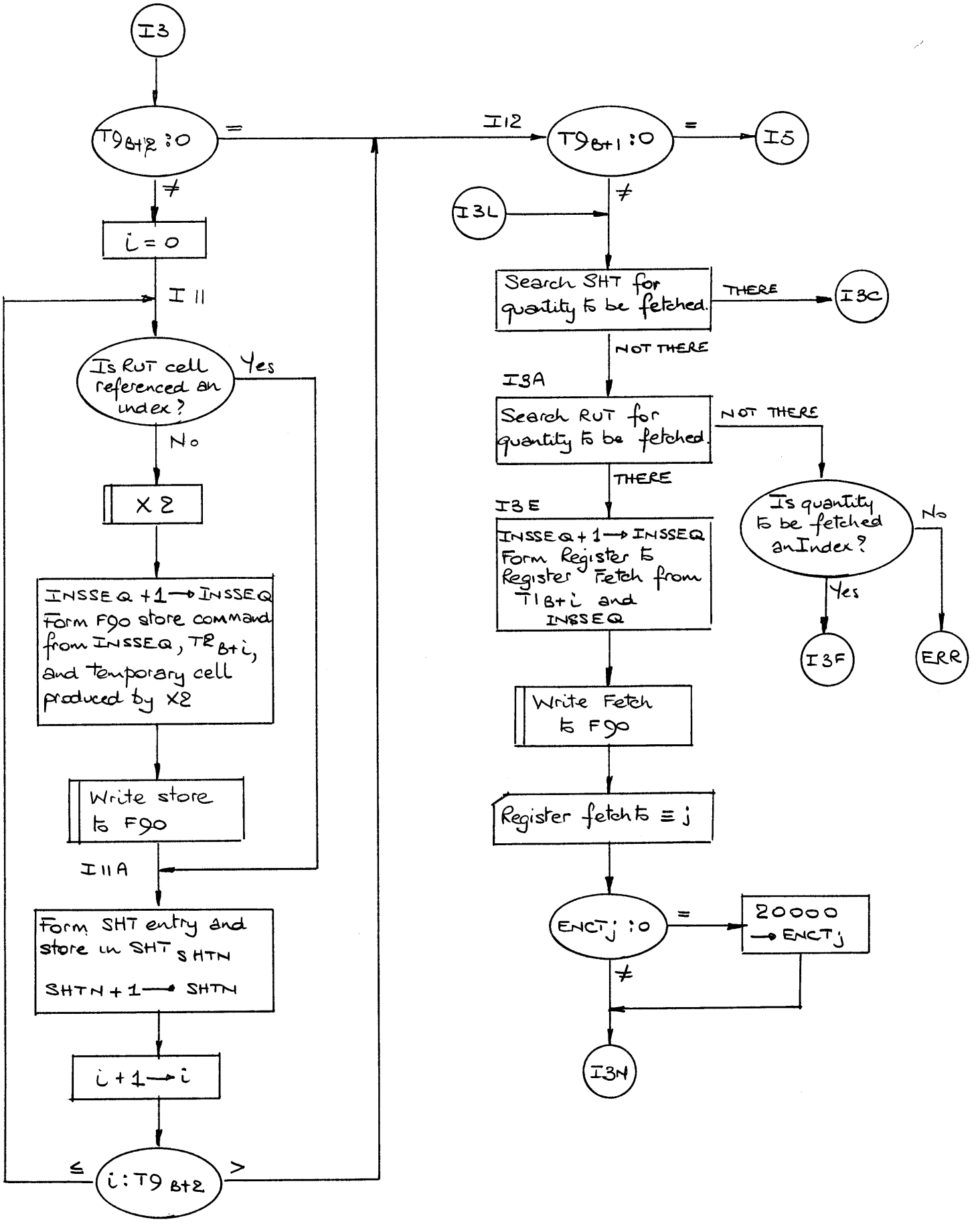
FORWARD SCAN



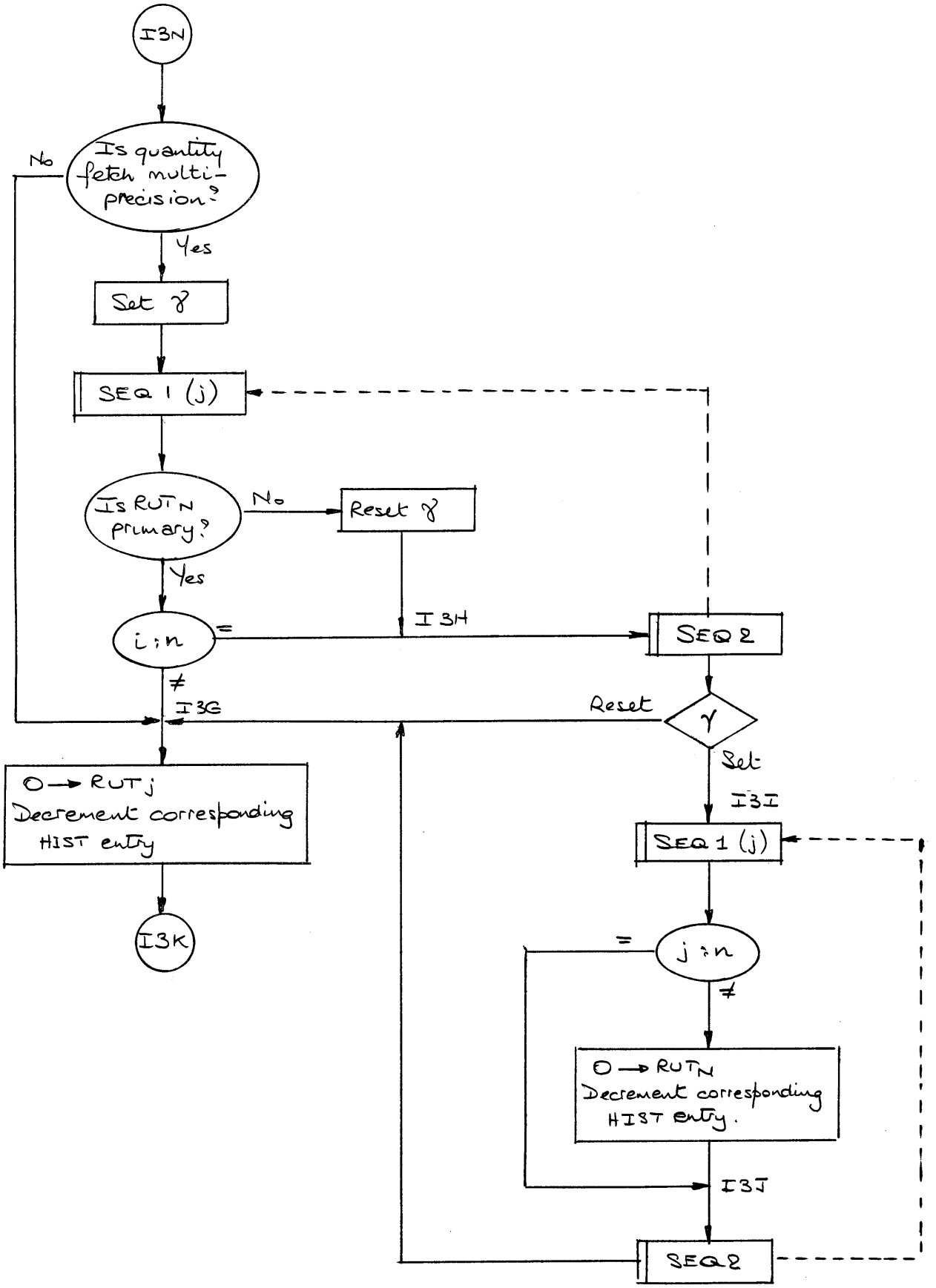
FORWARD SCAN



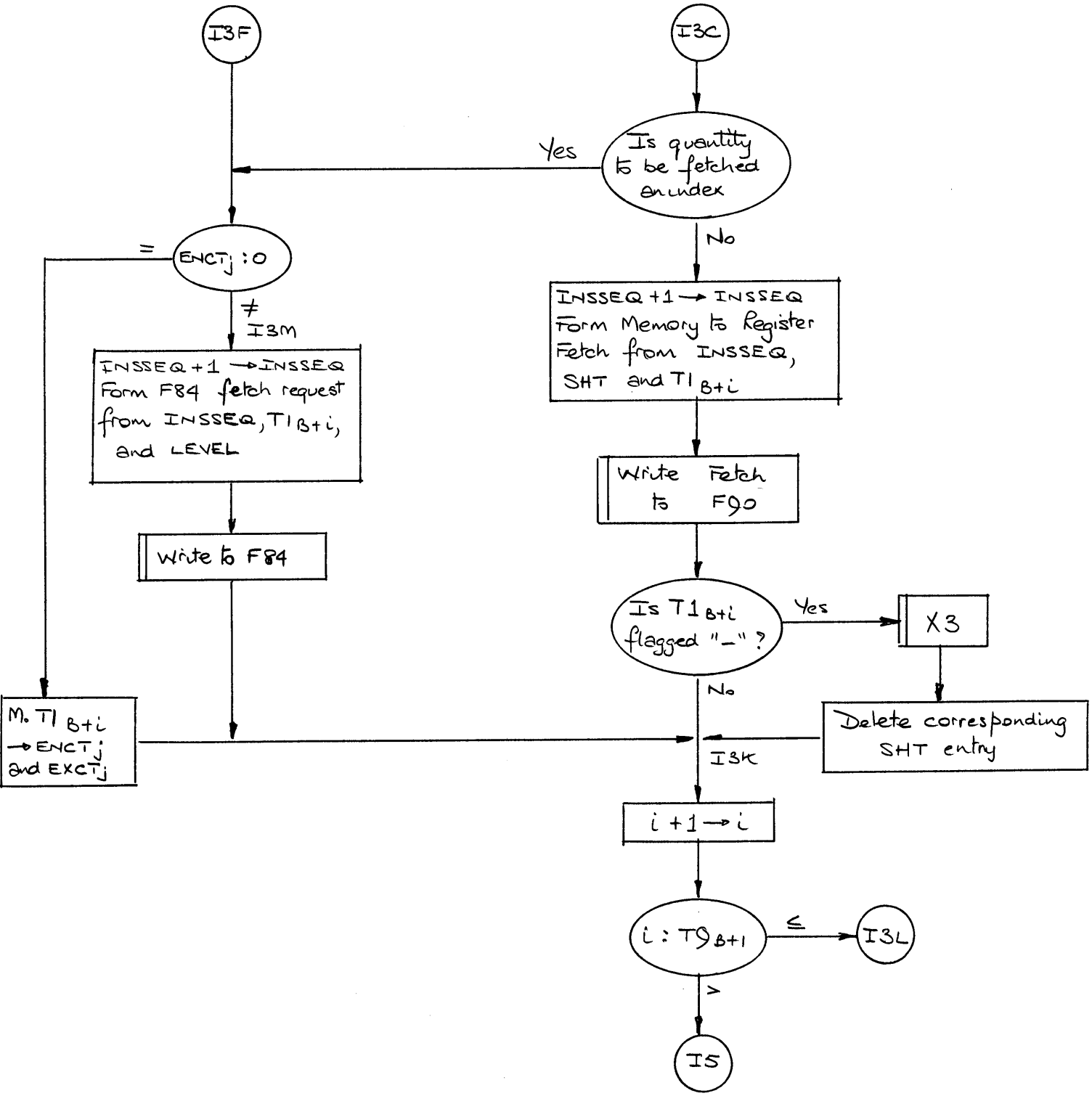
FORWARD SCAN



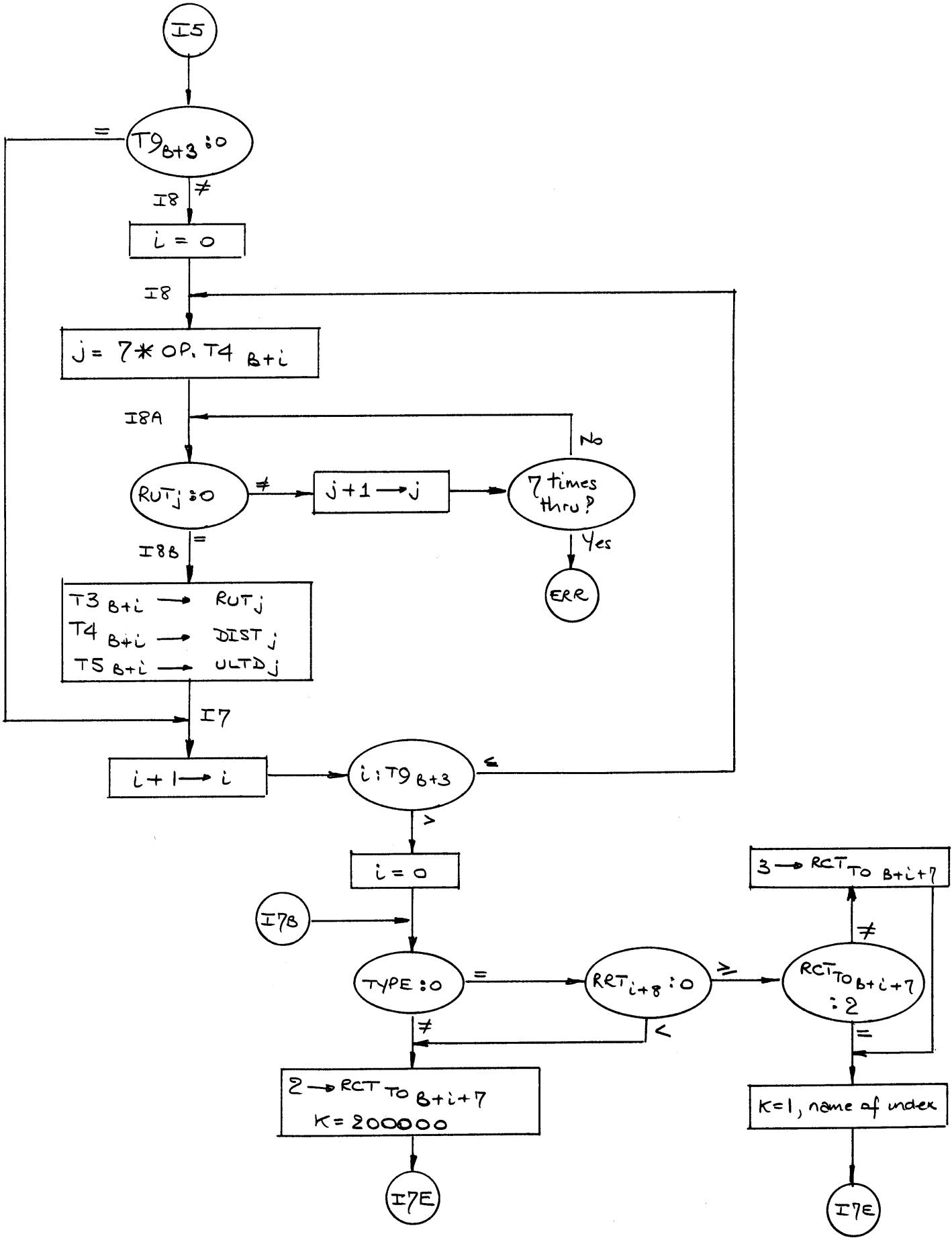
FORWARD SCAN



FORWARD SCAN

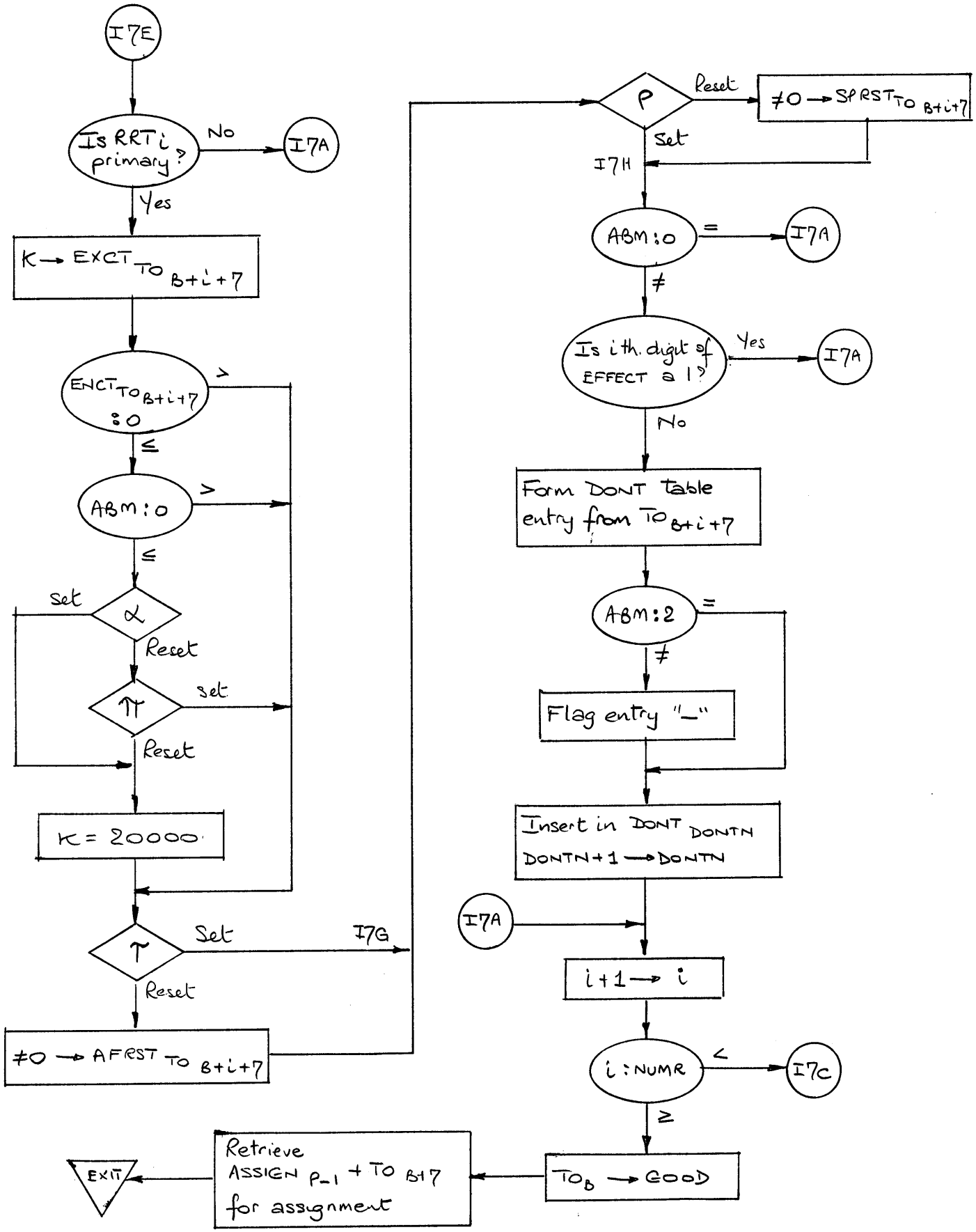


FORWARD SCAN



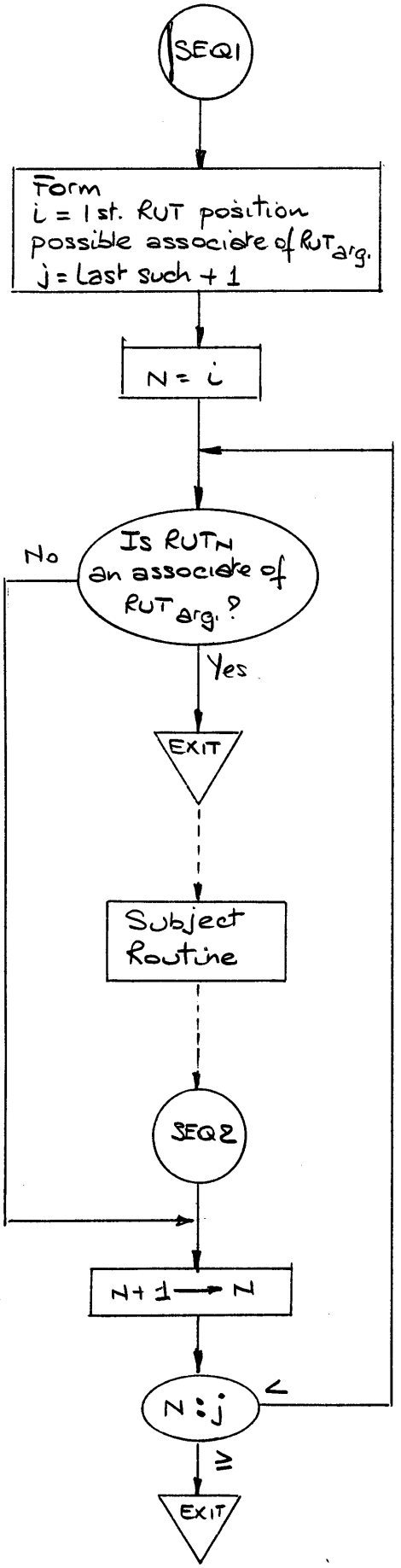
# LARC SCIENTIFIC COMPILER Phase VIIIB

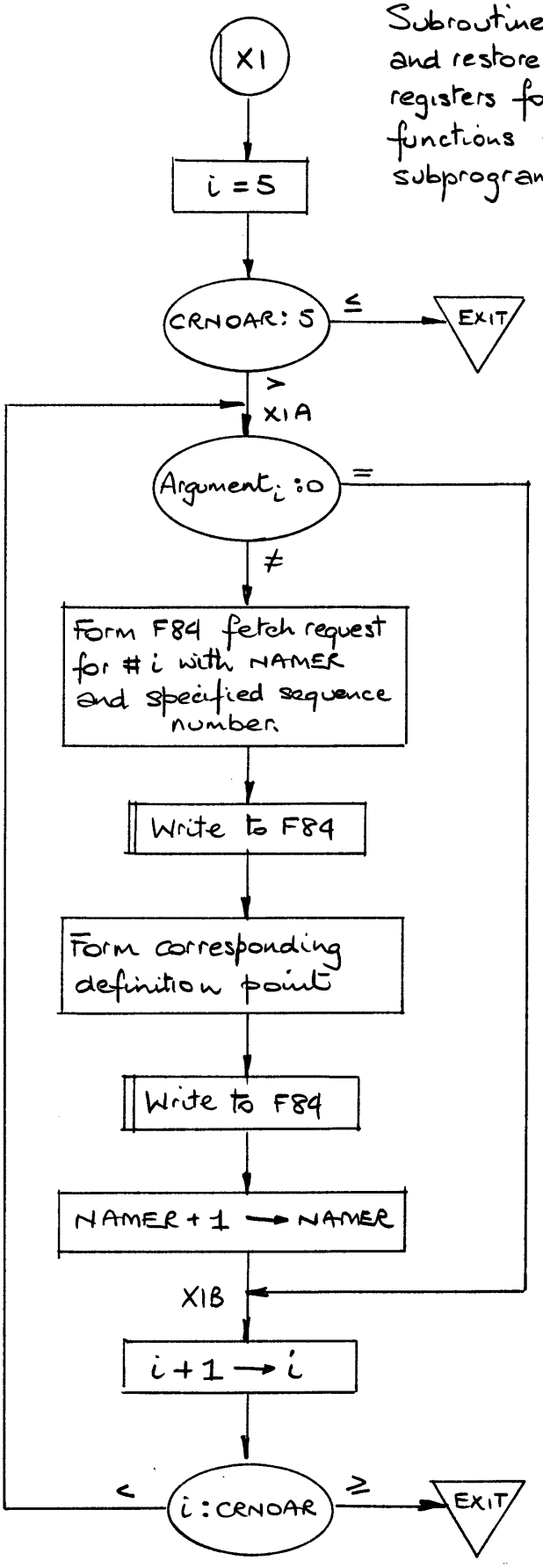
## FORWARD SCAN



# LARC SCIENTIFIC COMPILER Phase VIIIB

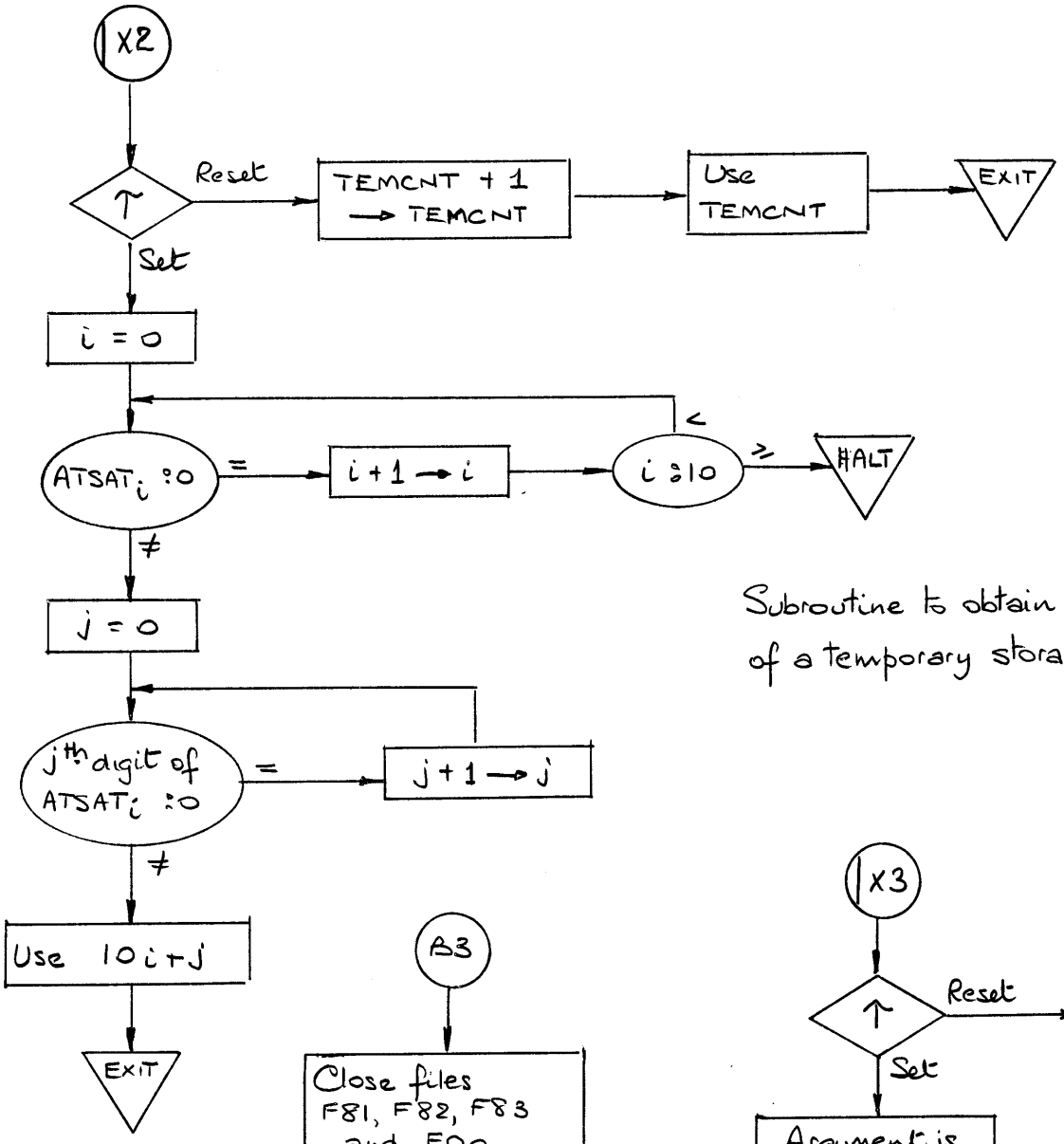
## FORWARD SCAN



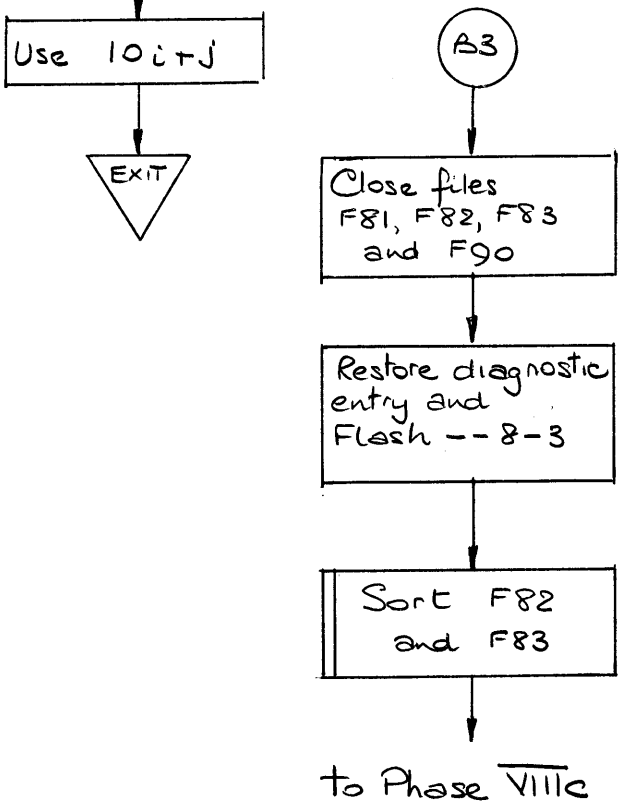




Subroutine to save and restore fast registers for arithmetic functions or subprograms.

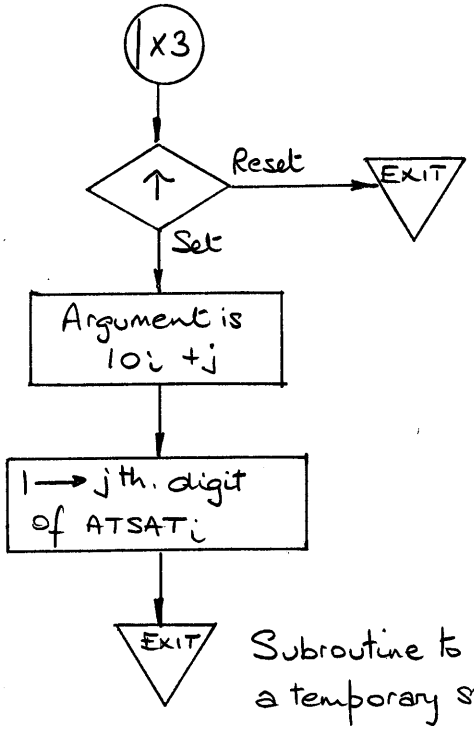
FORWARD SCAN




Subroutine to obtain the name of a temporary storage location.



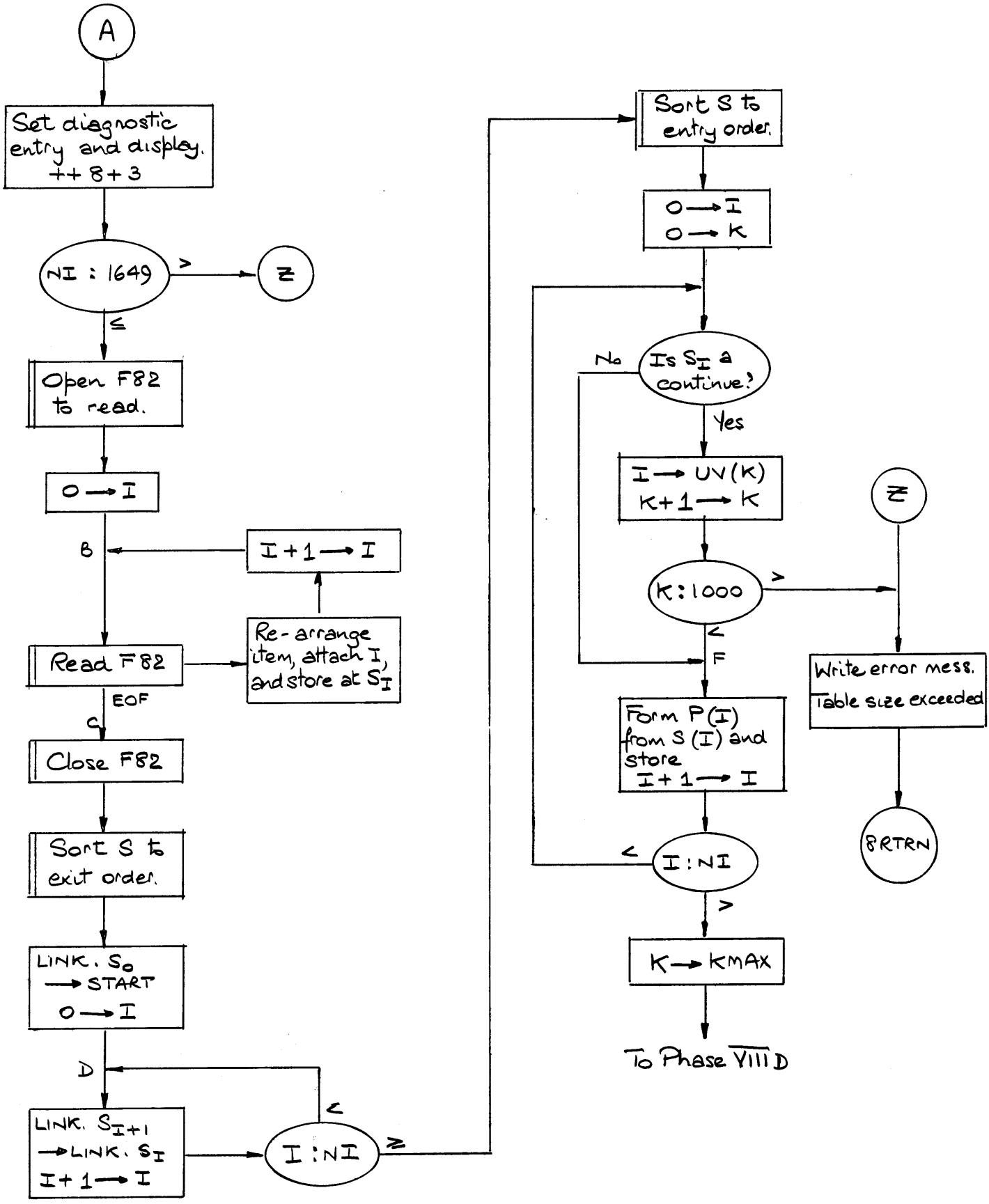

to Phase VIIIC



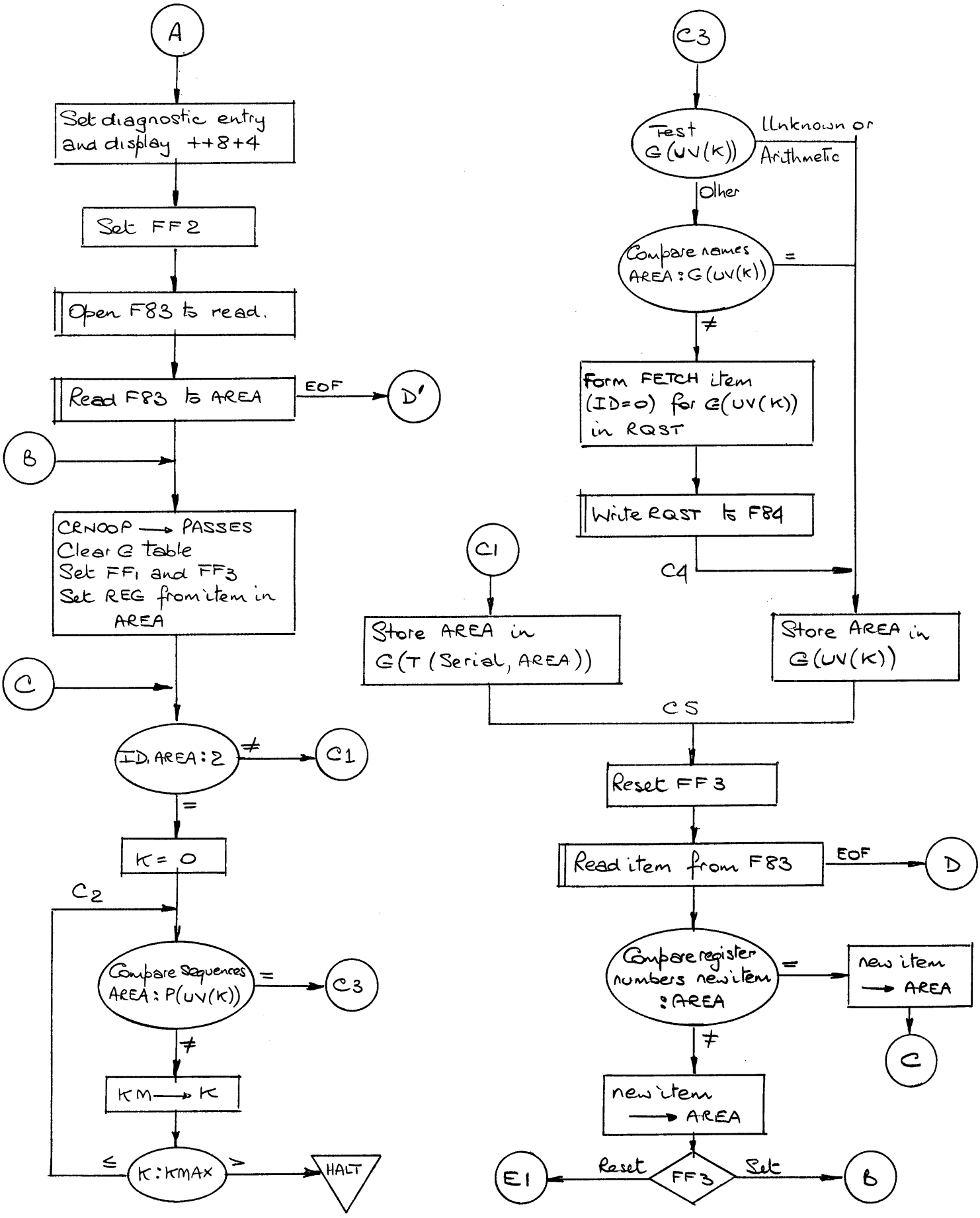

Subroutine to mark a temporary storage location as again available.

# LARC SCIENTIFIC COMPILER Phase VIIIc

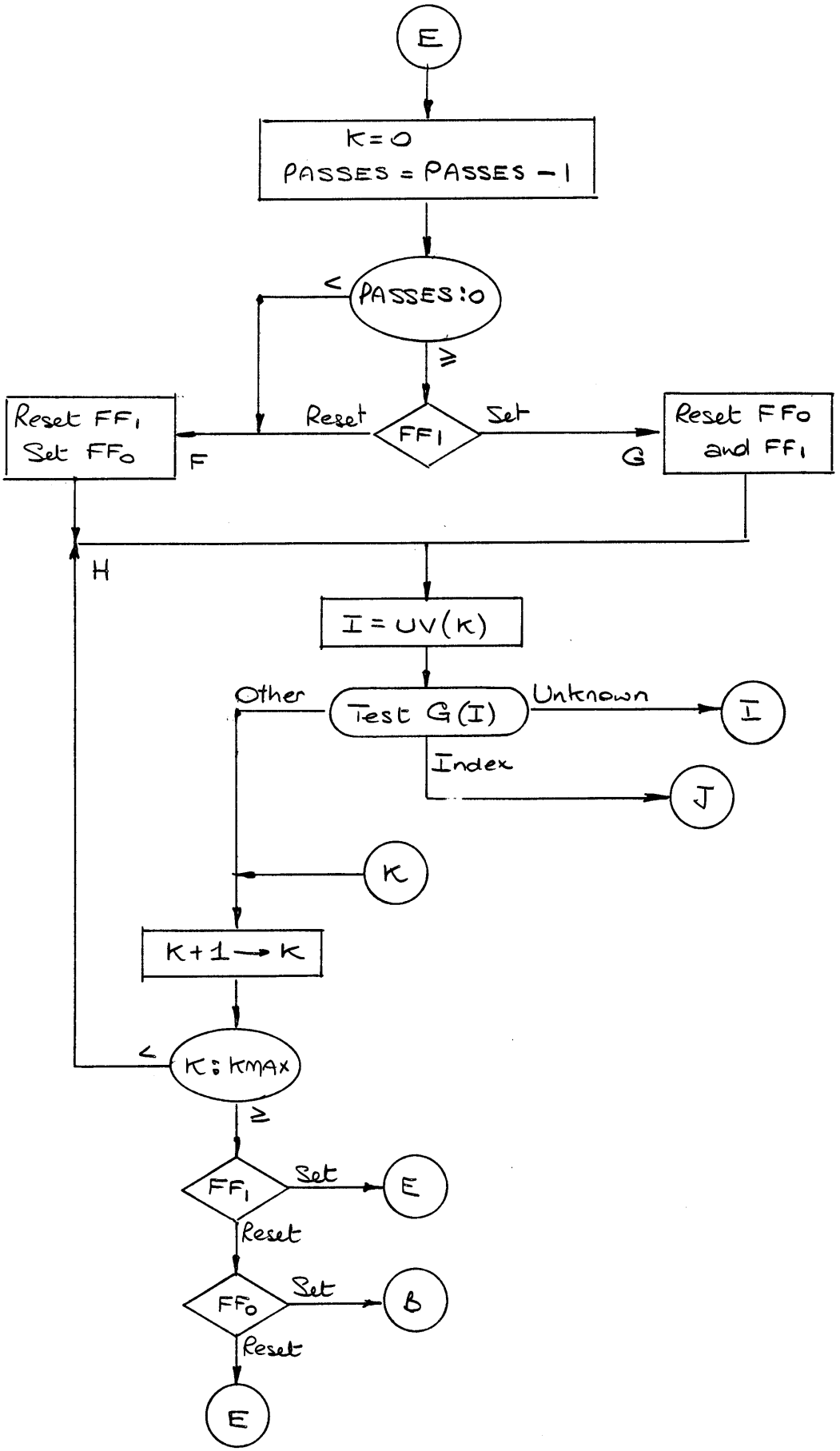
## BUILD FLOW TABLES



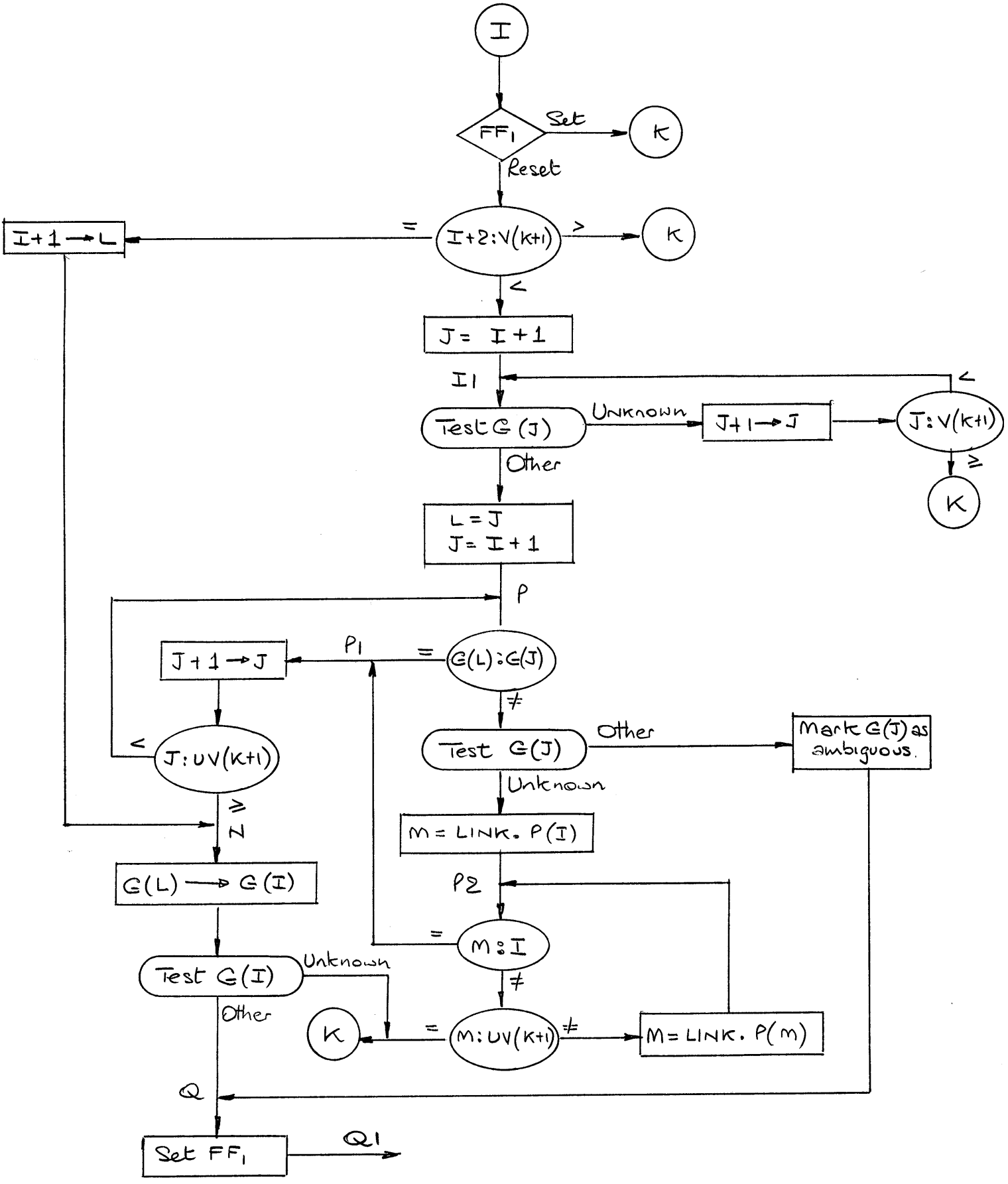
# INDEX ANALYSIS



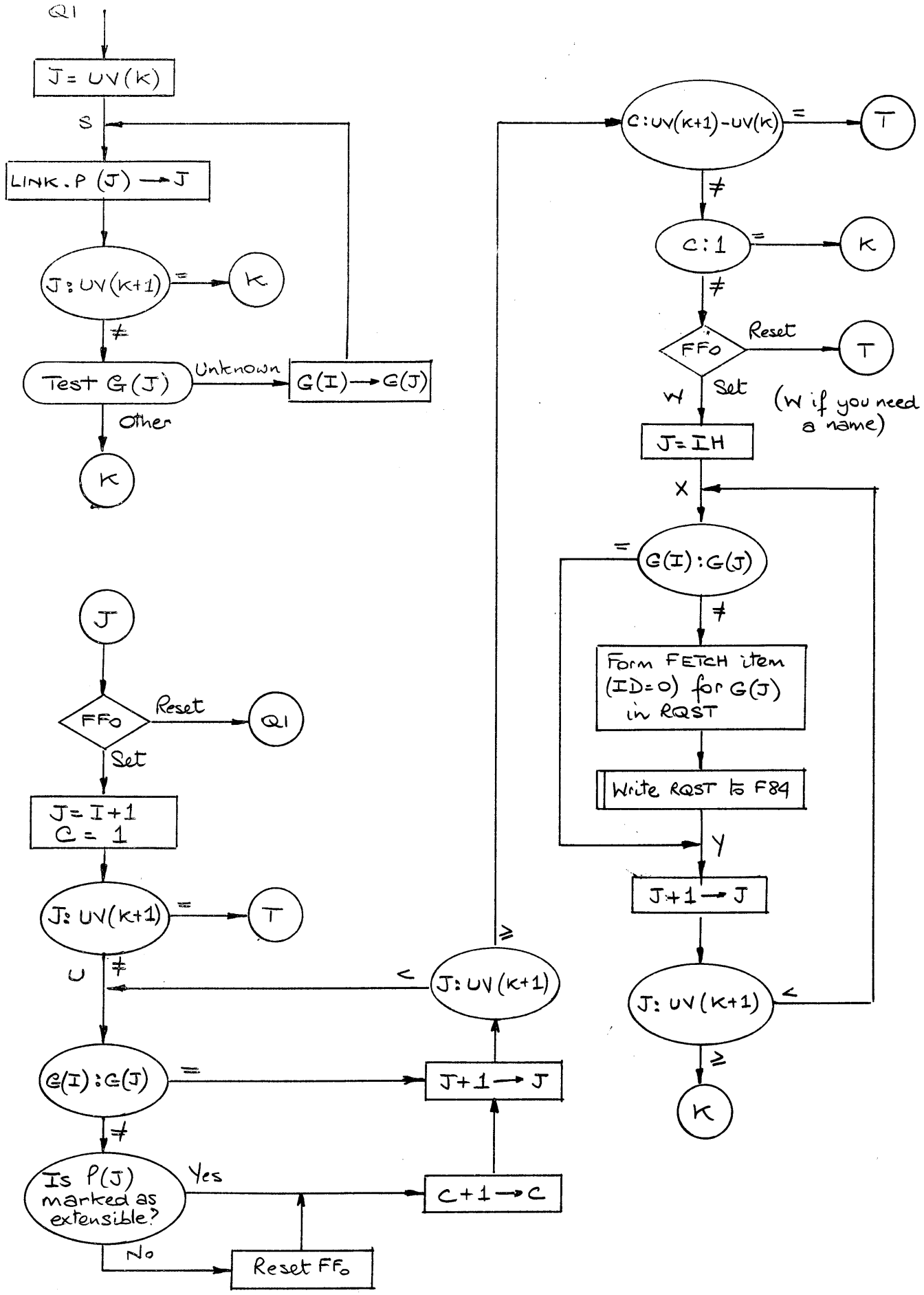
# INDEX ANALYSIS



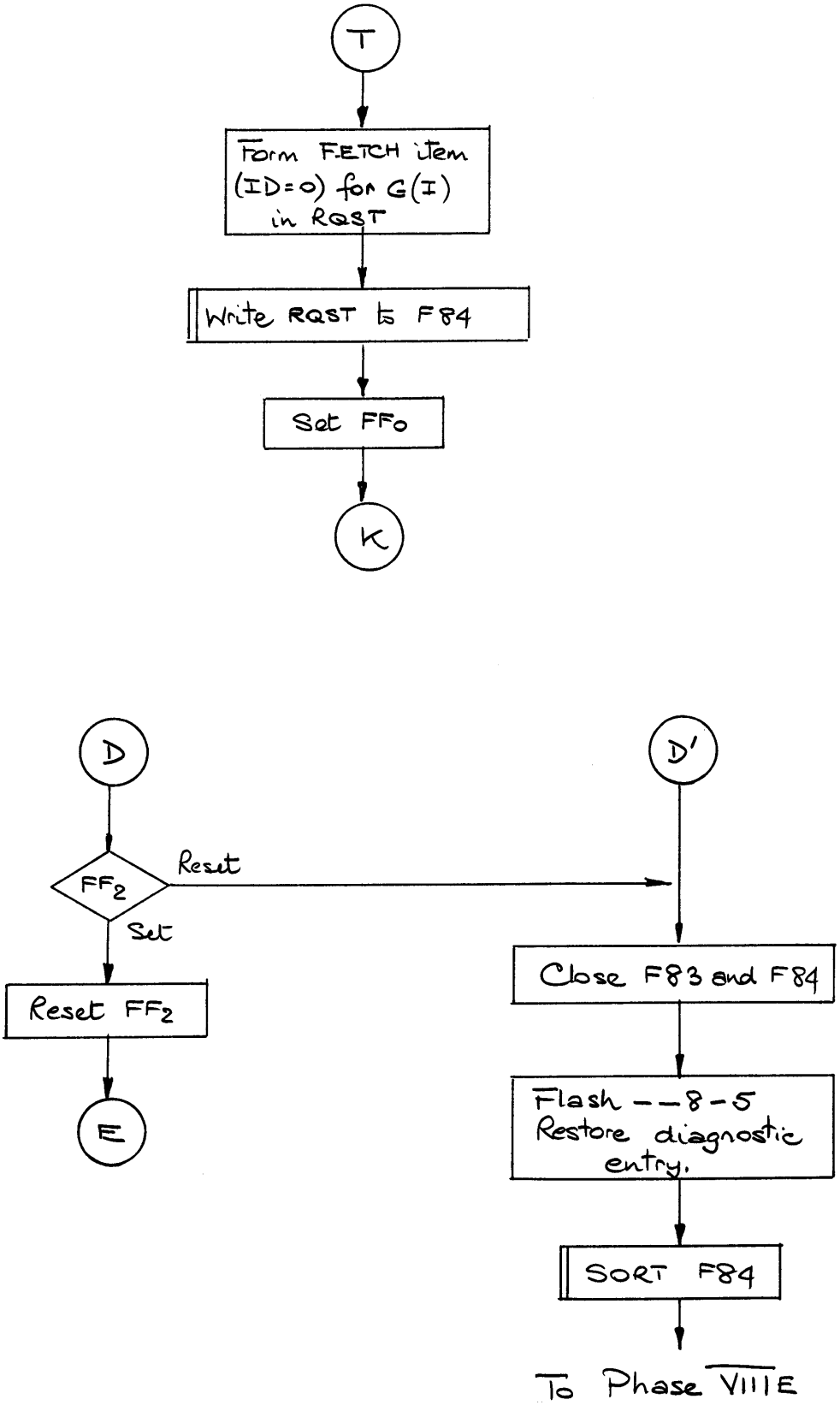
# INDEX ANALYSIS



# INDEX ANALYSIS

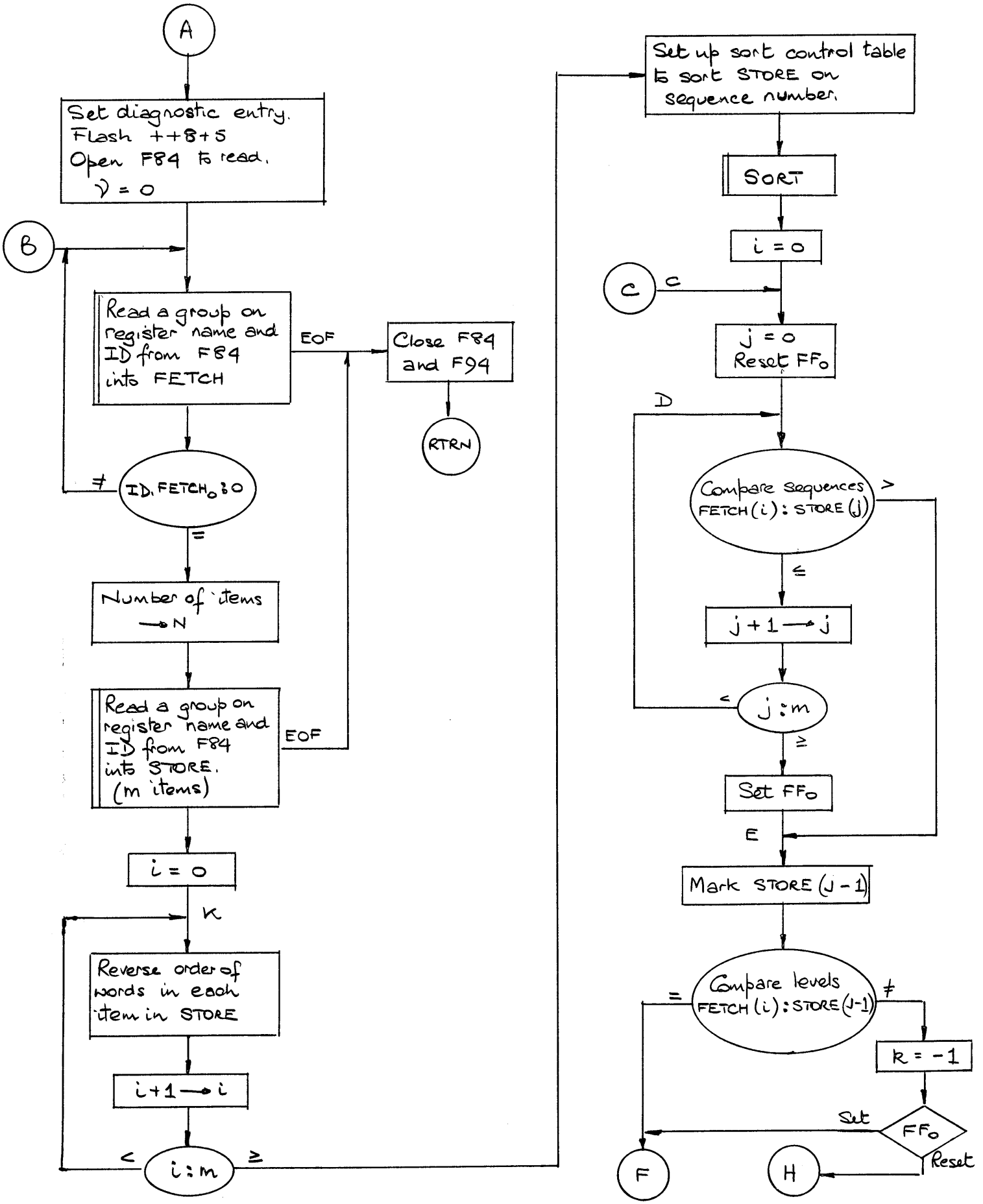


INDEX ANALYSIS

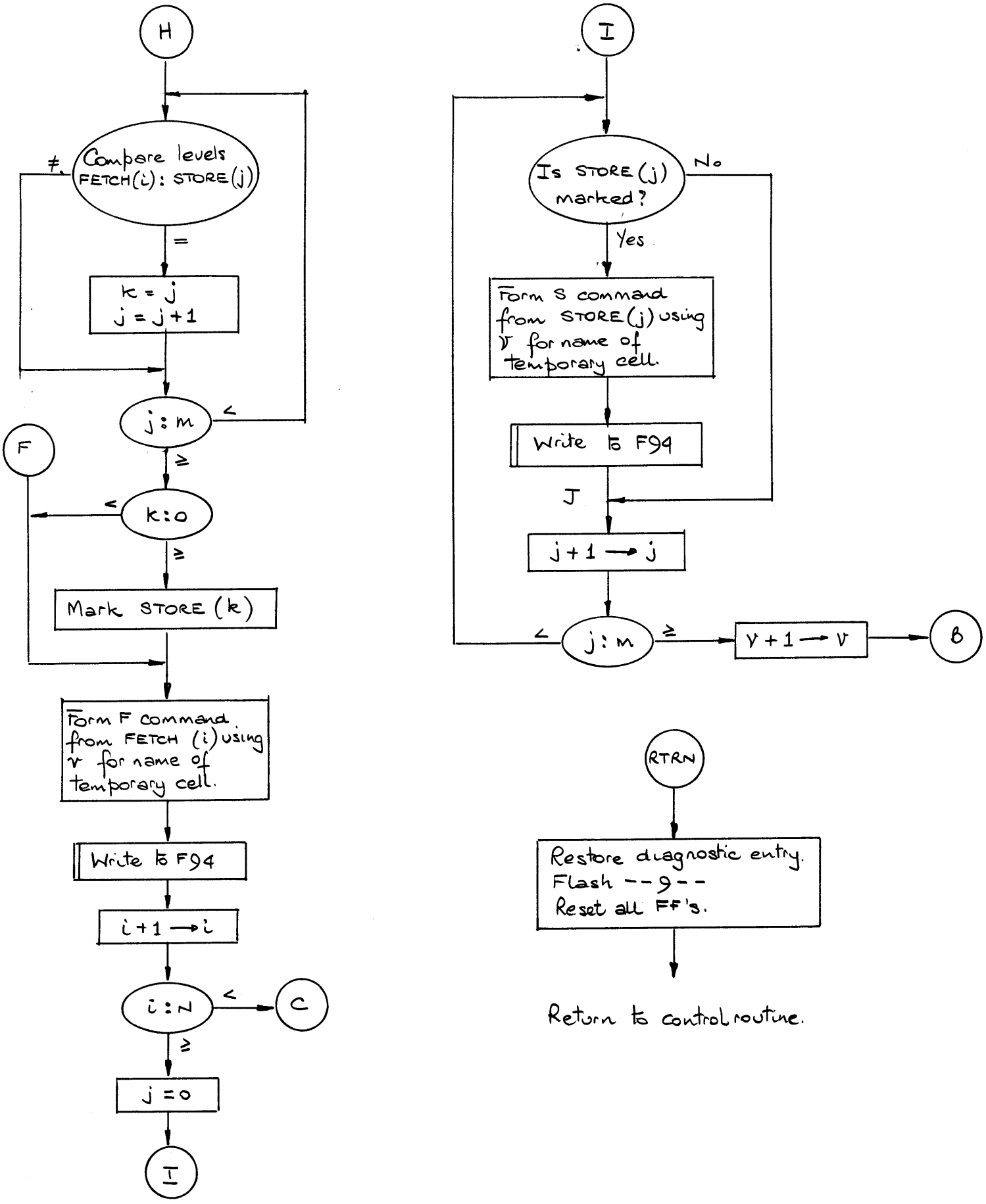


# LARC SCIENTIFIC COMPILER Phase VIIIE

## PRODUCE F & S COMMANDS



PRODUCE F & S COMMANDS



# PHASE IX

## GENERAL DESCRIPTION

Phase IX is the editing phase of LSC; it is divided into two parts. Phase IXa generates error messages from probable source program errors detected by Phases I through XIII. These errors are filed as two word error items in File 92 and the messages are written in File 93. Phase IXb produces the SAL lines from the items in File 90 and File 94 and merges them with the original source statements and error messages, File 93, to create the object program, File 10. The dictionary, File 91, is available throughout Phase IX to provide the parameters for the error messages and the names to replace numeric dictionary references in instruction items.

When the object program has been completely generated, Phase IX writes LSCOMPILED on the typewriter, reinitializes the communication region, and returns to the control program. Refreshing the communication region permits successive compilations without an external restart of LSC.

# PHASE IXa

Phase IXa writes an LSC error diagnostic message in File 93 for each probable source program error detected by Phase I through XIII. These detected errors were written in File 92 as two word items. The first word contains, the phase and error number, the sequence number to be assigned to the error message and an indication that the message does or does not contain a parameter. The second word will contain the parameter or the dictionary reference of the parameter. The error message skeletons which are assembled into Phase IX are located by phase and error number; a % character within the skeleton indicated the position for the parameter, if any. Phase IXa will delete all blanks following the parameter name since the skeleton has the proper number of blanks to obtain the desired spacing. The item written in File 93 is as follows:

| Sequence | F - ER | Diagnostic message |
|---|---|---|
| W1 | W2 | W3 ... W11 |

Detailed descriptions of file 91 and 92 are appended to this section of the manual.

# PHASE IXb

## GENERAL DESCRIPTION

After Phase IXa, File 93 and File 94 are sorted (independently) on sequence. File 93 contains the source program statements, ALPH lines generated by Phase I from Format statements and Hollerith arguments, and the error messages generated by Phase IXa. File 90 and File 94 contains pre-SAL items generated by Phases IV and XIII; these items are to be edited into SAL lines and merged with File 93 according to the assigned sequence numbers to form File 10, the object program.

## OUTPUT OPTIONS

File 10 is written on drum in an area specified by the communication region. The output may also be printed under user option by presetting of two other words in the communication region as follows:

1. CRHSP = 0; no printing

2. CRHSP $\neq$ 0 and CROUT = 0; output on on-line high speed print number one

3. CRHSP $\neq$ 0 and CROUT $\neq$ 0; output on the uniservo specified by CROUT for off-line listing

## DESCRIPTION OF FILE 10

The first line of the object program is a SAL LABEL line. The user may select any name which is acceptable to SAL for his program as long as the first character is alphabetic (to avoid possible conflict with LSC generated symbols). When compiling a SUBROUTINE or FUNCTION, the name of the subprogram will be the symbol on the LABEL line of File 10. For main programs, the symbol will be LSCODE unless

the first line of the source program is a SAL LABEL line; in this case, the name on the LABEL line will be the name of the object program. A directory may also be specified by the user for inclusion in LSC compiled programs, except for SUBROUTINE and FUNCTION. If a directory is to be specified it must appear immediately after the LABEL line, if any, in the source program and conform to SAL specifications, particularly (for recognition by LSC) beginning with DIRECTORY and terminating with ENDIRECTORY. The directory is merely copied by LSC without attempting to detect errors within or initiated by the included directory. In the absence of a directory, the standard SAL directory will be used by the assembler.

For main programs, the next line 10 will be a transfer to the first executable instruction compiled unless the source program contained SAL coding before the first LSC statement which generated executable code. In the later case, the transfer will be to the first SAL statement.

For SUBROUTINE and FUNCTION subprograms, the LABEL is immediately followed by the VARIABLES line (s). The "variables" will be 9CØM, 9TBL, 9STB and all requiried I/O programs, math, library routines, all referenced external functions. These "program variables" should generate LIB instructions for a main program by the operation program. There is no analgous transfer instruction as in main programs, since entry to subprograms is accomplished with a TB instruction to the name. These leading lines are followed by the merged original source statements, comments, and LSC generated instructions and diagnostic messages. The origin of these lines is indicated in columns 1 to 6 of File 10.

| Columns 1 2 3 4 5 6 | origin |
|---|---|
| F | source program LSC statement |
| C | source program comment |
| S | source program SAL line |
| F - ER | LSC error diagnostic message |
| (blank) | LSC generated line |

Each printed line of the object program will contain a % sign followed by the line's twelve digit sequence number; this number is not included in File 10 on drum.

Following the END statement LSC writes an END OF TAPE line in File 10 to signal to SAL the end of the program to be assembled.

A description of and corresponding examples for File 94 and File 90 items appear below.

NAME GENERATION

LSC renames source program names only when they might be internally doubly defined or incompatible with SAL requirements. There are three categories which are renamed.

1. Numeric statement names are appended with L. (e.g., 17 becomes 17L),

2. Special SAL lexicon names (HERE, SAME, SEGii) are prefixed with 9, (e.g., SEG 48 becomes 9SEG48),

3. References to the name of a FUNCTION are treated as references to the name prefixed by 8. (e.g.) FUNCTION PIE (A, PP, LE) . . . could generate:

```
PIE   F   #05   A
      A   #05   PP
      A   #05   LE
      S   #05   8PIE
```

FUNCTION NAME L

There are six categories of names that are generated by LSC. All of them have a numeric as the first character. These categories are:

1. COMMON storage is reserved via the symbol 9COM which is itself defined in one of two ways:

   A. If CRCOMM = 0 then: $STOR 9COM: N

   B. If CRCOMM = 0 then: DEF 9COM: (CRCOMM)

   C. If the program is a SUBPROGRAM, 9 COM is not defined.

2. Subscripted variables are located relative to 9COM or 9DAT by defining them equal to a symbol generated from their sequence number appended with A. e.g., DEF JEAN: 18A - 6 and DEF 18A: 9DAT + 147

3. Generated continues (statement names) are formed by appending X to the sequence of the item to which the name should be attached.

4. Temporary storage locations necessitated by the generated code are named according to type of quantity being stored and a sequence number (denoted as ijk): The types of storage are:

| | | |
|---|---|---|
| A. | Index | $9Xijk |
| B. | Parameter | $9Pijk |
| C. | Arithmetic | $9Tijk |
| D. | Subprogram | $9Mijk or $9Mijk; 2 |

5. Generated instruction literals are named 9ITijk.

e.g., column: 12 17 18 19 60

```
                              F     #13   9IT012
          9IT012        K   TG    #21   JEAN
```

6. A dictionary reference (i.e., name) is created for each argument of a SUBROUTINE, FUNCTION, or ARITHMETIC STATEMENT FUNTION immediately after the name of the program. This name is generated from the dictionary entry number appended by AR.
e.g., SUBROUTINE KSFO(AARON, DON, JIM)

| File 91 entry number | name |
|---|---|
| 93 | KSFO |
| 94 | 94AR |
| 95 | 95AR |
| 96 | 96AR |

## FILE 91 DICTIONARY FILE

File 91 consists of one word items containing one of the following:

1. 12 digits of alphanumeric information representing a given name (left adjusted).

2. The low order part of a double precision numeric quantity in floating point.

3. - Z A A A A A O g g g g

   Z = 0 for positive A A A A A

   Z = 1 for negative A A A A A

   A A A A A represents a numeric quantity to be added or subtracted (according to Z) from the alphanumeric symbol located in File 91 by (gggg)

The (f f f f) digits of the mode word of an item, which has a dictionary reference, will locate one of the above three words. If the sign is "-", then type (3.) above will apply.

FORMATION OF FILE 91:

At the start of each compilation DREF is set equal to 1, prior to filing a word in File 91. The program uses the contents of DREF for the dictionary reference digits (f f f f) in the mode word. The FILE program then advances DREF when the word is filed.

Phase I files names of library programs, I/O subroutines, the low order part of any double precision number, and the generated name of the Hollerith arguments of a CALL statement in File 91. Phase III files a name (in the dictionary) for each

group hence all references to the same name will have the same value (f f f f) in their mode words. Phase V generates a word for subscripted references to variables (format 3. above) whenever there is a numeric quantity ($\pm$ A A A A A) to be added to a symbol for instruction generation. In this case Phase V replaces the references to the name with reference to the word containing the increment and the original File 91 reference.

FILE 92 ERROR ITEMS FILE

Each LSC phase will write any detected error as a two word item in File 92. These items will generate error messages in File 93 during Phase IXa.

File 92 Item Format

| | | | Word 1 | | Word 2 |
|---|---|---|---|---|---|
| P | E | A | B | S | W |
| 2 | 3 | 1 | 1 | 5 | |

P: Number of the phase that produces error item.

E: Error item number

S: Sequence number of the item that generates the error.

W: Parameter for error messages (if there is one).

A: Determines mode of W as follows:

| A | Mode of W |
|---|---|
| 0 | alphanumeric (left adjusted) |
| 1 | numeric; use W as name dictionary reference to get error message parameter |

B: Degrees of fatality of error.

B = 0, not so serious

B = 1, serious source program error

The error messages themselves are in sort on the phase and error number (P and E), they have the following form:

| P | E | C | (Message) | % | (rest of Message) |
|---|---|---|---|---|---|
| W1 | | | W2 | ——— | W10 |

The entire item has 10 words.

C = 0, there is no parameter

C = 1, there is a parameter

The "%" character is in the position in the message where the parameter is to be placed (if there is one).

STORAGE RESERVATION AND DEFINITION ITEMS.

The file 10 items generated by these file 94 items are best described by examples of the original items and the generated SAL lines.
Assume that a portion of the dictionary (file 91) is :

| Entry number | Name. |
|---|---|
| 89 | JIM |
| 90 | WES |
| 91 | JACK |
| 92 | KEN |
| 93 | BOB |
| 94 | TED |
| 95 | ERV |
| 96 | 035697932385 |
| 97 | PI |

ID F94 item F10 item

02

| W1 | Sequence |
|---|---|
| W2 | 0 —— 002 |
| W3 | 0 —— 089 |
| W4 | 0 —— 06 |

DEF JIM : 89A - 6

03

| W1 | Sequence |
|---|---|
| W2 | 0 —— 003 |
| W3 | 0 —— 089 |
| W4 | 0 —— 014 |

DEF 89A : 9DAT + 14

04
05

| W1 | Sequence |
|---|---|
| W2 | 0 —— 004 |
| W3 | 0 —— 090 |
| W4 | 0 —— 017 |

DEF 90A : 9COM + 17

STORAGE RESERVATION AND DEFINITION ITEMS (continued)

ID F94 item F10 item

06

| W1 | Sequence |
|---|---|
| W2 | 0 —— 006 |
| W3 | 0 —— 091 |
| W4 | 0 —— 019 |

DEF JACK : 9DAT + 19

07

| W1 | Sequence |
|---|---|
| W2 | 0 —— 007 |
| W3 | 0 —— 092 |
| W4 | 0 —— 023 |

DEF KEN : 9COM + 23

08

| W1 | 0 —— 0900 |
|---|---|
| W2 | 0 —— 008 |
| W3 | 0 —— 0 |
| W4 | 0 —— 083 |

$STOR 9DAT ; 83

09

| W1 | Sequence |
|---|---|
| W2 | 0 —— 009 |
| W3 | 0 —— 094 |
| W4 | 0 —— 017 |

DEF TED : 17

10

| W1 | 0 —— 0800 |
|---|---|
| W2 | 0 —— 010 |
| W3 | 0 —— 0 |
| W4 | 0 —— 077 |

i) if (CRCOMM ≠ 0) DEF 9COM : 20000
(20000 for example)

ii) if (CRCOMM = 0) $STOR 9COM ; 77

STORAGE RESERVATON AND DEFINITION ITEMS (continued)

ID F94 item F10 item

col. 12 17 18 19

ERV K - 1 2 3

11

| | |
|---|---|
| W1 | sequence |
| W2 | a 1 c d e e e 00095 |
| W3 | -0 0123 |
| W4 | 0 ——— 0 |

WES K (0.112358132 (08))

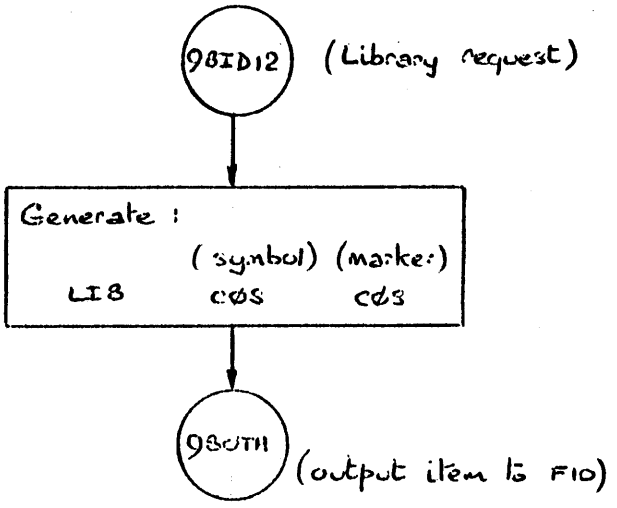
11

| | |
|---|---|
| W1 | sequence |
| W2 | a 2 c d e e e 00090 |
| W3 | 0 5 8 1 1 2 3 5 8 1 3 2 |
| W4 | 0 ——— 0 |

PI K (0..3141592653589793238 5 (01))

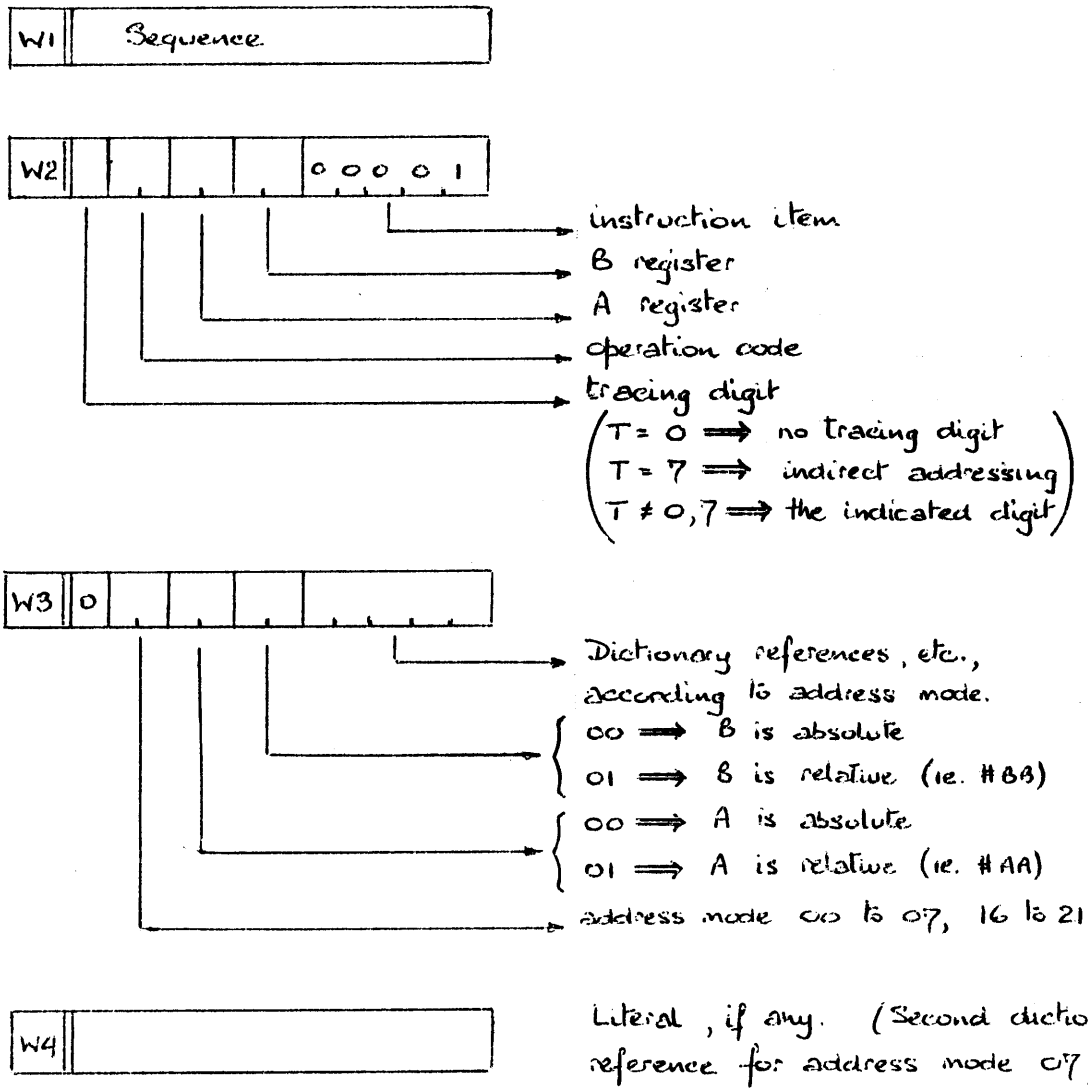
11

| | |
|---|---|
| W1 | sequence |
| W2 | a 3 c d e e e 00097 |
| W3 | 0 5 1 3 1 4 1 5 9 2 6 5 |
| W4 | 0 3 5 8 9 7 9 3 2 3 8 5 |



## Instruction Items

Instruction items in File 90 and 94 are four word items with the following general format.




The following examples assume that a portion of the dictionary (File 91) is:

| ENTRY | NAME |
|---|---|
| 146 | TRUCK |
| 147 | HI7 |
| 148 | CØS |
| 149 | EBB |
| 150 | FLO |
| 151 | .00000000000 |
| 152 | -10110000147 |

INSTRUCTION ITEMS

| Address Mode | F90 or F94 item | F10 item. |
|---|---|---|
| absolute | W1 Sequence<br>W2 0 93 00 00 00001<br>W3 0 00 00 00 23461<br>W4 0 ——— 0 | SLJ 23461 |
| relative forward | W1 Sequence<br>W2 0 70 06 00 00001<br>W3 0 01 01 00 00002<br>W4 0 ——— 0 | TE #06 HERE+2 |
| relative backward | W1 Sequence<br>W2 0 80 12 00 00001<br>W3 0 02 01 00 00017<br>W4 0 ——— 0 | BIT #12 HERE-17 |
| Symbolic | W1 Sequence<br>W2 7 48 08 22 00001<br>W3 0 03 01 00 00146<br>W4 0 ——— 0 | FF #08 TRUCK 12 X |
| symbolic | W1 Sequence<br>W2 0 23 03 03 00001<br>W3 0 03 00 01 00147<br>W4 0 ——— 0 | M 03 HI7-1100 #03 |
| symbolic | W1 Sequence<br>W2 0 90 00 01 00001<br>W3 0 03 00 01 20014<br>W4 0 ——— 0 | T 20014 X #01 |

INSTRUCTION ITEMS (continued)

| Address mode | F90 or F94 item | F10 item. |
|---|---|---|

temporary (parameter)

| W1 | Sequence |
|---|---|
| W2 | 0 40 16 00 00001 |
| W3 | 0 04 01 00 00000 |
| W4 | 0 ——— 0 |

S #16 $P0

temporary

| W1 | Sequence |
|---|---|
| W2 | 0 43 10 06 00001 |
| W3 | 0 04 00 00 10211 |
| W4 | 0 ——— 0 |

F 10 $9T211 06

temporary (arithmetic)

| W1 | Sequence |
|---|---|
| W2 | 0 40 04 00 00001 |
| W3 | 0 04 00 00 30052 |
| W4 | 0 ——— 0 |

S 04 $9M52

temporary (arithmetic)

| W1 | Sequence |
|---|---|
| W2 | 0 48 04 00 00001 |
| W3 | 0 04 01 00 40551 |
| W4 | 0 ——— 0 |

FF #04 $9M551;2

fast register.

| W1 | Sequence |
|---|---|
| W2 | 0 32 11 07 00001 |
| W3 | 0 05 01 00 00000 |
| W4 | 0 ——— 0 |

DR #11 99907

fast register

| W1 | Sequence |
|---|---|
| W2 | 0 01 09 07 00001 |
| W3 | 0 05 01 01 00000 |
| W4 | 0 ——— 0 |

AX #9 #7

INSTRUCTION ITEMS (continued.)

Address mode — F90 or F94 item — F10 item

marker & symbol

| | Sequence |
|---|---|
| W1 | |
| W2 | 0 92 00 00 00001 |
| W3 | 0 06 01 00 00148 |
| W4 | 0 ——— 0 |

TB #00 CØS

marker & symbol

| | Sequence |
|---|---|
| W1 | |
| W2 | 0 91 00 05 00001 |
| W3 | 0 07 00 01 00149 |
| W4 | 0 ——— 000150 |

TR EBB & FLO #05

floating literal

| | Sequence |
|---|---|
| W1 | |
| W2 | 0 02 06 00 00001 |
| W3 | 0 17 01 00 00000 |
| W4 | 05 13 14 159 000 |

A #06 (0.314159(01)

double precision literal

| | Sequence |
|---|---|
| W1 | |
| W2 | 0 16 00 00 00001 |
| W3 | 0 18 01 00 00151 |
| W4 | -42 12345 0000 |

NN #00 (-0..12345(-08)

fixed literal

| | Sequence |
|---|---|
| W1 | |
| W2 | 0 01 14 00 00001 |
| W3 | 0 19 01 00 00000 |
| W4 | 0 ——— 081 |

AX #14 (81)

B Box literal

| | Sequence |
|---|---|
| W1 | |
| W2 | 0 43 09 00 00001 |
| W3 | 0 20 00 00 00000 |
| W4 | 003 0031 00882 |

F 09 (BB 003 0031 00882)

INSTRUCTION ITEMS (continued)

Address mode — F90 or F94 item — F10 item.

instruction literal

| | |
|---|---|
| W1 | Sequence |
| W2 | 0 43 20 00 00001 |
| W3 | 0 21 00 00 00000 |
| W4 | 0 ——— 0 |

F 20 K9IT123

(The next item will be given the name of this literal.)

next item

| | |
|---|---|
| W1 | Sequence |
| W2 | 5 00 09 08 00001 |
| W3 | 0 03 00 00 00146 |
| W4 | 0 ——— 0 |

K9IT123 SK 09 TRUC
08 5

temporary (index)

| | |
|---|---|
| W1 | Sequence |
| W2 | 0 43 19 00 00001 |
| W3 | 0 04 01 00 20079 |
| W4 | 0 ——— 0 |

F #19 $9x79

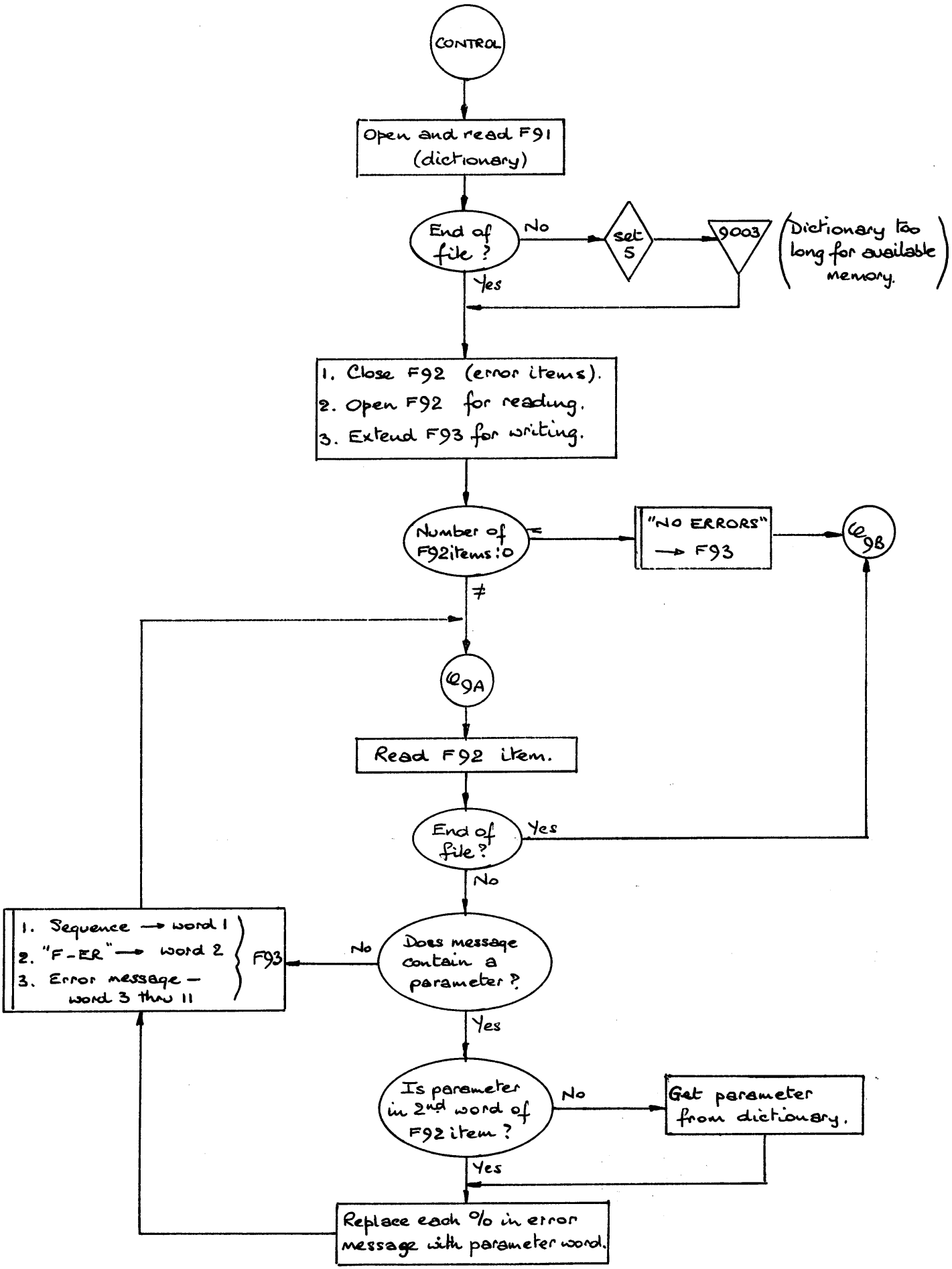
STATEMENT NAME ITEMS

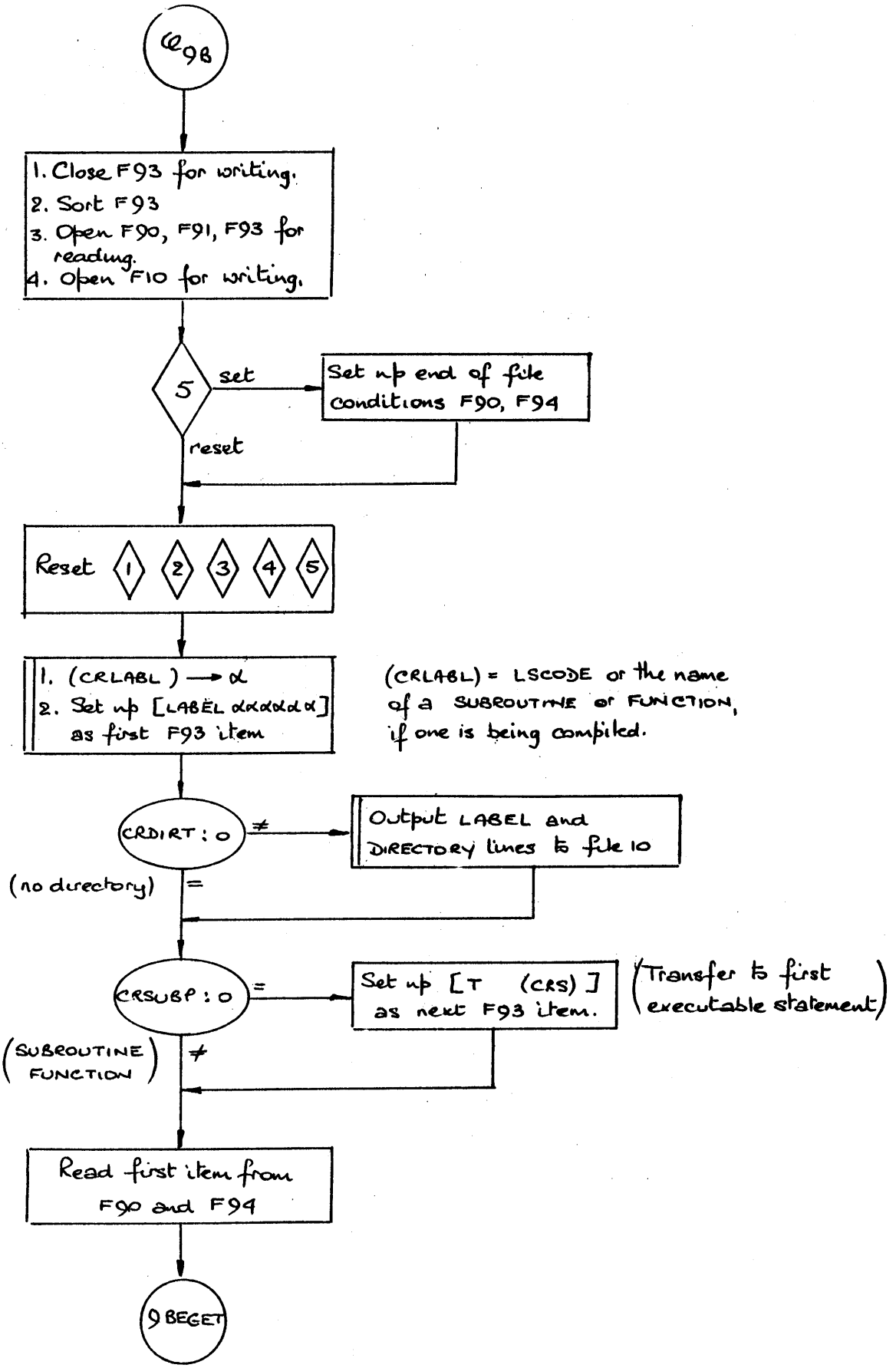
| W1 | Sequence |
|---|---|
| W2 | 0 ——— 0 |
| W3 | 0 ——— 00123 |
| W4 | 0 ——— 0 |

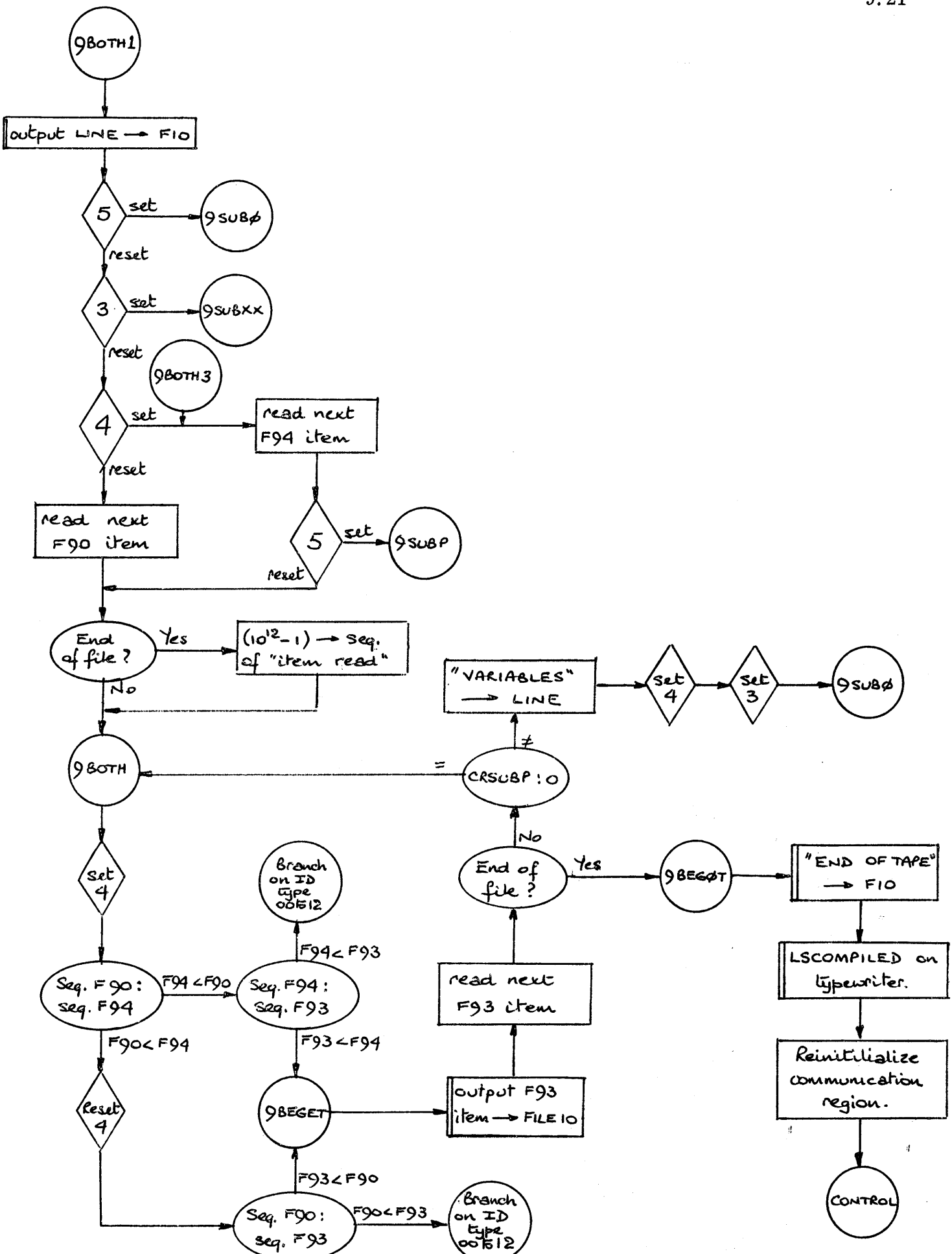
Name will be the contents of the 123rd entry in the dictionary File 91.

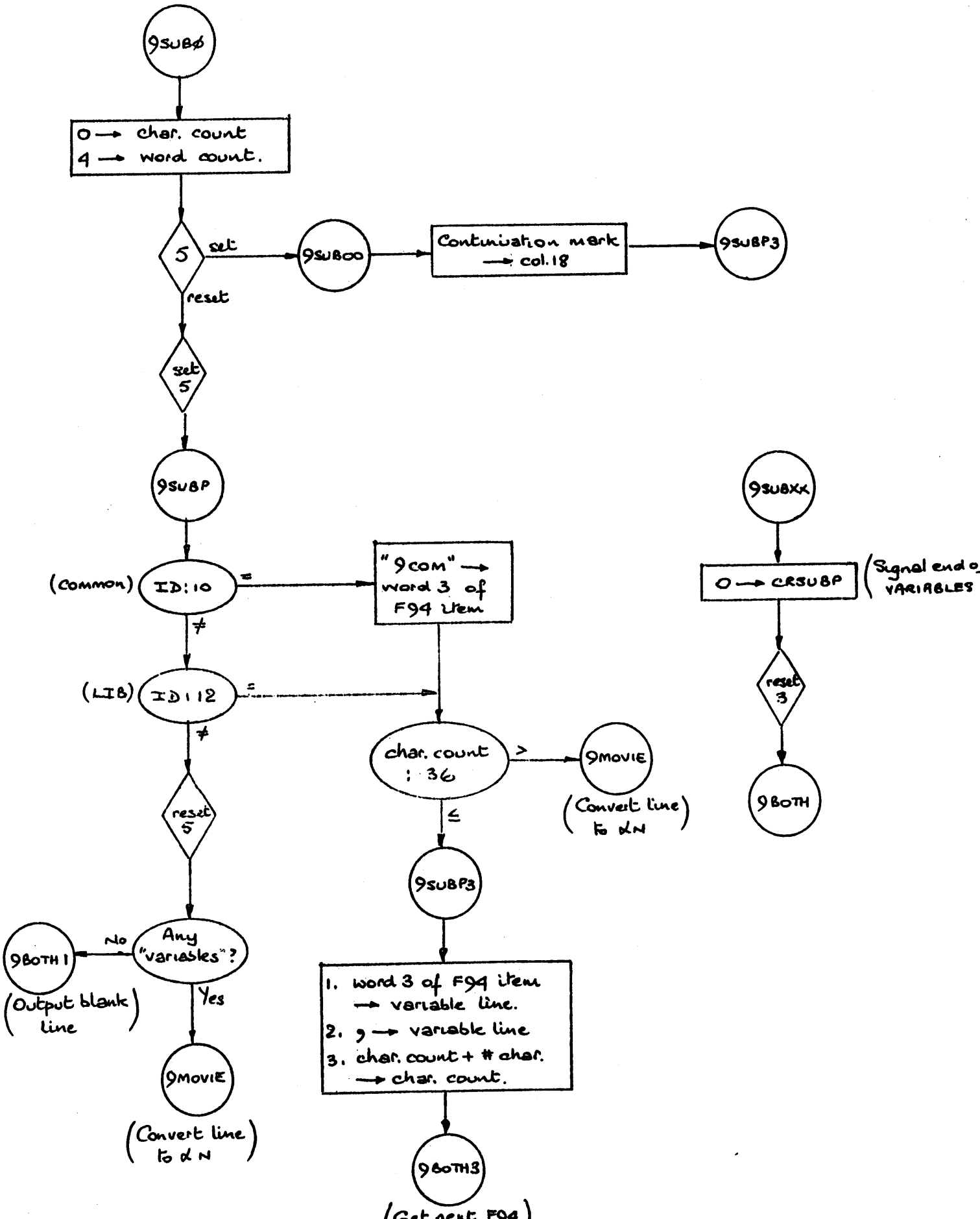
| W1 | Sequence |
|---|---|
| W2 | 0 ——— 0 |
| W3 | 0 ——— 023579 |
| W4 | 0 ——— 0 |

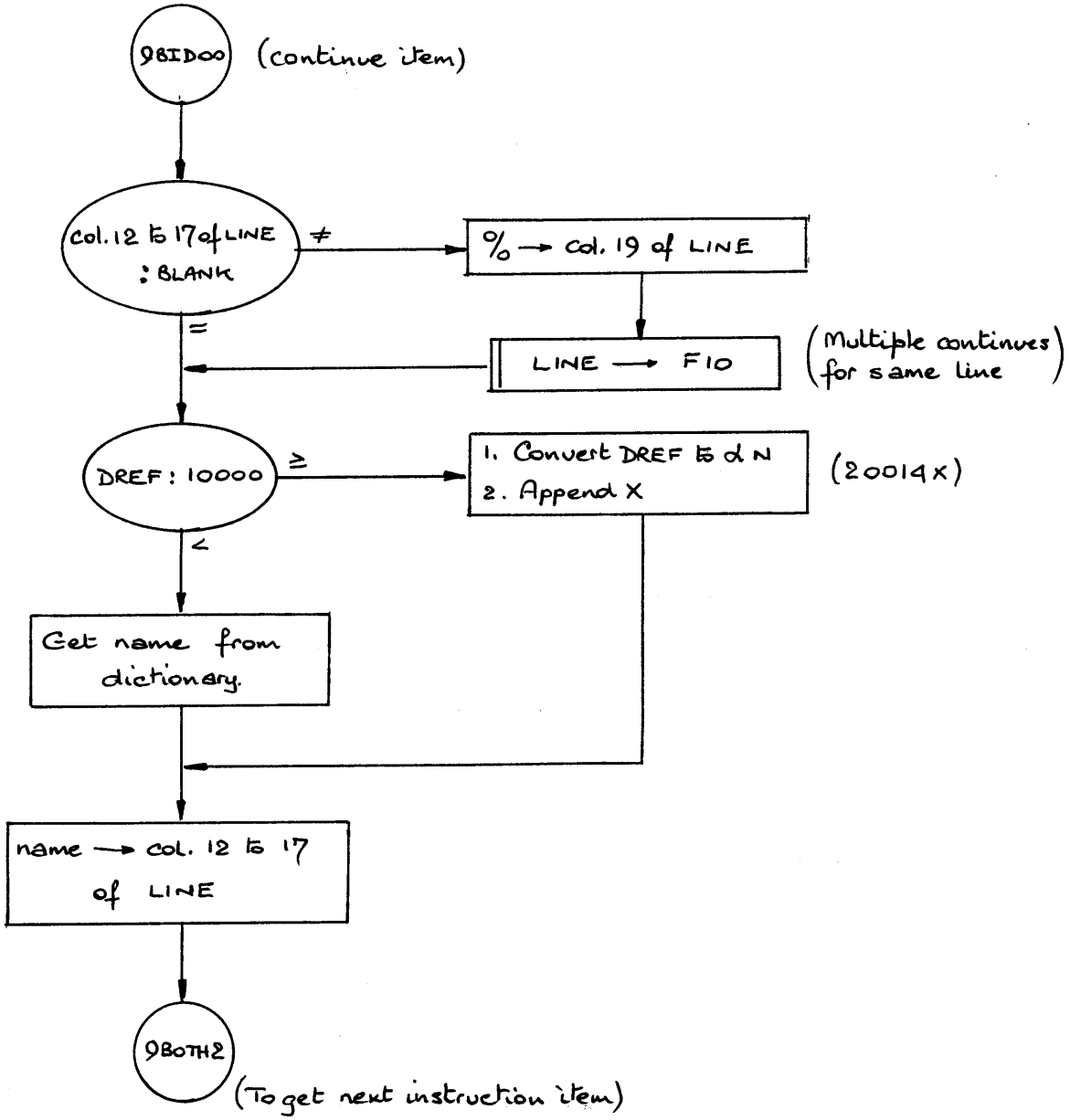
Name will be : 23579 X

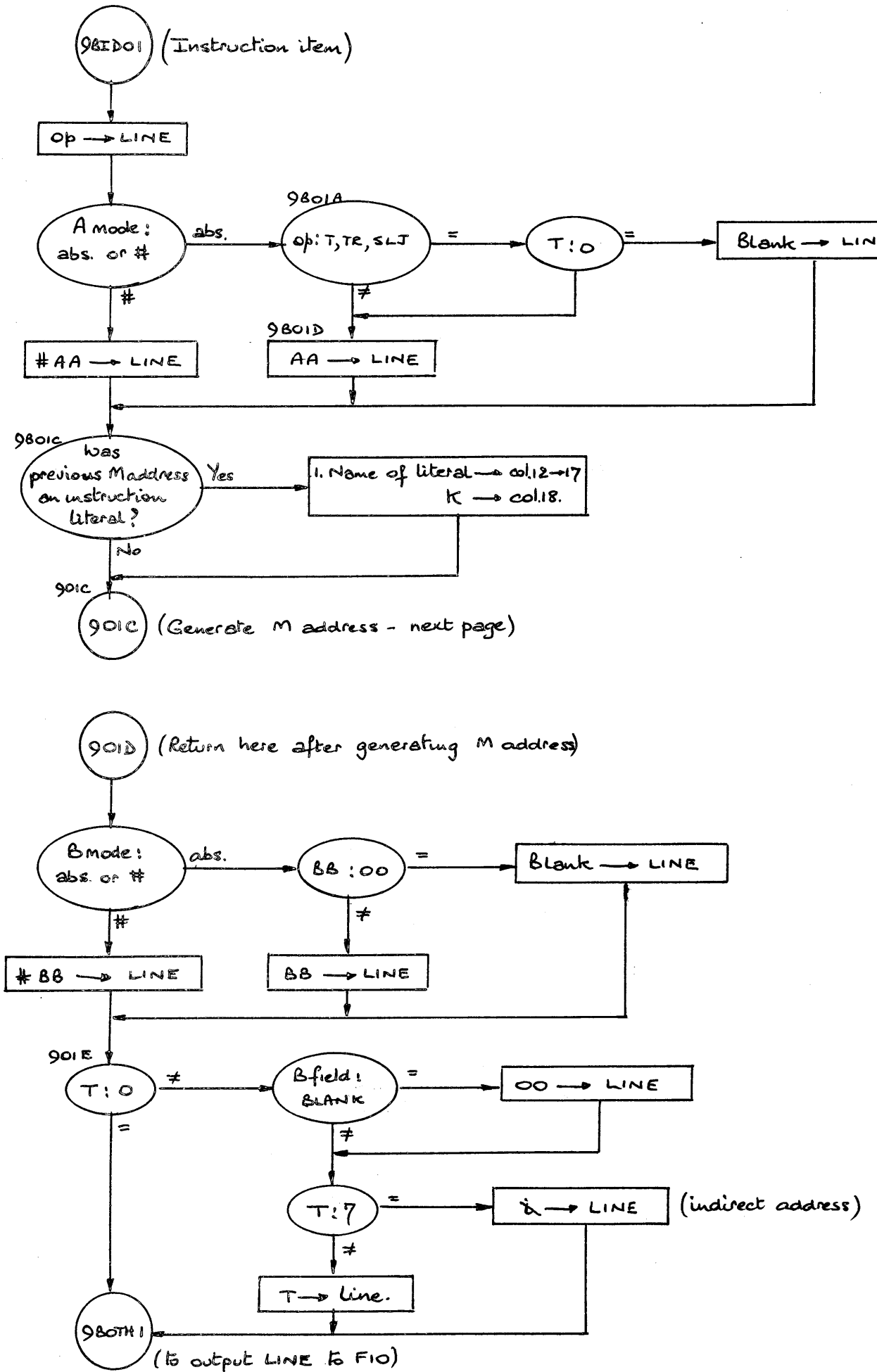

CONTROL
Open and read F91 (dictionary)
End of file ?
No
set 5
9003
Dictionary too long for available memory.
Yes
1. Close F92 (error items).
2. Open F92 for reading.
3. Extend F93 for writing.
Number of F92items:0
=
"NO ERRORS" → F93
98
≠
9A
Read F92 item.
End of file?
Yes
No
1. Sequence → word 1
2. "F-ER" → word 2
3. Error message — word 3 thru 11
F93
No
Does message contain a parameter?
Yes
Is parameter in 2nd word of F92 item ?
No
Get parameter from dictionary.
Yes
Replace each % in error message with parameter word.

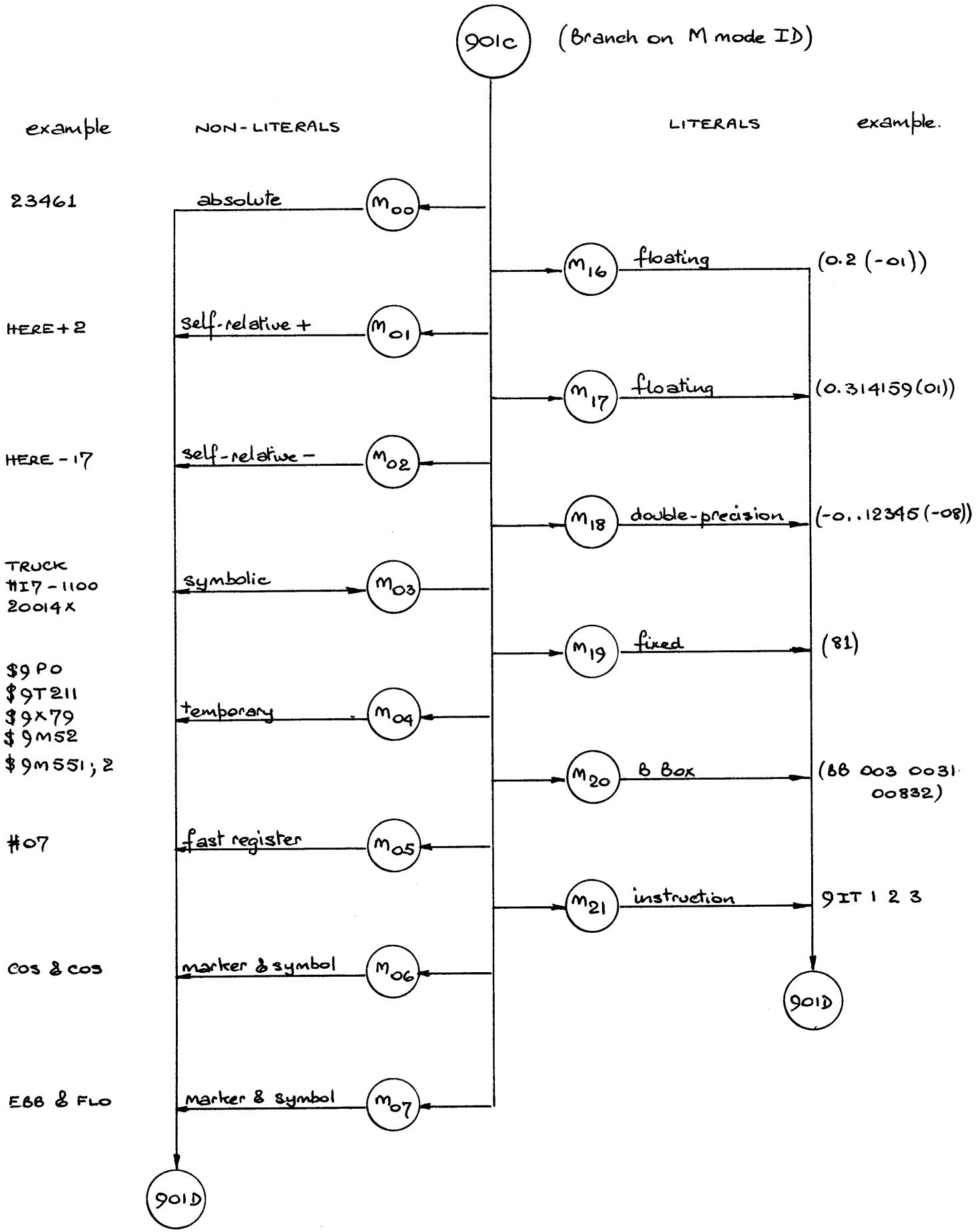

@98
1. Close F93 for writing.
2. Sort F93
3. Open F90, F91, F93 for reading.
4. Open F10 for writing.
5
set
Set up end of file conditions F90, F94
reset
Reset 1 2 3 4 5
1. (CRLABL) → α
2. Set up [LABEL αααααα] as first F93 item
(CRLABL) = LSCODE or the name of a SUBROUTINE or FUNCTION, if one is being compiled.
CRDIRT : 0
≠
Output LABEL and DIRECTORY lines to file 10
(no directory)
=
CRSUBP : 0
=
Set up [T (CRS)] as next F93 item.
(Transfer to first executable statement)
(SUBROUTINE FUNCTION)
≠
Read first item from F90 and F94
9BEGET

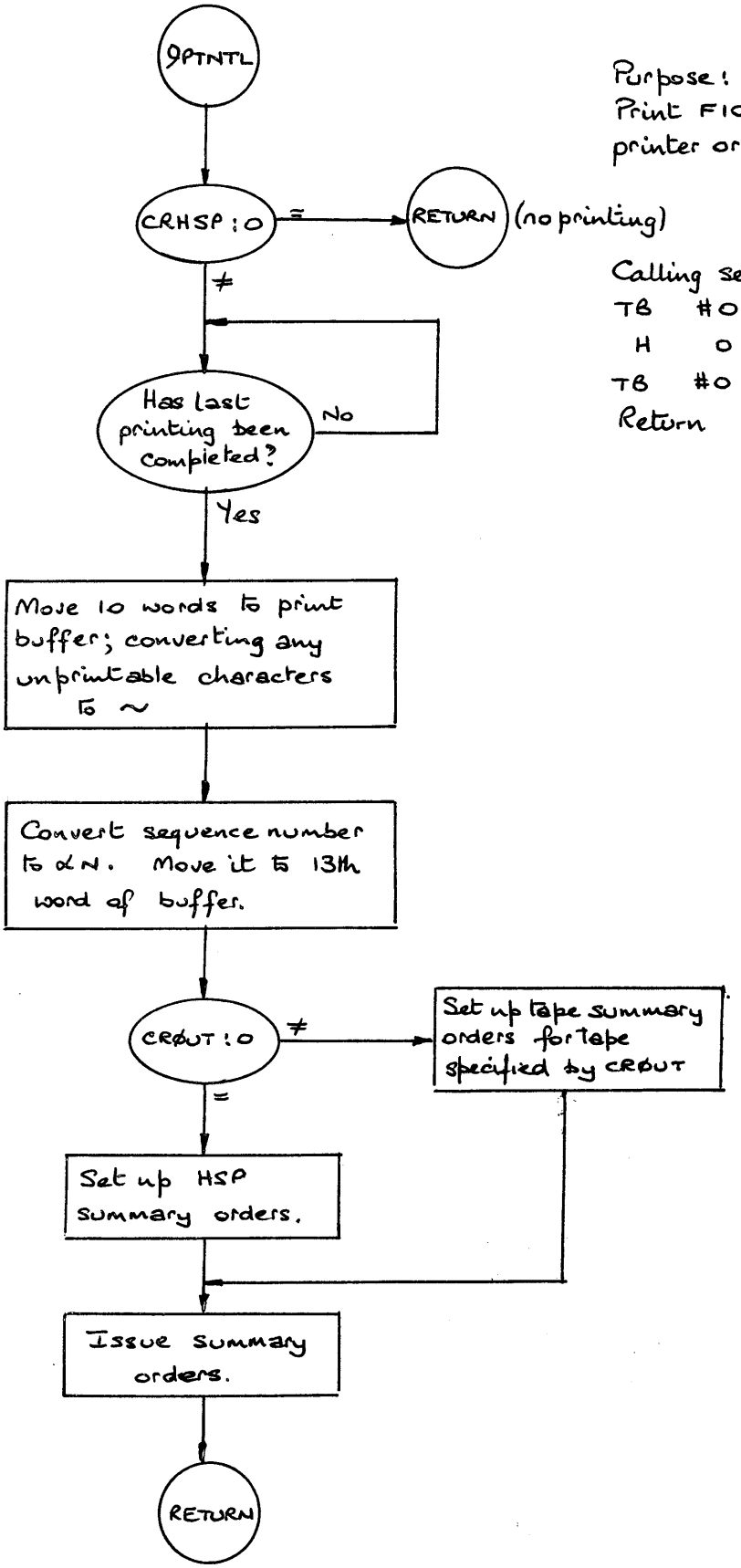

9BOTH1
output LINE → F10
5
set
9SUBØ
reset
3
set
9SUBXX
reset
9BOTH3
4
set
read next F94 item
reset
read next F90 item
5
set
9SUBP
reset
End of file ?
Yes
(10^12 - 1) → seq. of "item read"
No
"VARIABLES" → LINE
set 4
set 3
9SUBØ
9BOTH
=
CRSUBP : 0
≠
set 4
Branch on ID type 00 to 12
End of file ?
Yes
9BEGØT
"END OF TAPE" → F10
LSCOMPILED on typewriter.
Seq. F90 : seq. F94
F94 < F90
Seq. F94 : Seq. F93
F94 < F93
read next F93 item
F90 < F94
F93 < F94
Reset 4
9BEGET
output F93 item → FILE 10
Reinitialize communication region.
F93 < F90
Seq. F90 : seq. F93
F90 < F93
Branch on ID type 00 to 12
CONTROL


9SUBØ
0 → char. count
4 → word count.
5
set
reset
9SUBOO
Continuation mark → col. 18
9SUBP3
set 5
9SUBP
(COMMON) ID:10
=
≠
"9COM" → word 3 of F94 item
(LIB) ID:12
=
≠
char. count : 36
>
9MOVIE
(Convert line to ØN)
≤
reset 5
9SUBP3
No
Any "variables"?
9BOTH1
(Output blank line)
Yes
9MOVIE
(Convert line to ØN)
1. word 3 of F94 item → variable line.
2. 9 → variable line
3. char. count + # char. → char. count.
9BOTH3
(Get next F94)
9SUBXX
0 → CRSUBP
(Signal end o VARIABLES
reset 3
9BOTH


9BID00
(continue item)
col. 12 to 17 of LINE : BLANK
≠
% → col. 19 of LINE
=
LINE → F10
(Multiple continues for same line
DREF : 10000
≥
1. Convert DREF to d N
2. Append X
(20014X)
<
Get name from dictionary.
name → col. 12 to 17 of LINE
9BOTH2
(To get next instruction item)


9BID01
(Instruction item)
Op → LINE
A mode: abs. or #
abs.
9B01A
op: T, TR, SLJ
=
T: 0
=
Blank → LINE
≠
#
9B01D
#AA → LINE
AA → LINE
9B01C
Was previous M address an instruction literal?
Yes
1. Name of literal → col.12→17
K → col.18.
No
901C
901C
(Generate M address - next page)
901D
(Return here after generating M address)
B mode: abs. or #
abs.
BB: 00
=
Blank → LINE
≠
#
# BB → LINE
BB → LINE
901E
T: 0
≠
B-field: BLANK
=
00 → LINE
=
≠
T: 7
=
i → LINE
(indirect address)
≠
T → Line.
9BOTH1
(To output LINE to FIO)


901C
(Branch on M mode ID)
example
NON-LITERALS
LITERALS
example.
23461
absolute
M00
M16
floating
(0.2 (-01))
HERE+2
self-relative +
M01
M17
floating
(0.314159(01))
HERE-17
self-relative -
M02
M18
double-precision
(-0..12345(-08))
TRUCK
#I7-1100
20014X
symbolic
M03
M19
fixed
(81)
$9P0
$9T211
$9X79
$9M52
$9M551;2
temporary
M04
M20
B Box
(BB 003 0031 00832)
#07
fast register
M05
M21
instruction
9IT 1 2 3
COS & COS
marker & symbol
M06
901D
EBB & FLO
marker & symbol
M07
901D




Purpose:
Print FIO on high speed printer or specified uniservo

Calling sequence:

```
TB    #0    FILE+FIO
 H     0    LOC
TB    #0    9PTNTL
Return
```