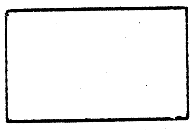
The flowcharts and remarks included in this report are intended to serve as a guide to Edinburgh University, Atlas Autocode, version I. They were prepared from a version written almost completely in Atlas Autocode and therefore do not necessarily reflect the intricacies of the usercode version currently in use on KDF-9. It is suggested that anyone wishing to change the compiler or permanent material use these notes as a guide to the code, rather than drawing conclusions directly from the notes.

TABLE OF CONTENTS

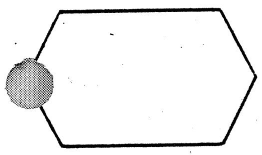
Sections

The flowcharts used in this report are usually flowcharts of routines. The following symbols are used

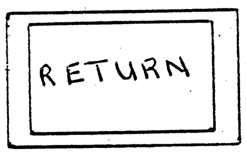
for entry points



for unconditional instructions



for two path branches



to indicate return from a routine

Small numbers above a box indicate a label (in the code) associated with the instruction in that box.

Two kinds of flowcharts are used, one kind in English, the other in Atlas Autocode. Where both cover the same section of the compiler, they are presented side by side, as much as possible.

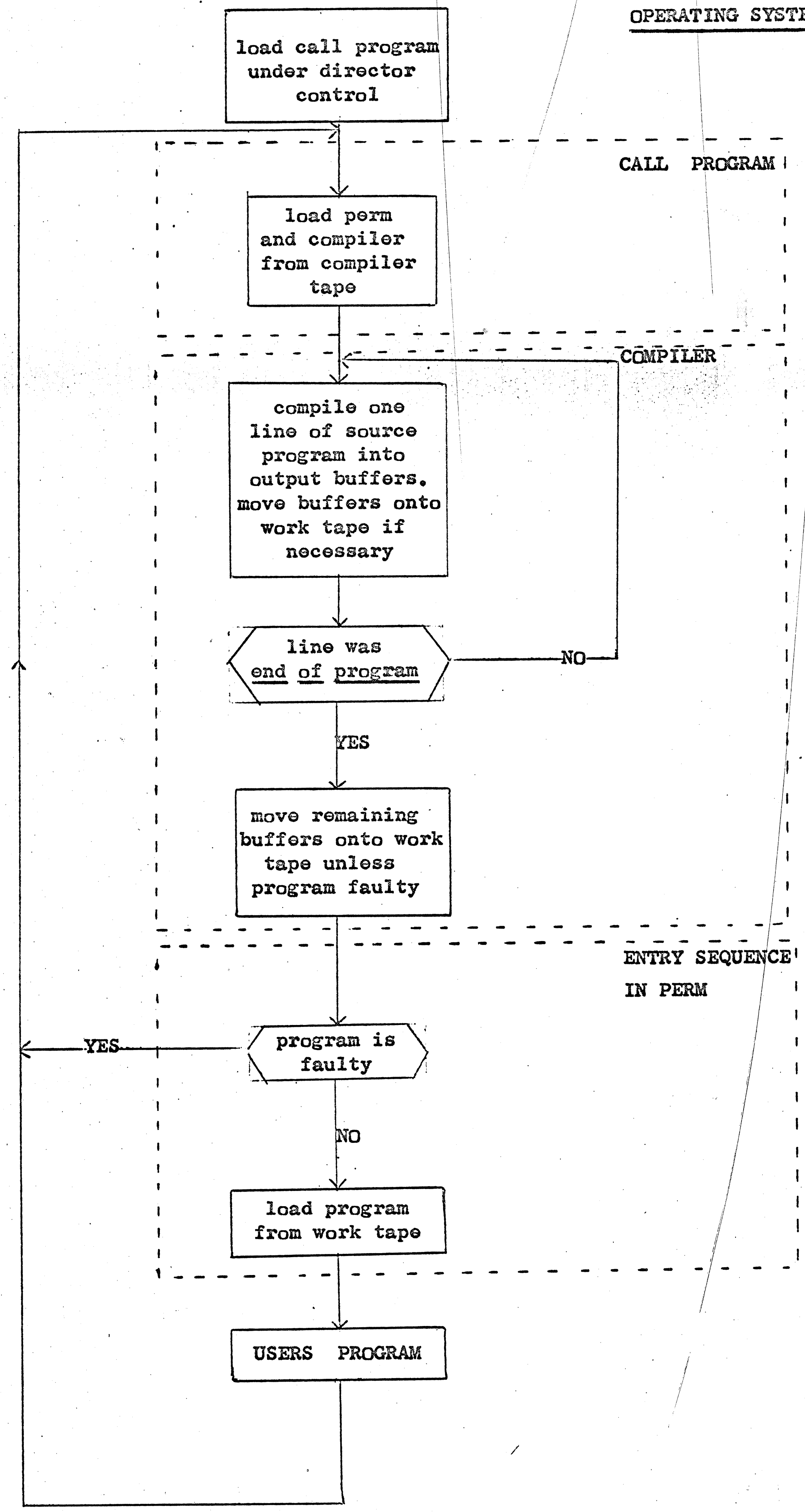
In the notes to the flowcharts, names refering to routines arrays, etc. declared in compiler are enclosed in pointed brackets < >.

P R O G R A M

M A P O F

C O M P I L E R

OUTLINE OF AA OPERATING SYSTEM



The Atlas Autocode operating system may be thought of as three pieces of code. The first is a call program which will load the other two sections from the (magnetic) compiler tape. The second is a group of routines and sub programs which are permanent material for all Atlas Autocode programs. This permanent material (perm) includes the I/O and math function routines, user program entry and exit sequences, array handling, routine calls,error checking, etc. The third section is the compiler itself, a binary program for compiling AA programs and able to directly reference locations in perm. (In many respects the compiler may be thought of as an ordinary compiled AA program, as it is capable of compiling itself and was originally written in Atlas Autocode.)

The core layout of the system may be pictured as follows:

Location

0	Eo	Eo+150	58P		
Director	Compiler call Program	PERM	COMPILER	Compiler Stack and Routine Addresses	Free store For Lists and arrays

The compiler compiles a program statement by statement, and at intervals dumps the resulting code onto magnetic tape *(the work tape). When the program is completely compiled, it is loaded into core *, over laying the compiler, by a program entry sequence in perm. When loading is finished, a user program will occupy core as follows:

0	EO	Eo+150	58P	<u>end of program</u>	
Director	Compiler Call Program	Perm	User Program	Program Stack and Routine Addresses	Free Store For Arrays

Control is then transfered to the user program, which executes and upon termination jumps to a <u>stop</u> sequence in perm, which ultimately jumps to the call block where perm and compiler are reloaded.

*Unless, of course, it is faulty.

PROGRAM MAP OF COMPILER

Opposite the block diagram of the compiler are listed the sections which cover the material.

```
begin
    begin
phrase struction      routine read string      } not covered
read-in               routine record
                      routine lookup
    end
    begin
                      routine initialize                  RUN TIME COMPILER INITILIZATION
                      routine splash
                      integer fn ca                           CODE DUMPING
                      routine dump stack and routines
                      routine compare                         RECOGNITION PHASE

                      routine cSS                             COMPILATION PHASE
                          begin
                              routine readsym
job                           routine read key word           not covered
headings                      routine initial output
                          end
                          routine cRSPEC                cRSPEC
                          routine cUI                   cUI
expression                routine cSEXP                 cSEXP
compiler                      routine print orders
conditional               routine cCOND
Compiler                  routine cCC
                          routine cSC                   cCOND
                          routine cCOMP
```

name handling	**routine** cNAME	cNAME
	routine cMOD	not covered
print compile time	**routine** fault	
fault diagnostic		
	routine print name	
test whether name set twice	**routine** test nst	BACKGROUND ROUTINES
usercode compiler	**routine** cUCI	not covered
	routine find label	
	routine copy tag	
	routine replace tag	
	routine From list 2	
	routine pop up 2	
	routine store tag	
	routine push down 2	
	routine link	BACKGROUND ROUTINES
	routine more space	
	routine new cell	
	routine return cell	
	routine insert after 2	
	routine skip exp	
	routine store jump	
	routine store name	
	routine p SET	
	routine pSH	
	routine pN	
	routine pQ	CODE DUMPING
	routine fill label	
	routine fill set	
	routine pJ	
	routine PMS	

end

end

end

end

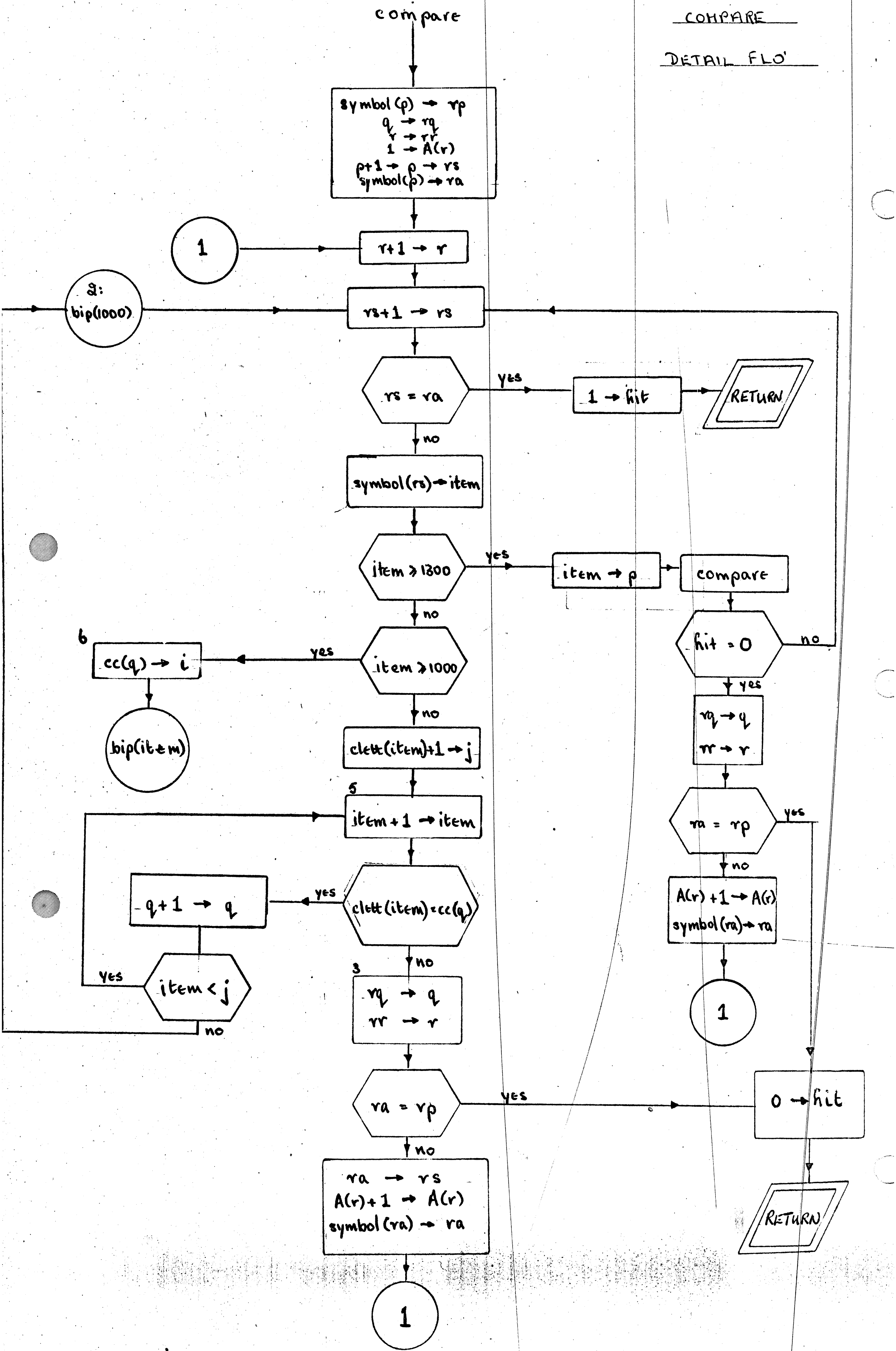
FLOWCHARTS OF COMPARE

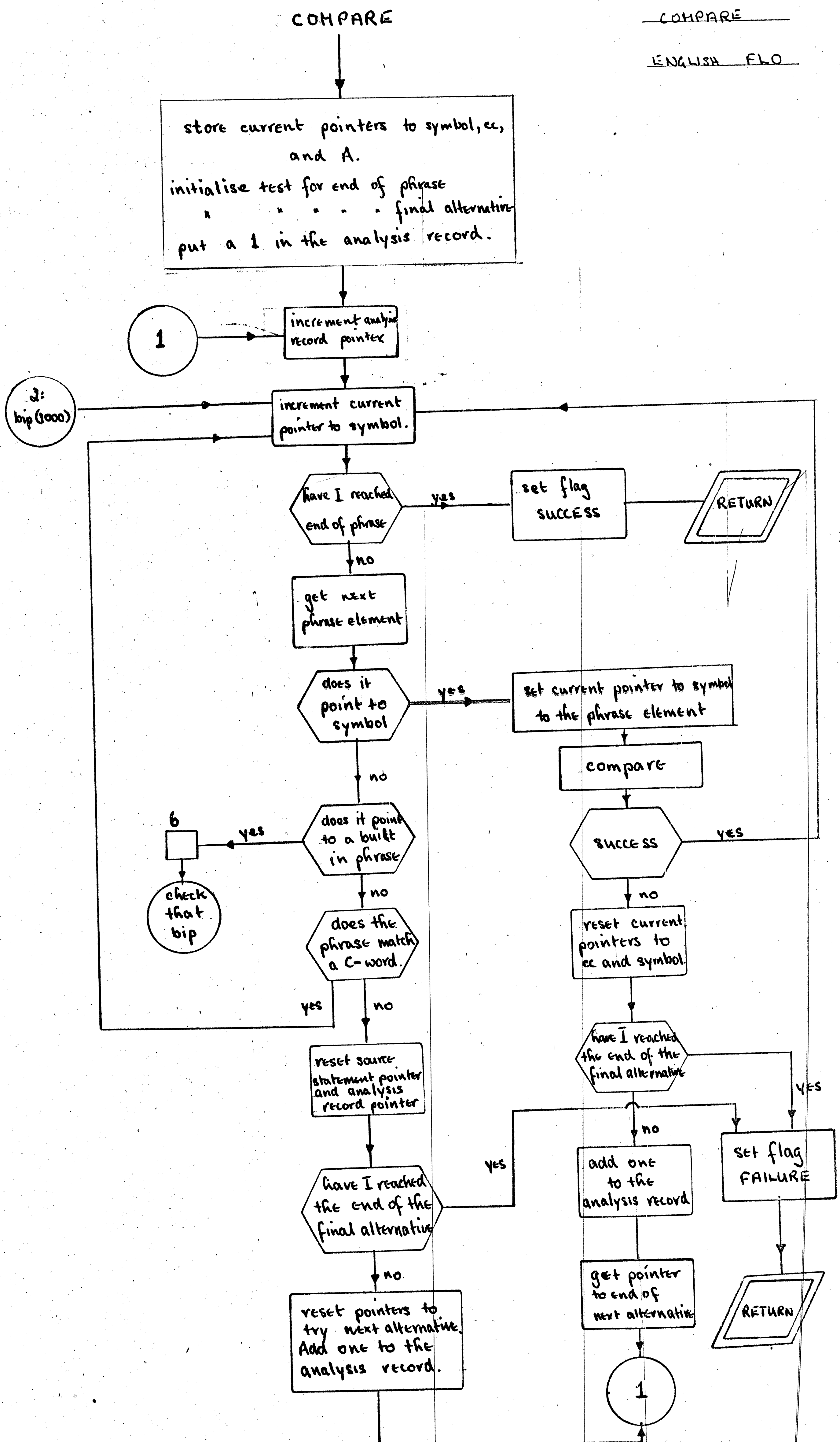
Notes on compare

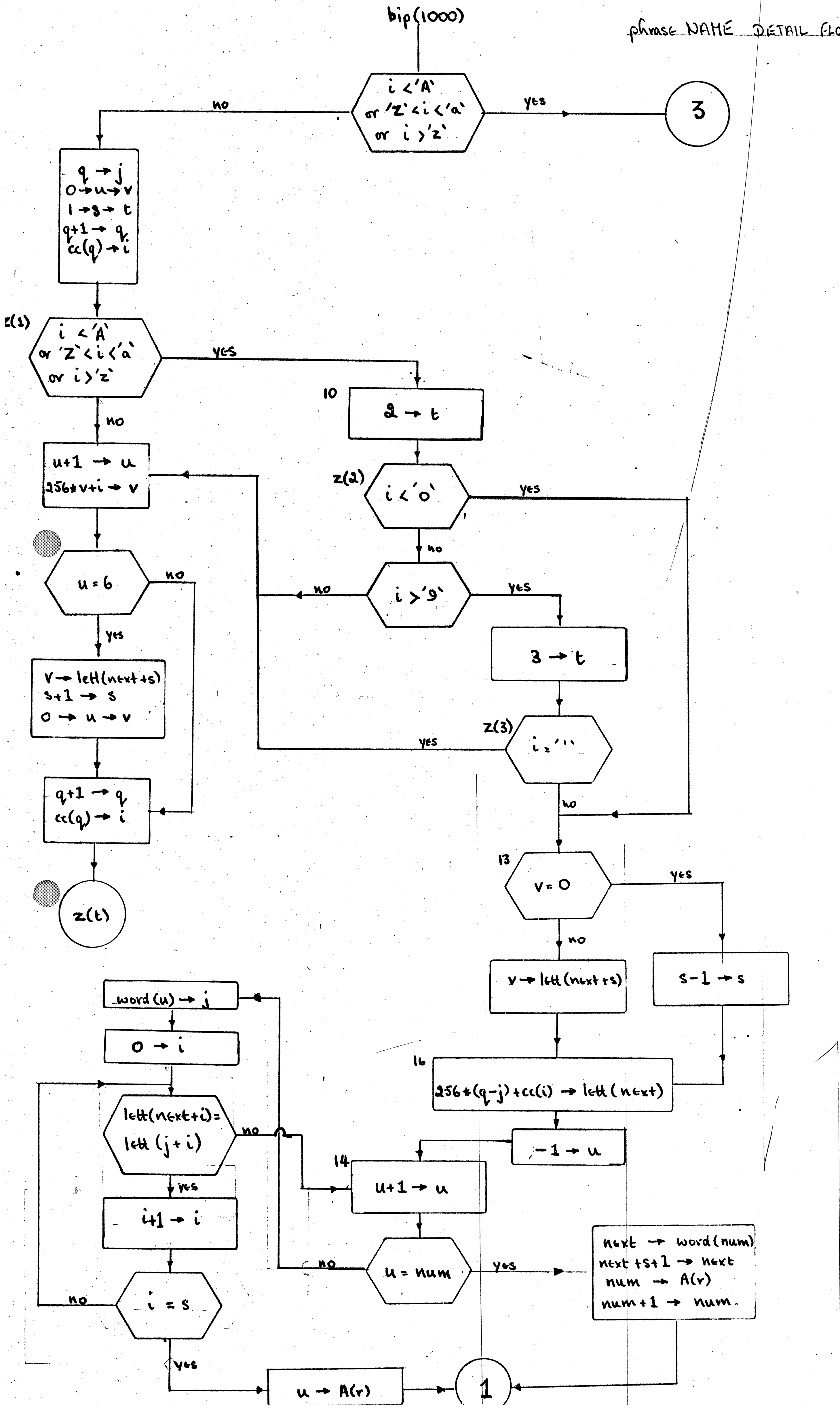
storage	cc	array containing the source statement, one character in each word
	symbol	array containing phrase definitions
	clett	array containing text literals
	A	array. Analysis record.
	p	global pointer to symbol; p points to the entry in symbol for the current <u>definition</u>.
	r	global pointer to analysis record; r is used to plant entries in A, and is reset if compare fails.
	rr	holds reset value of r
	ra	holds pointer to next alternative
	rs	holds pointer to next element of current alternative
	rp	holds pointer to next definition
	q	global pointer to source statement text(cc)
	rq	holds reset value of q

COMPARE

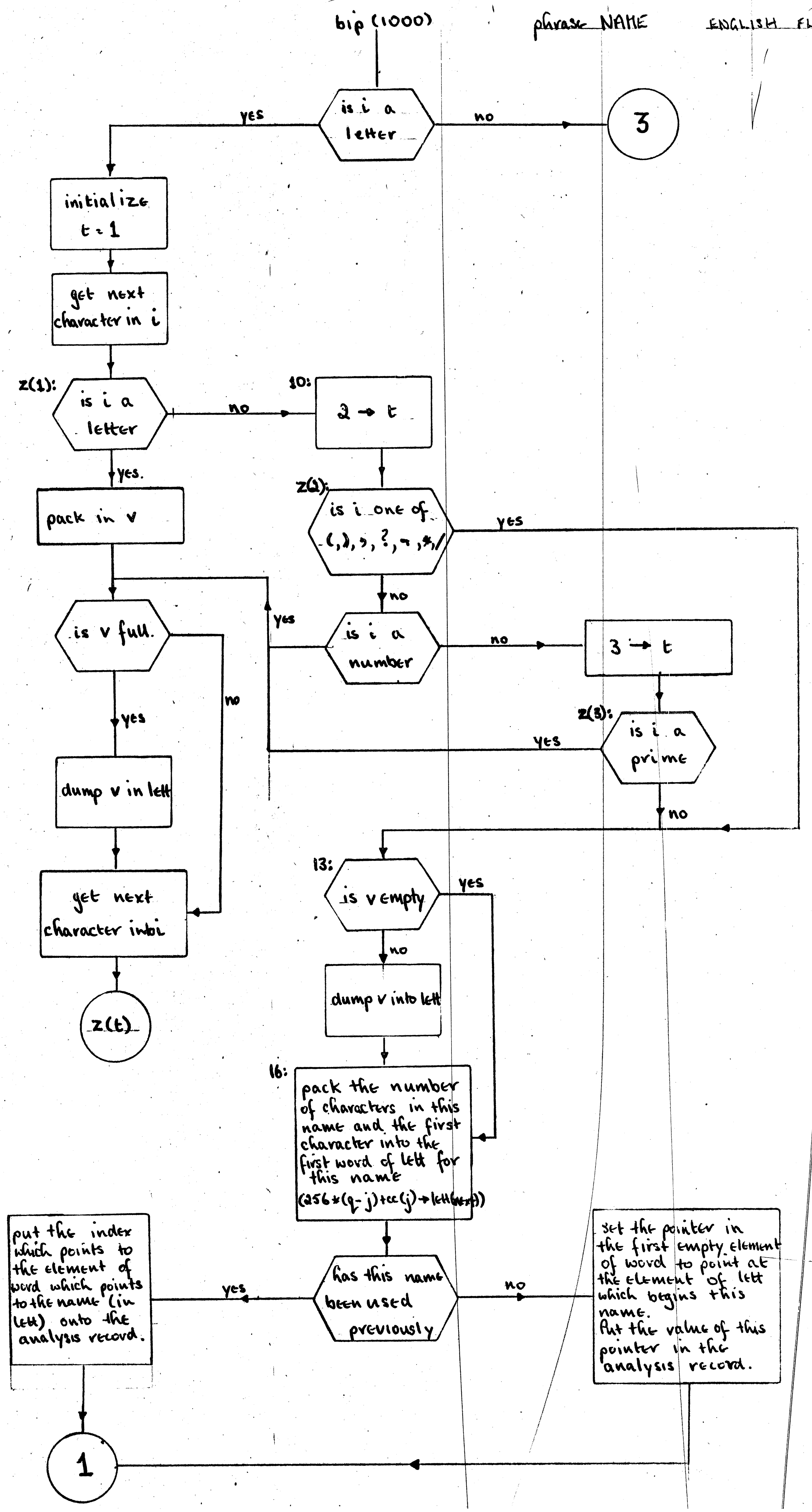
DETAIL FLO'




COMPARE
store current pointers to symbol, cc, and A.
initialise test for end of phrase
" " " " " final alternative
put a 1 in the analysis record.
1
increment analysis record pointer
2: bip(1000)
increment current pointer to symbol.
have I reached end of phrase
yes
set flag SUCCESS
RETURN
no
get next phrase element
does it point to symbol
yes
set current pointer to symbol to the phrase element
compare
no
SUCCESS
yes
does it point to a built in phrase
yes
6
check that bip
no
no
reset current pointers to cc and symbol
does the phrase match a C-word.
yes
no
have I reached the end of the final alternative
yes
reset source statement pointer and analysis record pointer
no
add one to the analysis record
set flag FAILURE
have I reached the end of the final alternative
yes
no
get pointer to end of next alternative
RETURN
reset pointers to try next alternative. Add one to the analysis record.
1

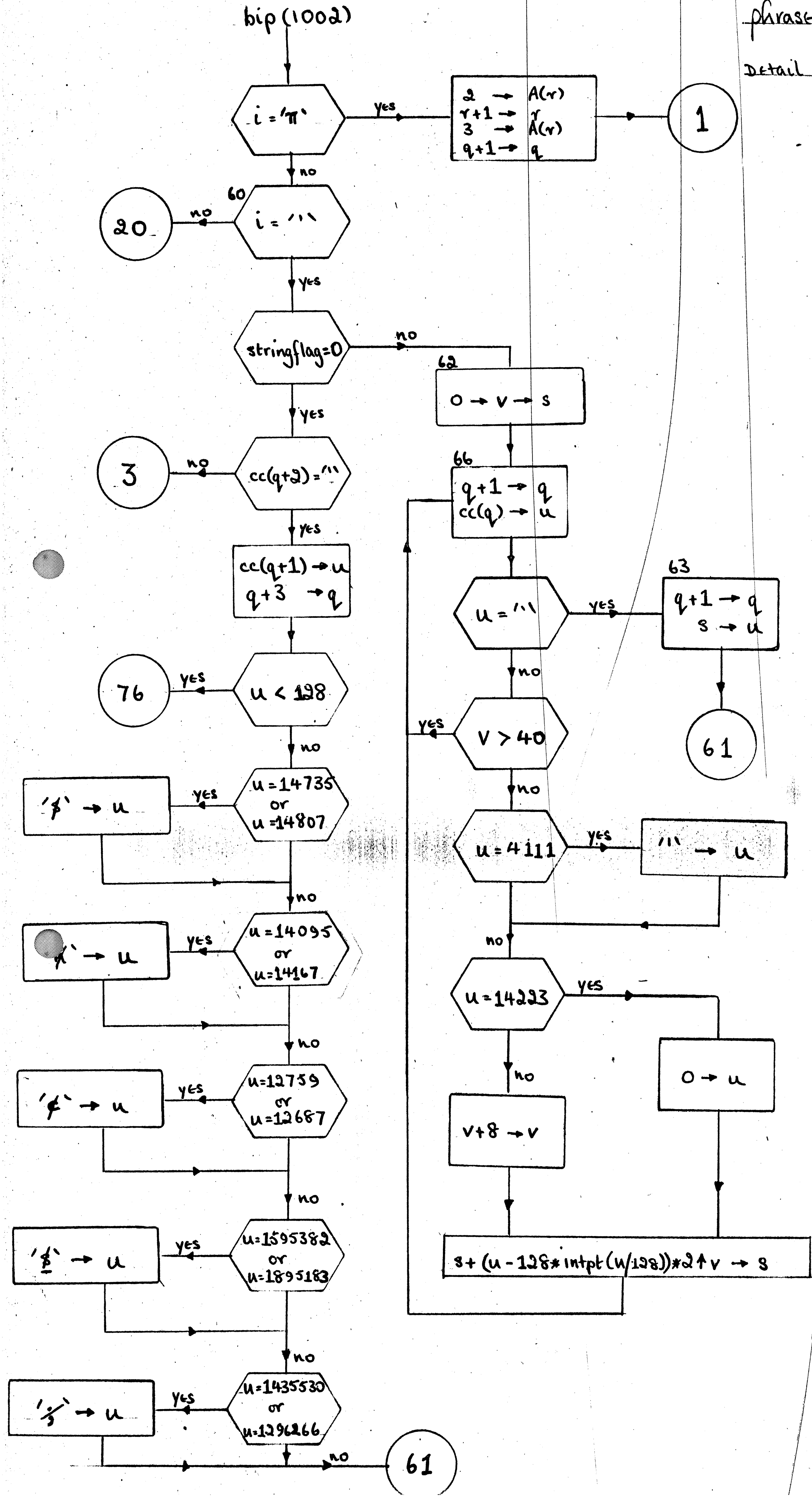

bip(1000)
i < 'A' or 'Z' < i < 'a' or i > 'z'
yes
3
no
q → j
0 → u → v
1 → s → t
q+1 → q
cc(q) → i
z(1)
i < 'A' or 'Z' < i < 'a' or i > 'z'
yes
10
2 → t
z(2)
i < '0'
yes
no
i > '9'
no
yes
3 → t
z(3)
i = ' '
yes
no
u+1 → u
256*v+i → v
u = 6
no
yes
v → lett(next+s)
s+1 → s
0 → u → v
q+1 → q
cc(q) → i
z(t)
13
v = 0
yes
no
v → lett(next+s)
s-1 → s
16
256*(q-j)+cc(i) → lett(next)
-1 → u
14
u+1 → u
u = num
no
yes
next → word(num)
next+s+1 → next
num → A(r)
num+1 → num.
word(u) → j
0 → i
lett(next+i) = lett(j+i)
no
yes
i+1 → i
i = s
no
yes
u → A(r)
1

phrase NAME ENGLISH FLOW.

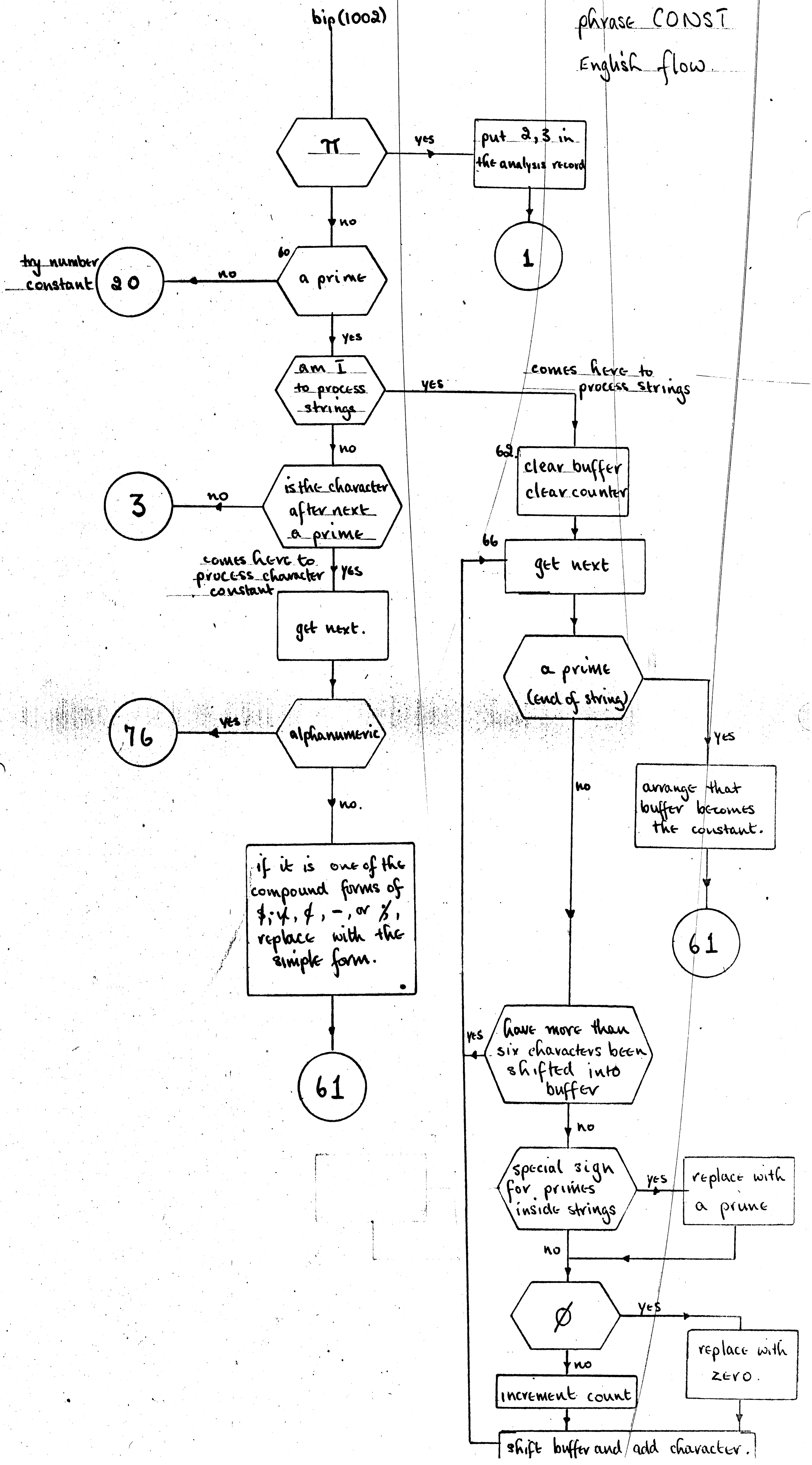


phrase CONST

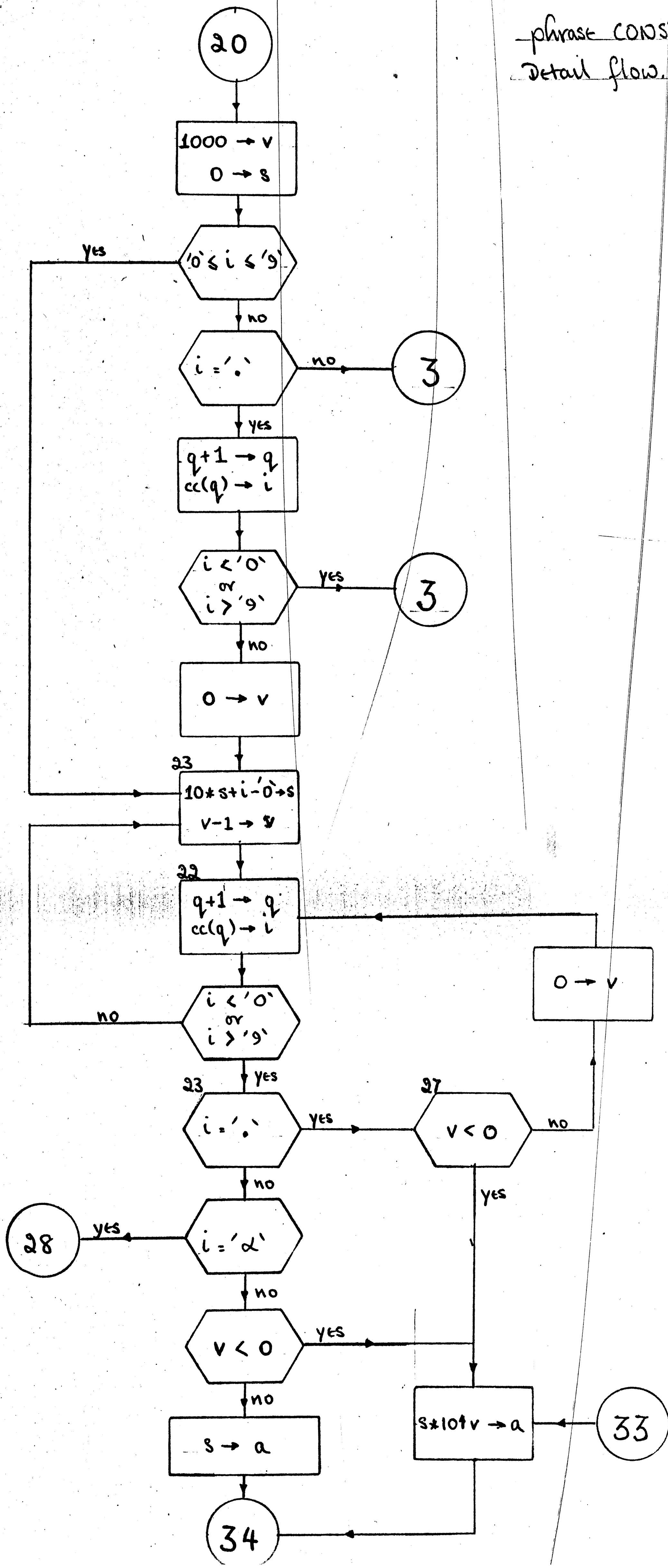
Detail Flow #1



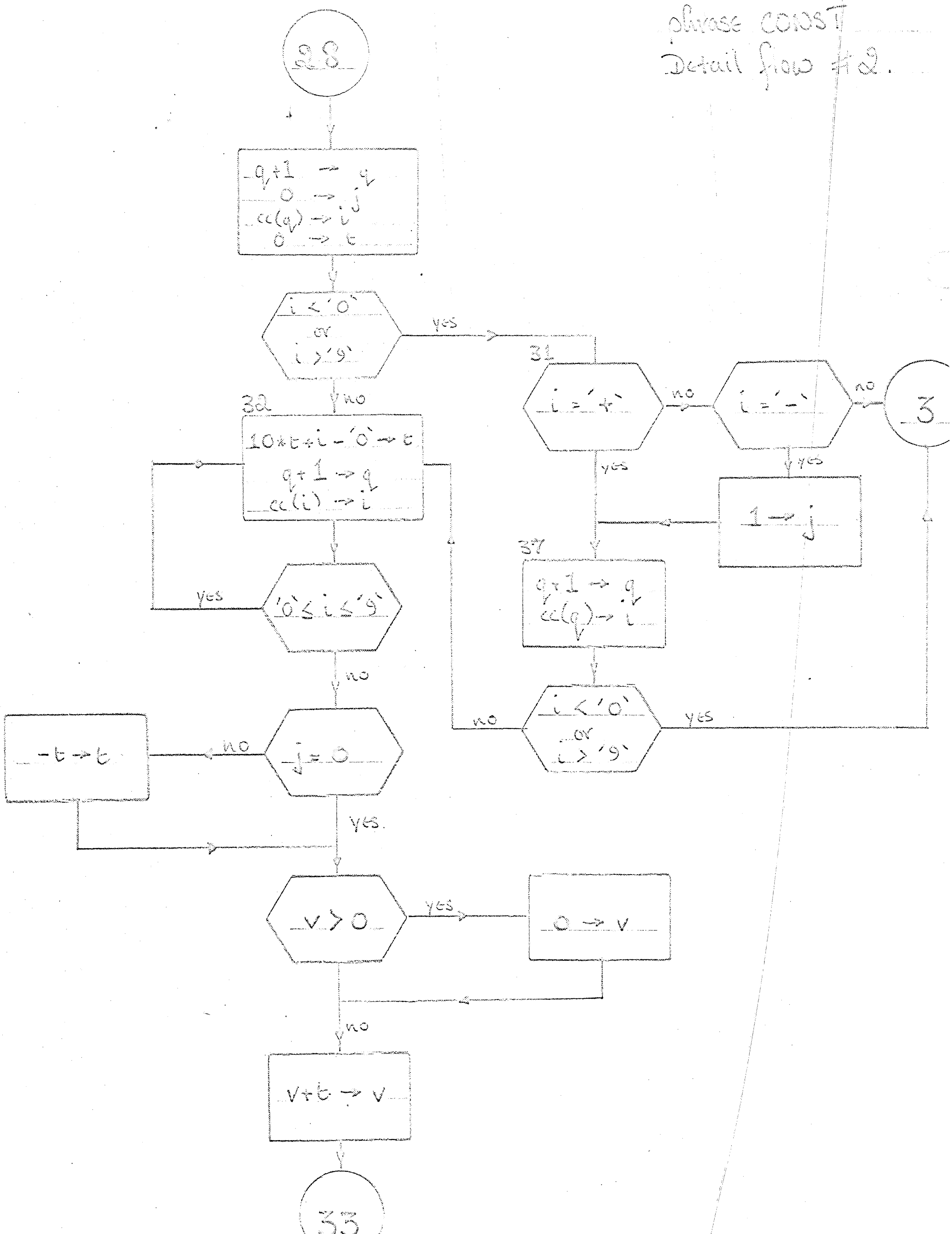
phrase CONST
English flow

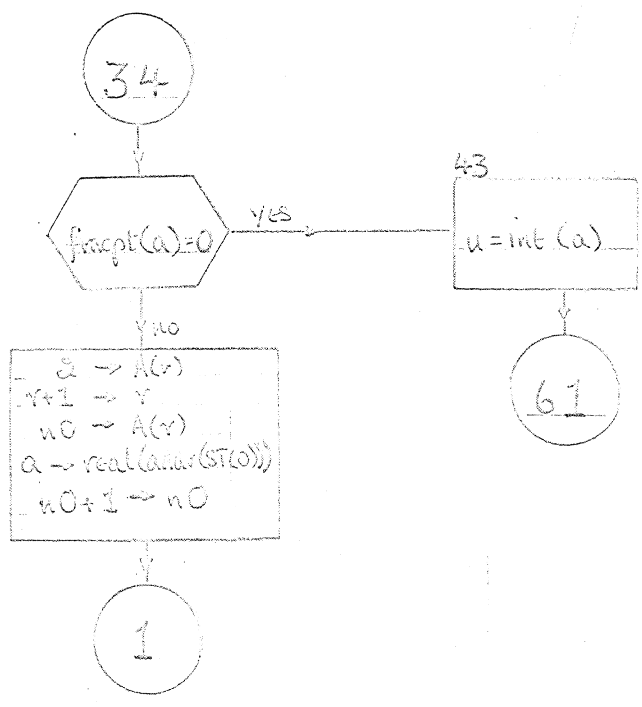


phrase CONST
Detail flow.



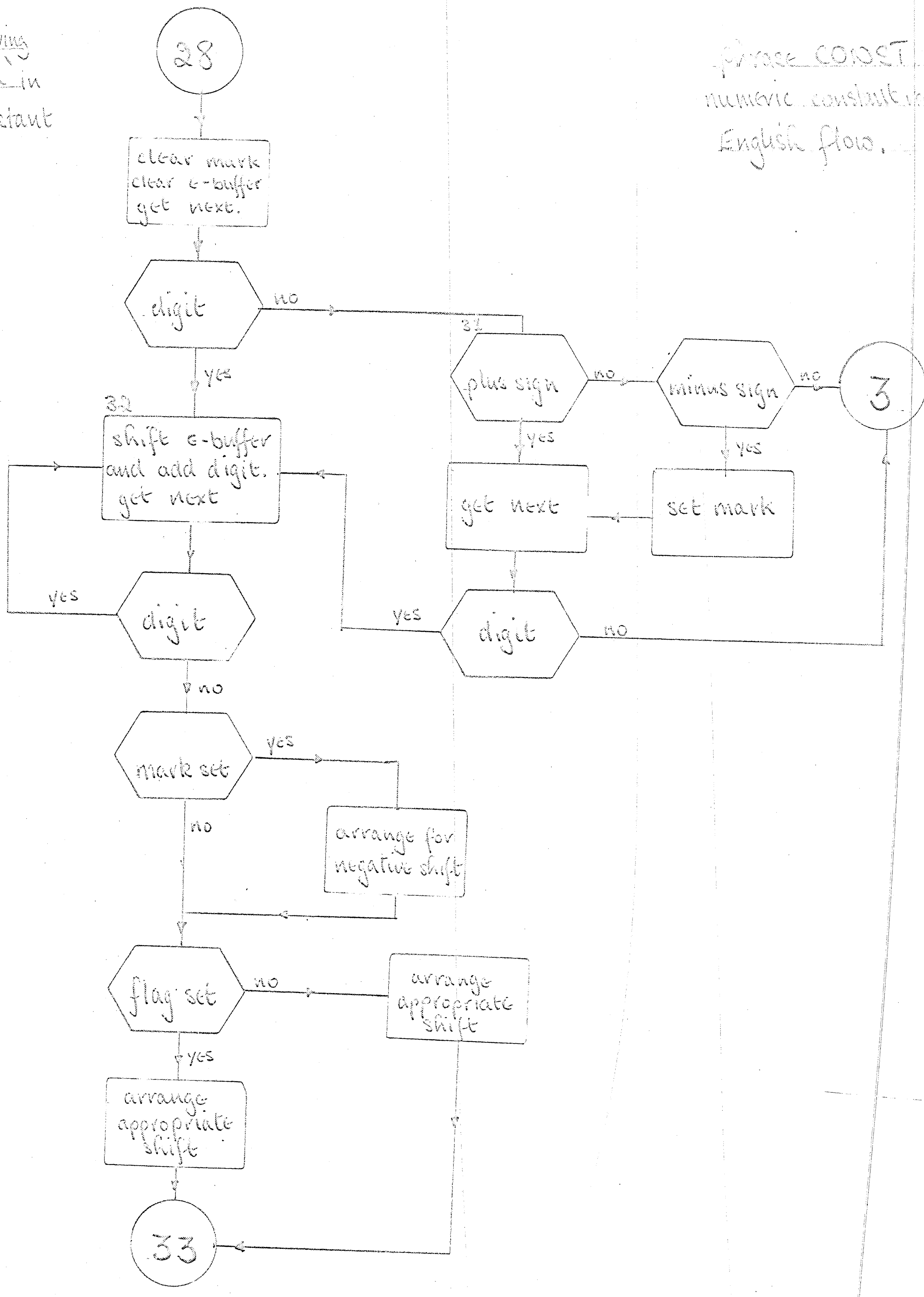
phrase CONST
Detail flow #2.



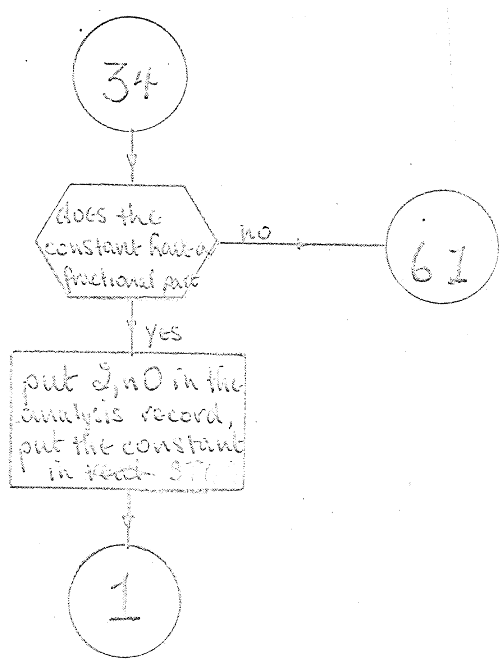


Process CONST ... numeric constant using English flow.

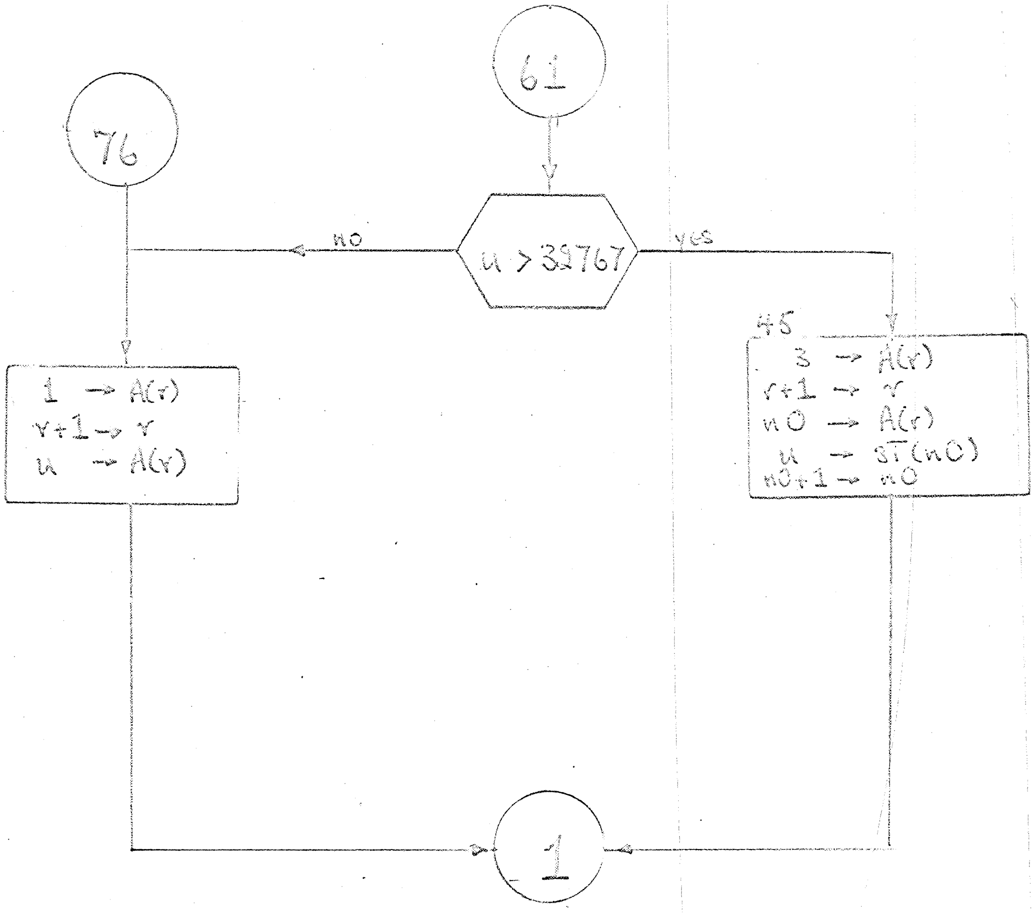
Enter here having just found 'd' in numeric constant



Enter here to decide whether integer or floating point

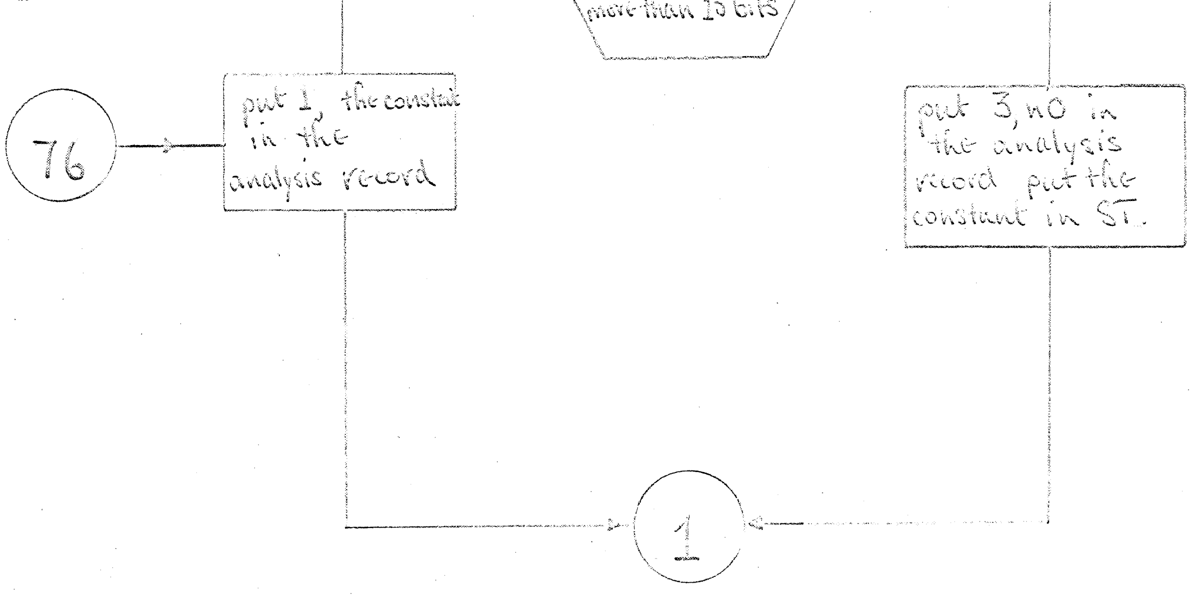


phrase CONST
Detail flow.



Enter here to decide whether to SET the constant or to store it at compile time

English flow.



COMPILATION PHASE cSS

Compilation of an analysis record is accomplished by using certain of its elements as switch indices to jump to particular routines, using others as references to name, constant and label lists, and interrogating others as flags. There is no general way of explaining all the techniques as each analysis record will represent a different tree structure. The following general remarks do apply, however.

The first entry in the analysis record is always interpreted by <cSS> as a switch to code which handles the alternative of [SS] represented. The first command in <cSS> is

->sw(A(1))

and the rest of the analysis record is dealt with by the code to which control is switched. Briefly these sections are

A(1)

1 unconditional instructions,conditionals in alternate form

2 cycles

3 repeats

4 labels

5 conditionals in normal form

6 ! comments

7 integer & real declarations

8 ends

9 routine specs

10 specs

11 comment comments

12 array declarations

13 job headings

14 begins

15 end of program

16 upper case delimiters

17 switch labels

18 switch declarations

19 compile queries

20 ignore queries

21 machine code permit

22 P-labels

23 machine code instructions

24 fault trap declarations

25 normal delimiters

26 string permit

27 end of perm

28 turn off machine code permit

29 define compiler

The most important routines in <cSS> are:

A)<cSEXP(Z)> the expression compiler, which handles phrase types

[±'] [OPERAND] [OP'] [REST OF EXPR].

<cSEXP(Z)>compiles code to evaluate the expression in whatever mode is convenient, and leave the result in the (program) nest in the mode specified by Z.

Z=1	real
Z=2	integer
Z=3	integer if possible,

B)<cNAME (Z)> handles references to names, depending on Z:

Z=0	compile a routine call
Z=1	compile store into cell named
Z=2	compile fetch from cell named
Z=3	compile fetch address of cell named.

C) cCOND and associated routines cSC,cCC,cCOMP, Handle conditionals.

D) cRSPEC handle routine specs

E) cUI handles phrase types [UI]

Separate flowcharts are given for each of these routines.

Generally the analysis record is processed from ''left to right'', and in all of the routines, <p> is a global pointer to the part of the record currently being processed. For example in calls to < cNAME >, the name being referenced is pointed to by <A(p)> upon entry to < cNAME,>

Throughout the compilation phase a number of references are made to list processing routines:

pushdown 2
popup 2
copy tag
store tag
replace tag
find label
link
newcell
return cell
insert after 2
store name
fill label
fill set
store jump.

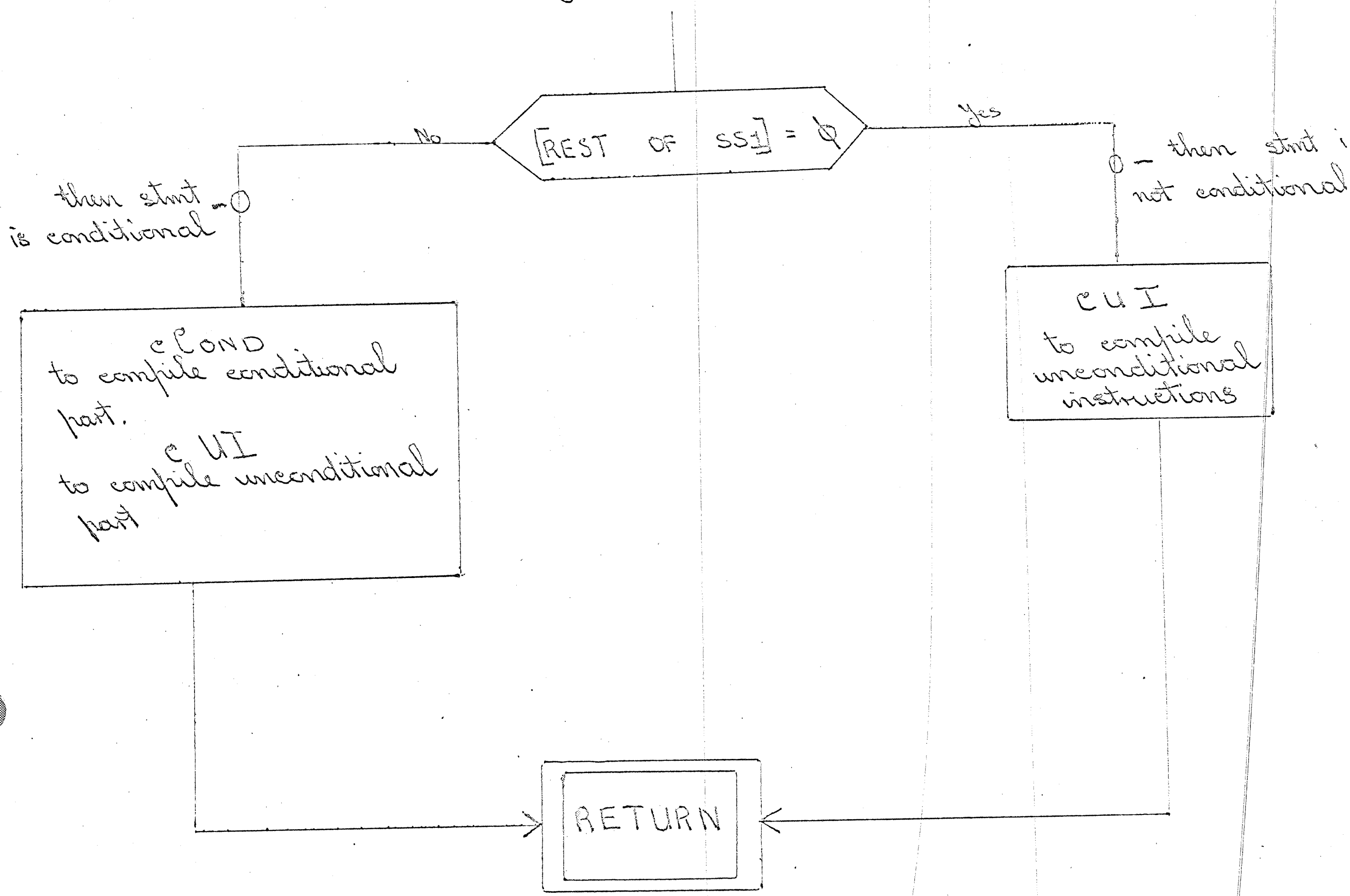
These routines are flowcharted and explained in the section ''Background Routines.''

The actual code planting routines are not explained, but the general nature of the code planting sequence is explained in the section ''Code Dumping.''

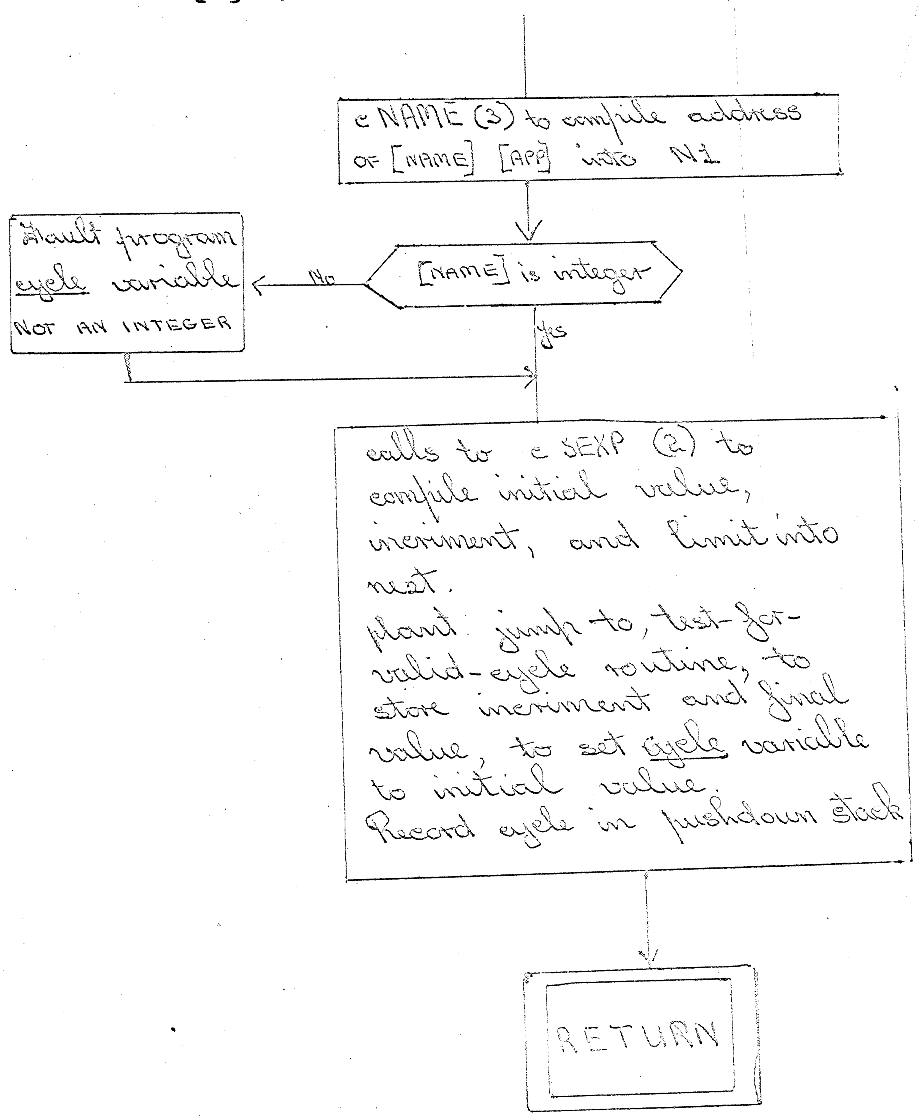
(1)

[uI] [SET MARKER 2] [REST OF SS 1]

1 Note – marker 2 points to the alternative of [REST OF SS1] which was recognised.



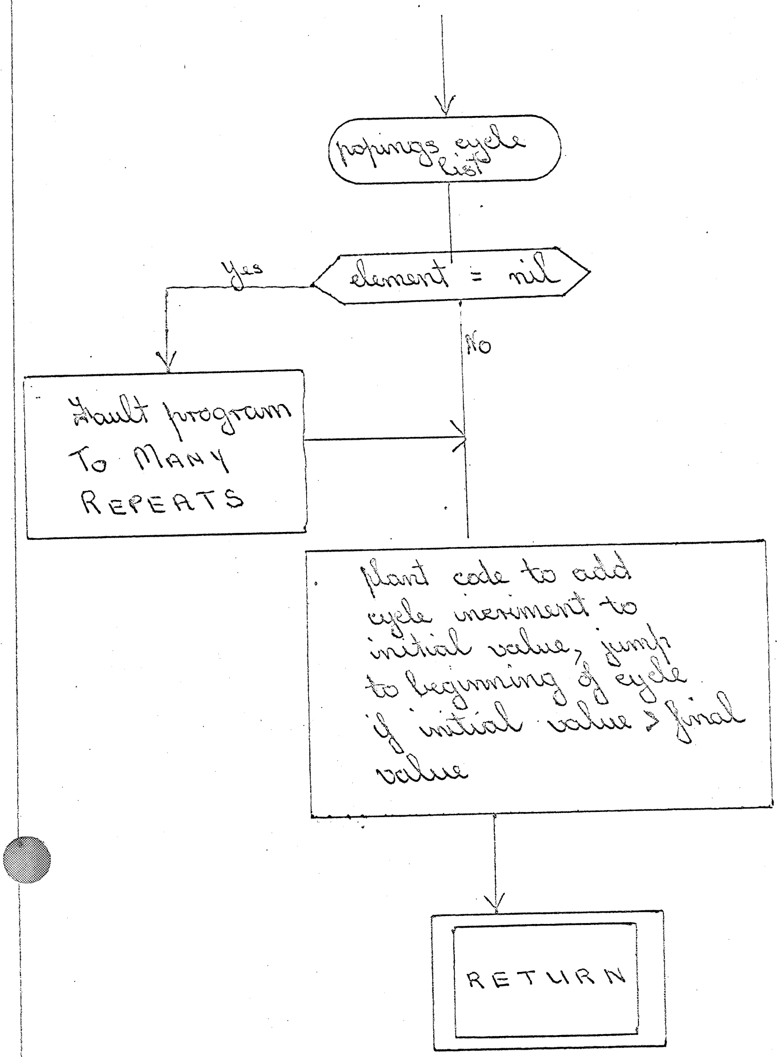
2 {cycle} [NAME] [APP] {=} [±'] [OPERAND] [REST OF EXPR] {,} [±'] [OPERAND] [REST OF EXPR] {,} [OPERAND] [REST OF EXPR] [S]



code planted is
(& cycle variable) by c NAME
= En M level
(initial value) by c SEXP
DUP
(increment) by c SEXP
DUP
= E n+1 M level
JS 5P:
En M level
= M13
j: = M0 M13

1)

{repeat} [S]

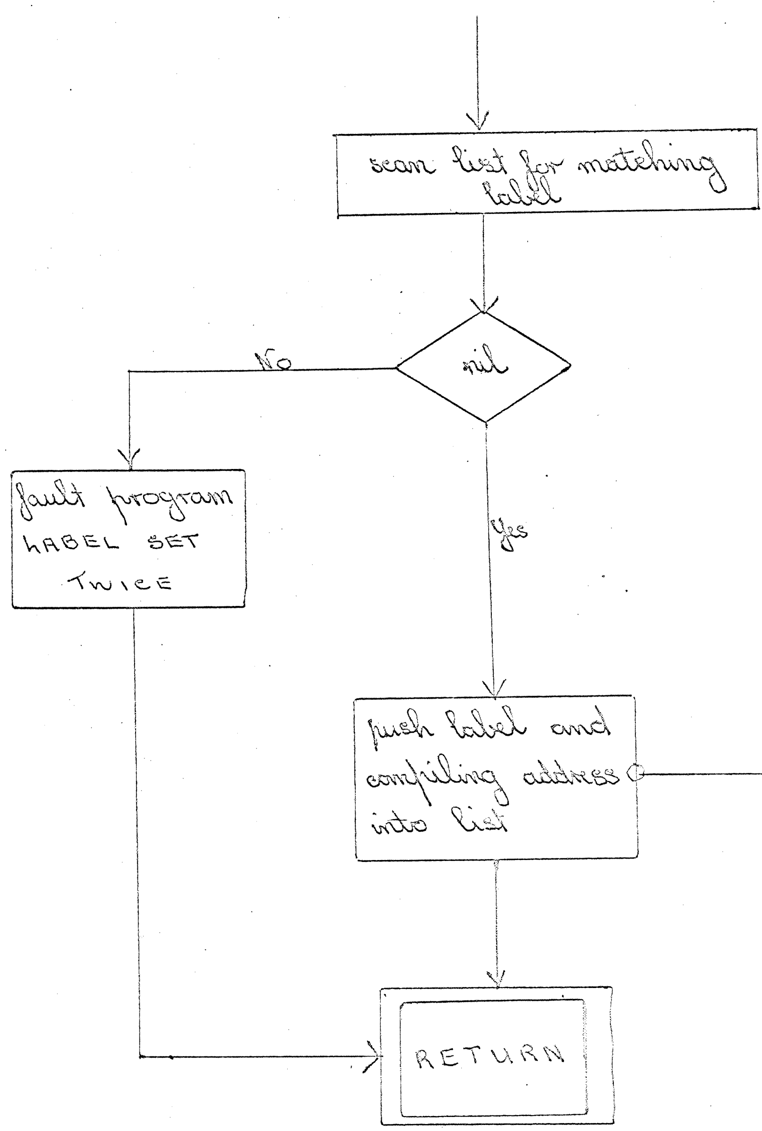


code planted is

En Mlevel	initial value address
= M13	
M0 M13	initial value
En+1 Mlevel	increment
+	add
Jj ≠	
ERASE	

Note: cycle variable replaced at j. see c SS for A(1) = 2

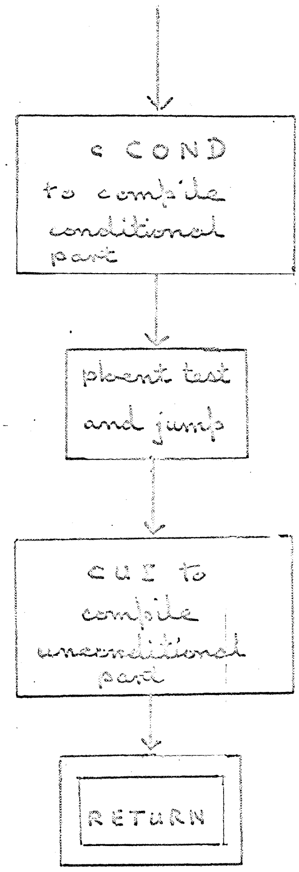
[N] {;}



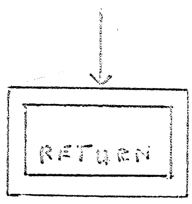
pushdown 2 (label (level), ca, k)
where k is label number

(1)

5 [_iu_] [SC] [REST OF COND] {_then_} [UI] [S]



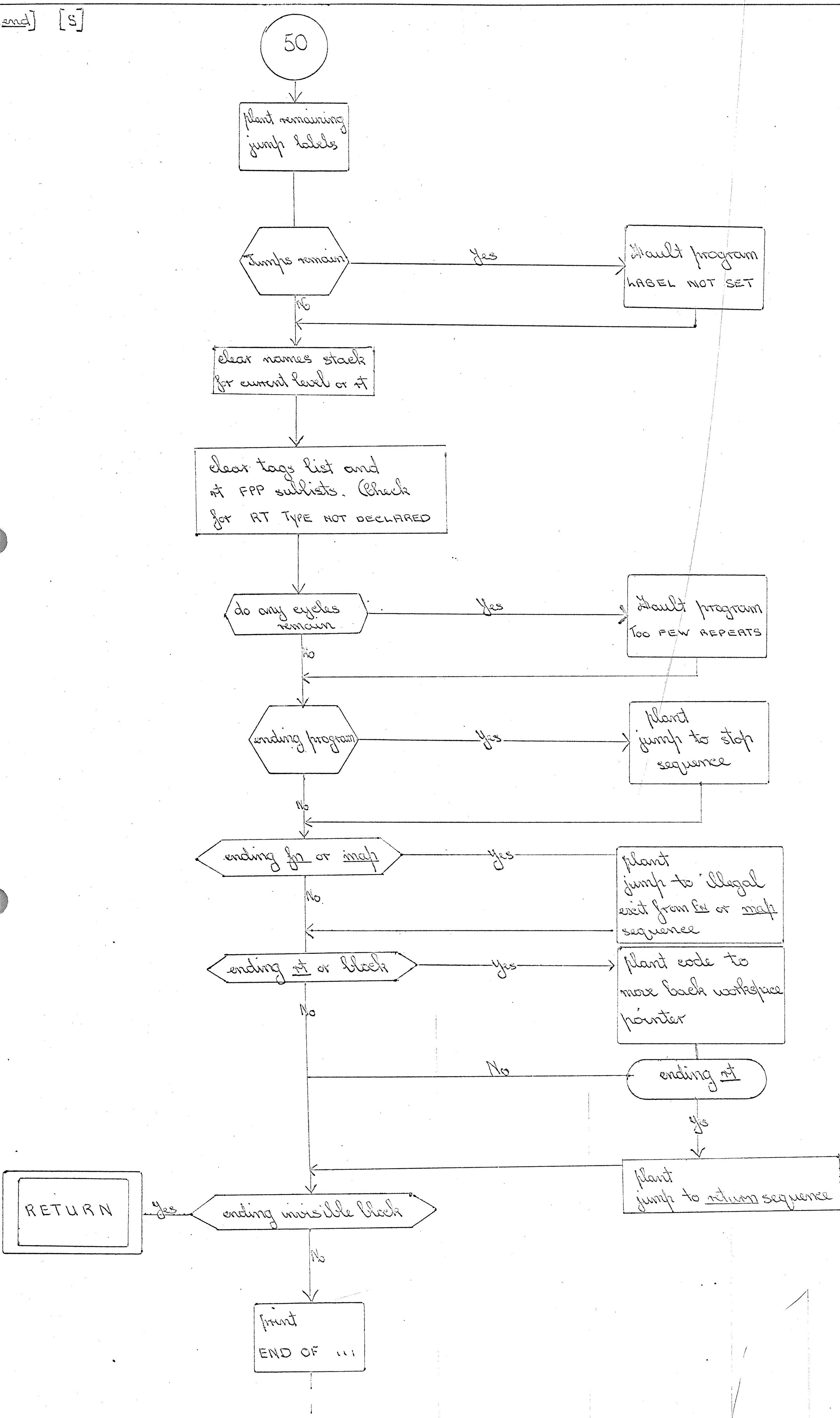
6 {!} [TEXT] [S]

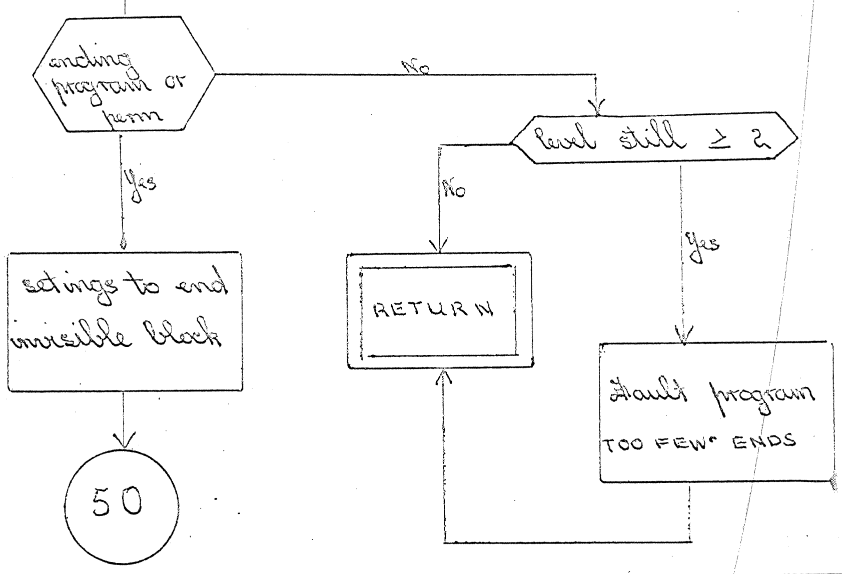


7 [TYPE] [NAME] [REST OF NAME LIST] [S]

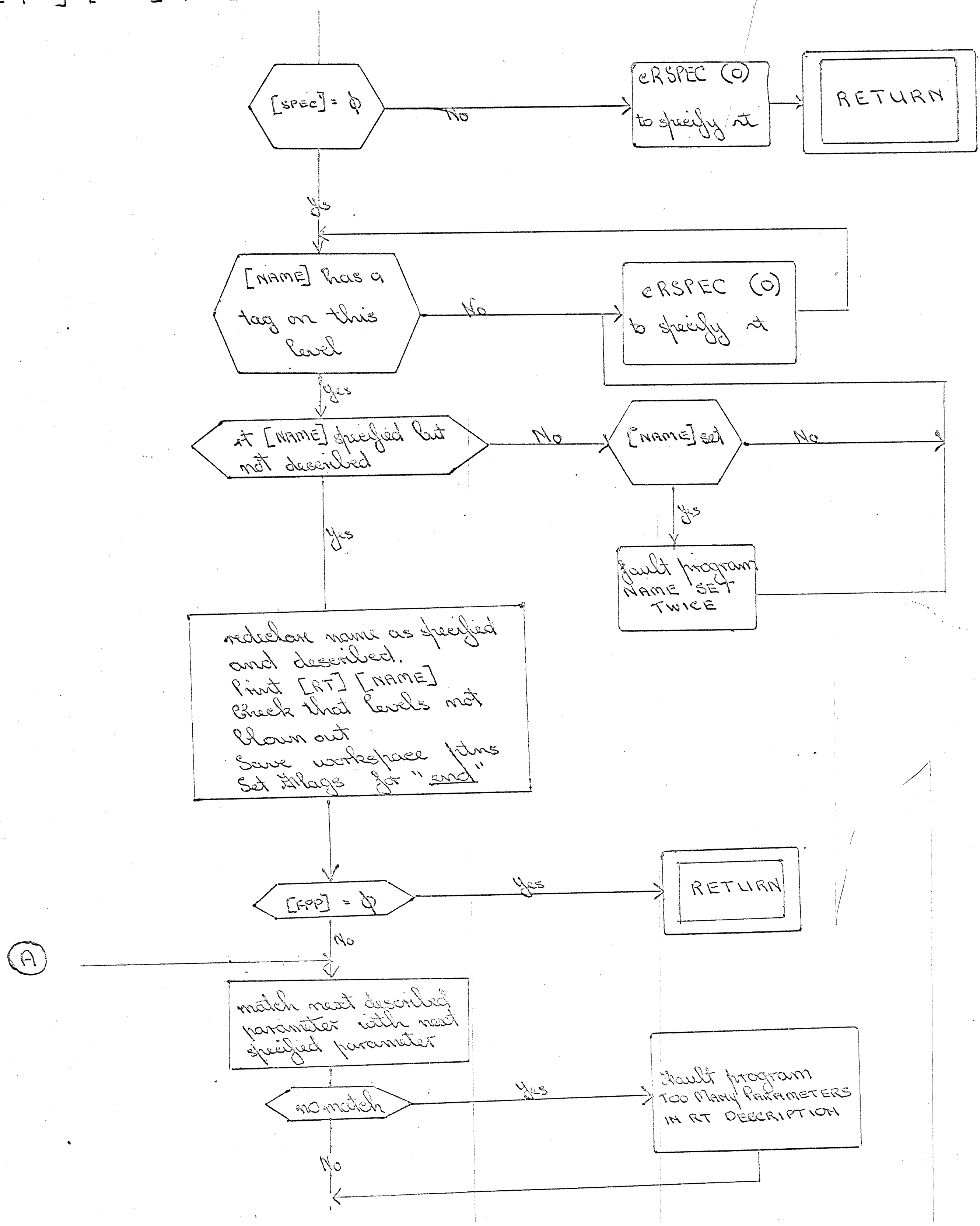
declare names, Fault program if name set twice,
set up tags array.

[end] [S]


50
plant remaining jump labels
Jumps remain
Yes
Hault program
LABEL NOT SET
No
clear names stack for current level or rt
clear tags list and rt FPP sublists. Check for RT TYPE NOT DECLARED
do any cycles remain
Yes
Hault program
TOO FEW REPEATS
No
ending program
Yes
plant jump to stop sequence
No
ending fn or map
Yes
plant jump to 'illegal exit from fn or map sequence
No
ending rt or block
Yes
plant code to move back workspace pointer
No
No
ending rt
Yes
plant jump to return sequence
RETURN
Yes
ending invisible block
No
print
END OF ...

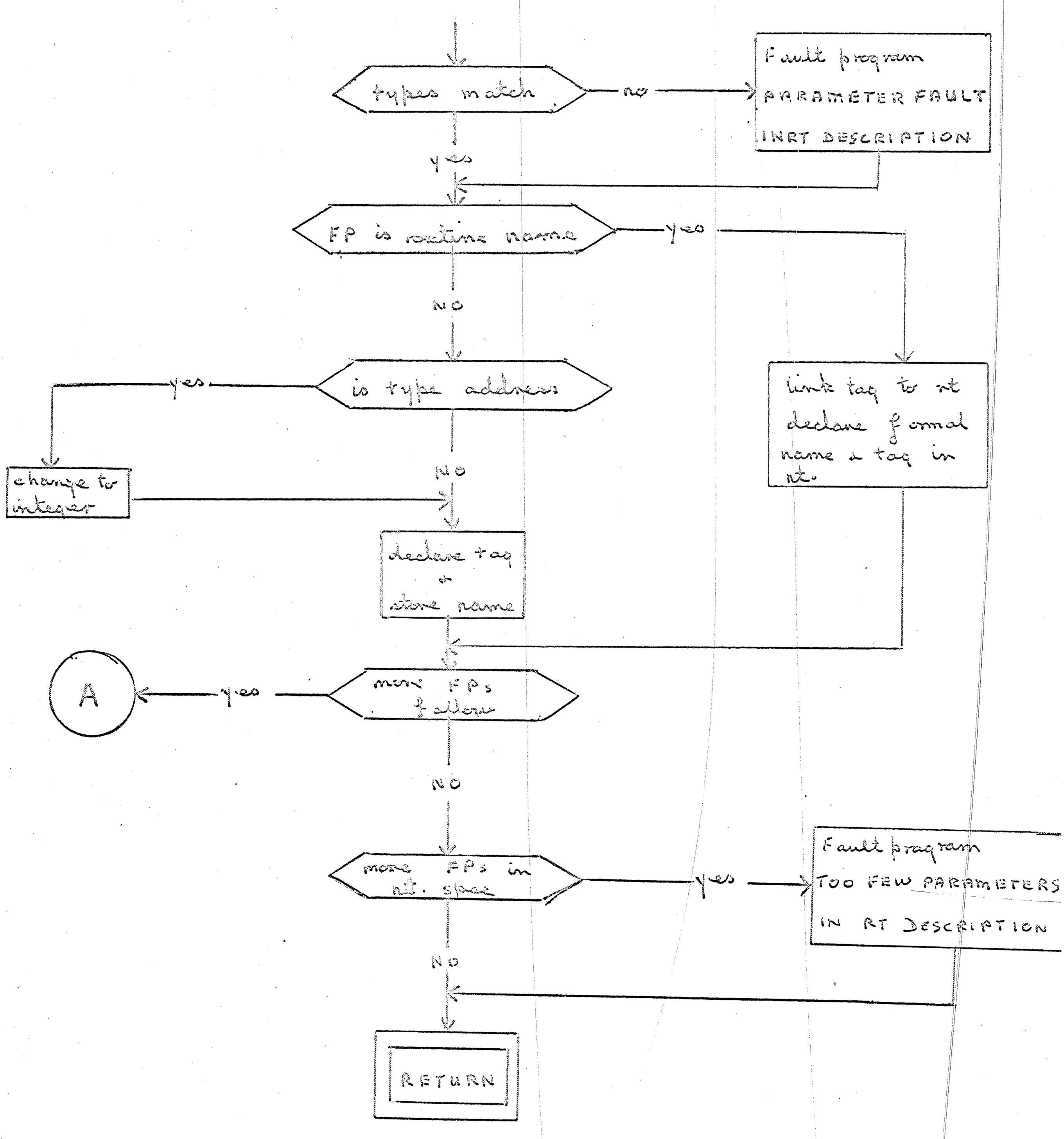


9 [RT] [spec] [NAME] [FPP] [S]

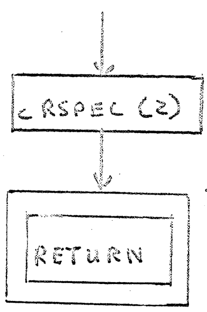


(1)

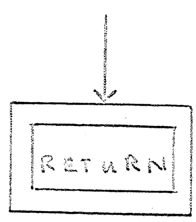
(9)
CONT



10 [spec] [NAME] [FPP] [S]

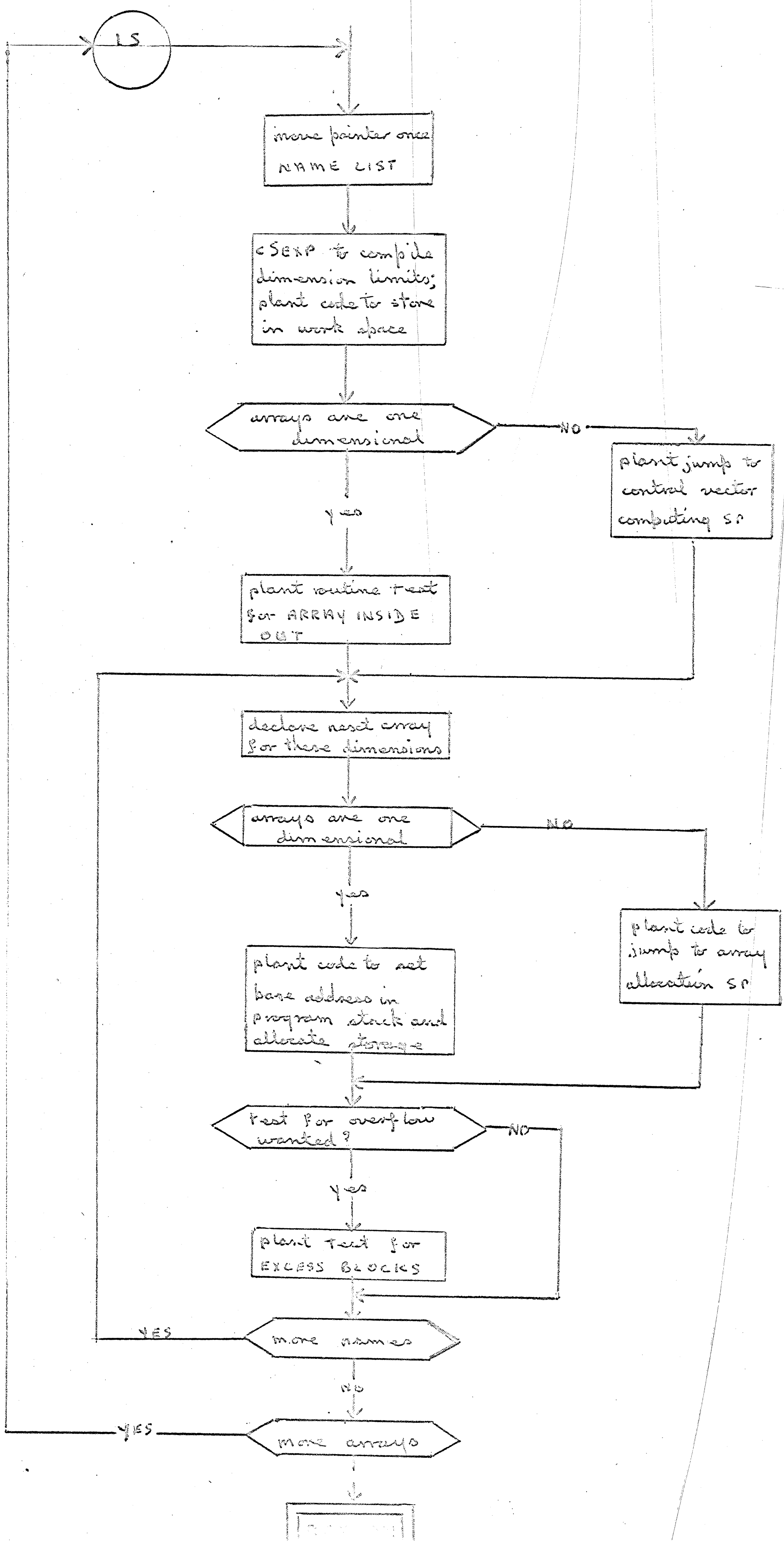


11 {comment} [TEXT] [S]



(1)

12 [TYPE'] {array} [NAME][REST OF NAME LIST] {(} [±'] [OPERAND]
[REST OF EXPR] {:} [±'] [OPERAND] [REST OF EXPR]
[REST OF BP-LIST] {)} [REST OF ARRAY LIST] [S]



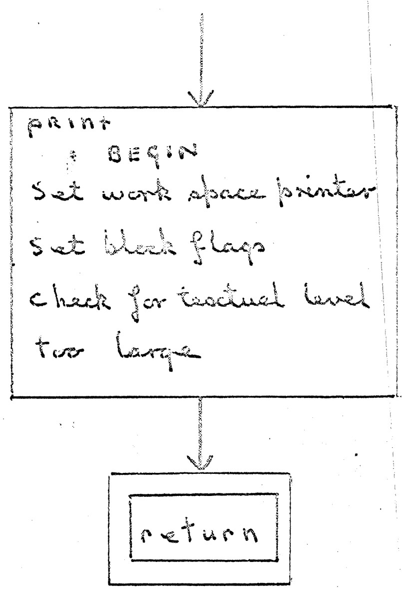
(1)

13 {*} {*} {*} {A} {S}

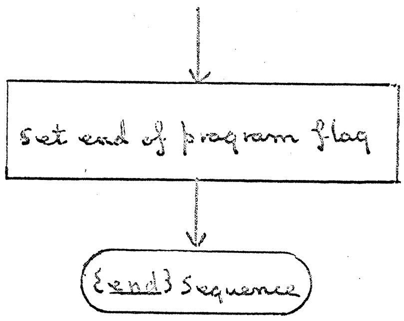
process entire job heading; output devices, magnetic tape allocation, execution times recorded.

(1)

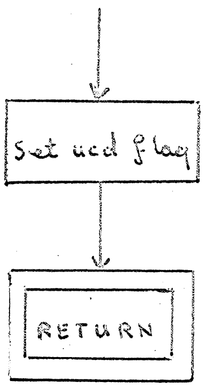
4 {begin} [S]



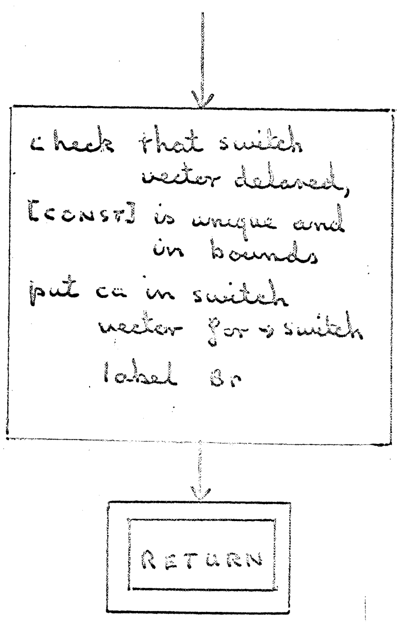
5 {end} {of} {program}



16 {upper} {case} {delimiters}



17 [NAME] {(} [±'] [CONST] {)} {:} (switch label)



(1)

22 [N]{P}{:}

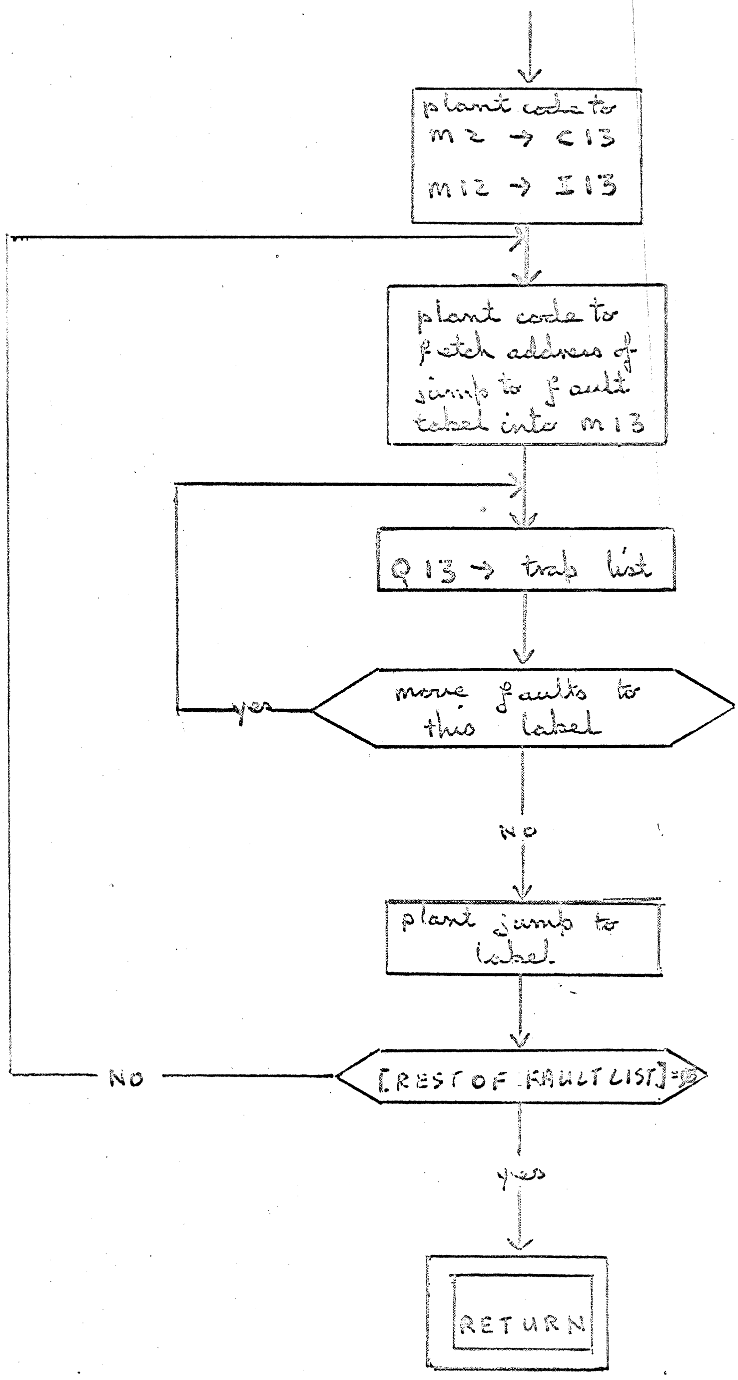
check that machine code switched on,
that P-label not already set
set label

3 {*}[UCI][S]

Fault program if machine code not switched on
c UCI to compile user code

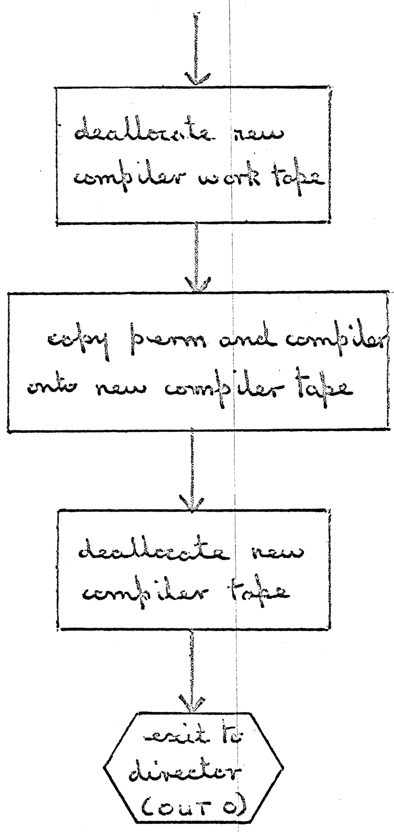
4 {<u>Fault</u>} [N][REST OF NLIST]{→}[N][REST OF FAULT LIST][S]

NOTE: When a run time fault occurs in a user program, control transfers to 90P: in perm, where a check is made to see whether the user has trapped the fault or not. If he has, there will be a list element for the fault containing restart information: the values of M2 (stack base address for level 2,) M12 (end of stack pointer for level 2) and the address of a jump instruction to the required label.



(1).

25	{normal} {delimiters} [S] clear ucd flag
26	{strings} [S] set string flag
27	{end} {of} {perm} [S] clear perm and machine code flags reset line count set overflow test permit
9	{define} {compile} {c} [S] causes execution of program in compiler which copies the perm and compiler currently in core into a magnetic tape whose device number is in E418.



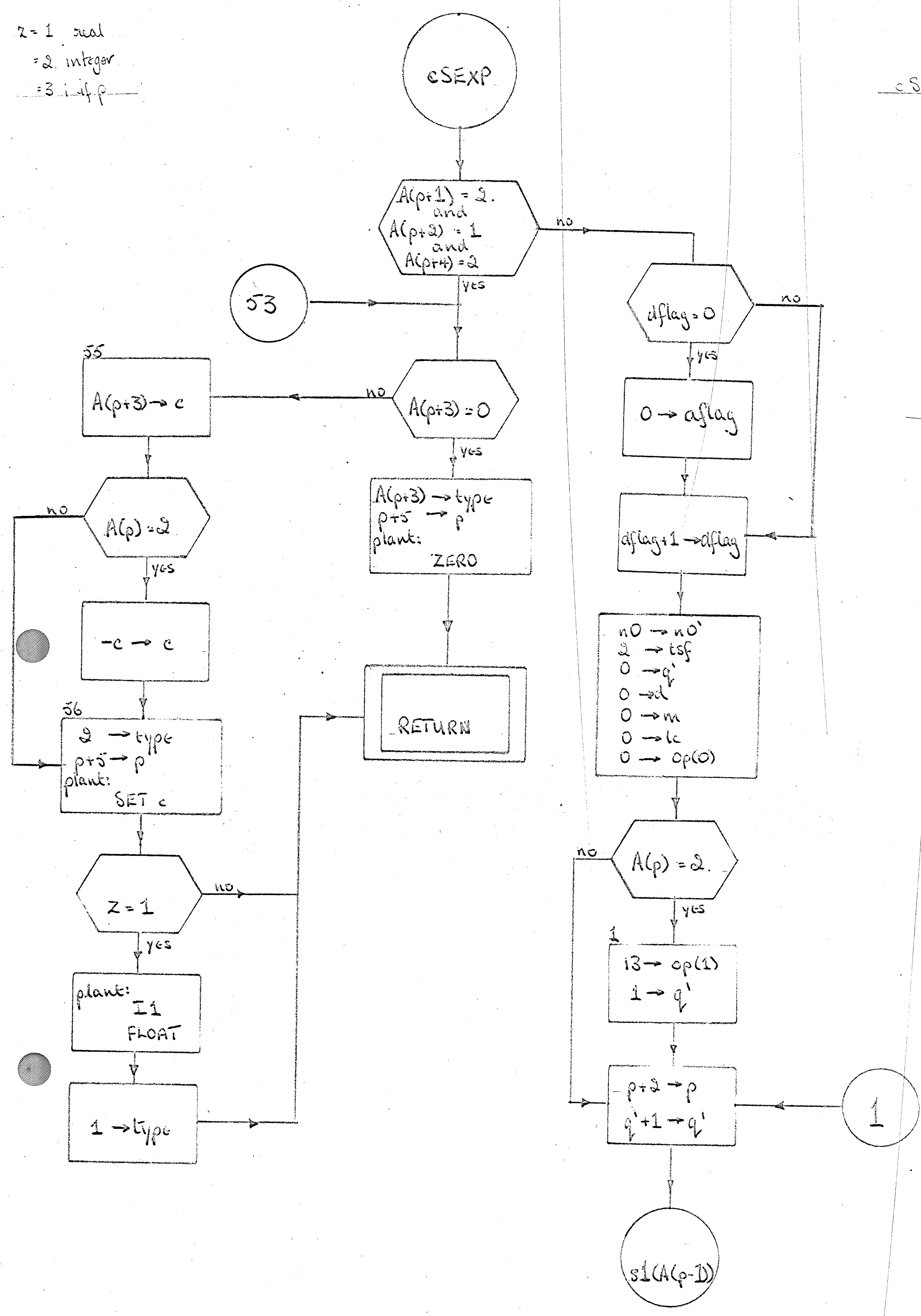
Note on the meaning of constants

dflag	depth of recursion counter
tsf	represents the current type (integer or real) of the expression being compiled. = 1 for real, = 2 for integer.
lc	long constant flag - set if a long constant is among the current operands.
op( )	array, contains the operators currently being examined.
n0, n0'	stack (ST) pointers.
m	absolute value flag, set if absolute value of an expression is to be taken.

Meaning of call parameter z

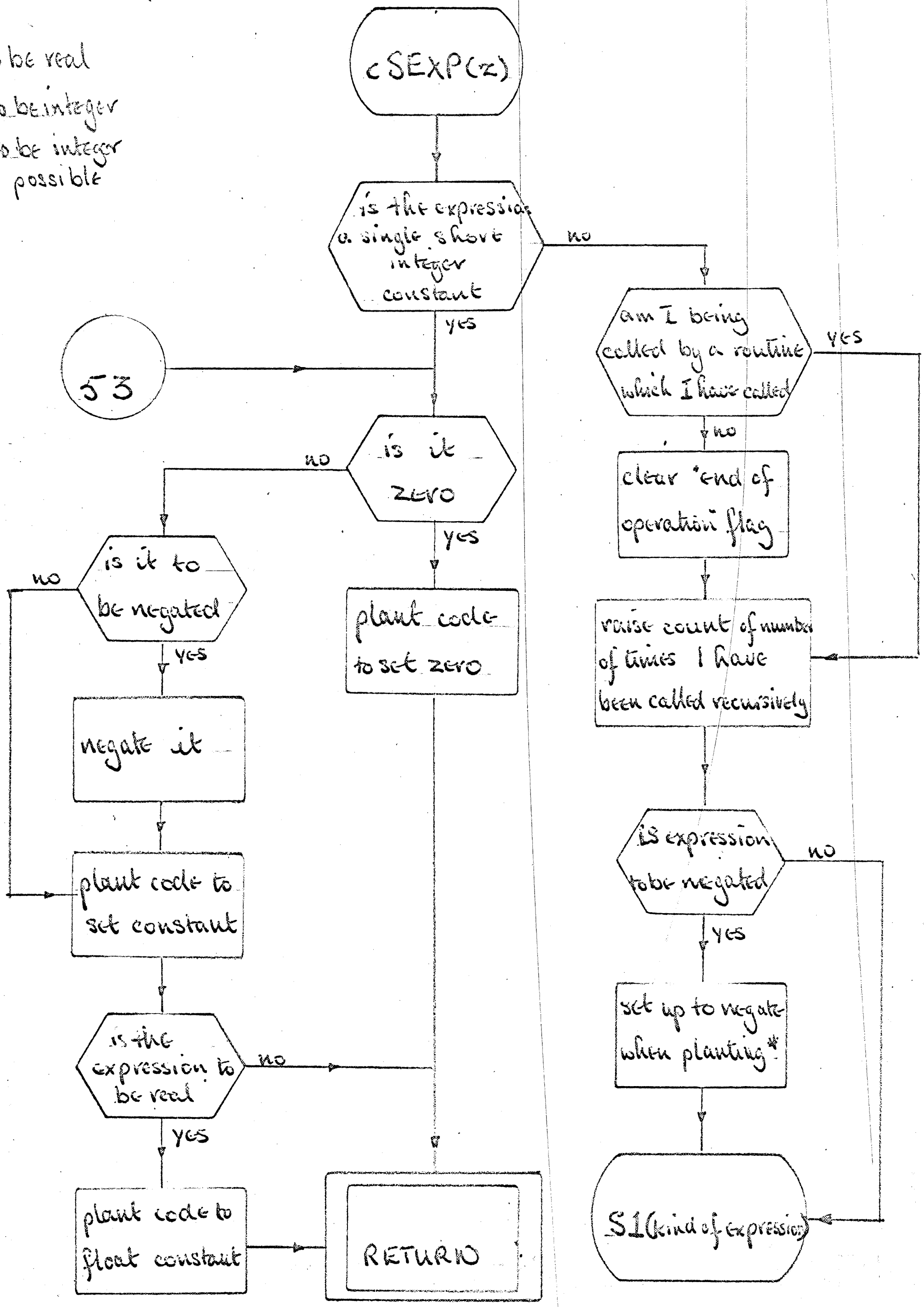
1	final value to be real
2	final value to be integer
3	final value to be integer if possible.

z = 1 real
= 2 integer
= 3 i if p

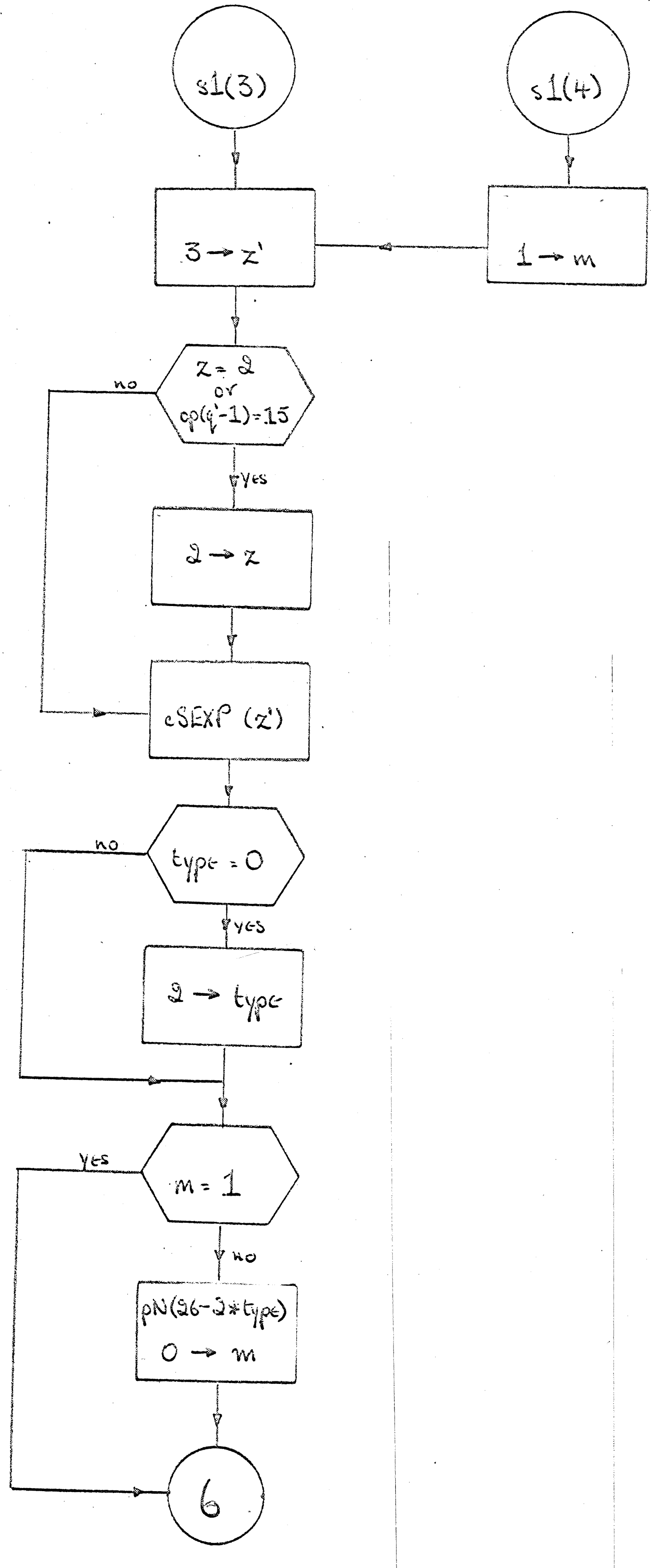


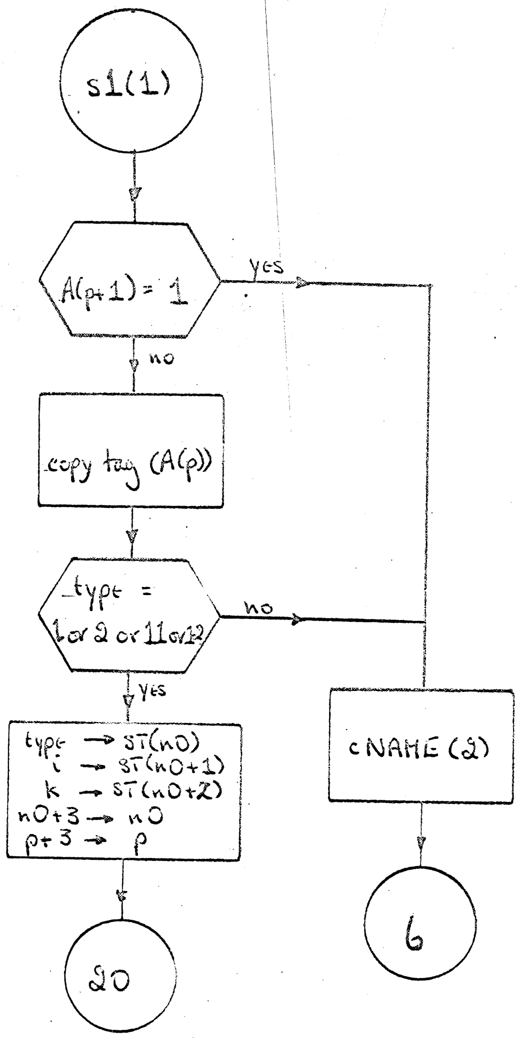
z = 1 if value is to be real
z = 2 if value is to be integer
z = 3 if value is to be integer if possible

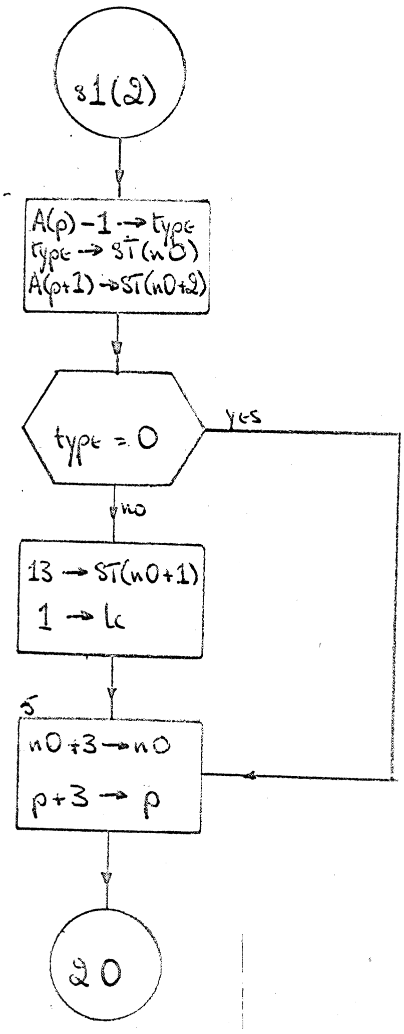
cSEXP
English



* The "set ups" are finally executed by routine plant orders. See note opposite label (30).

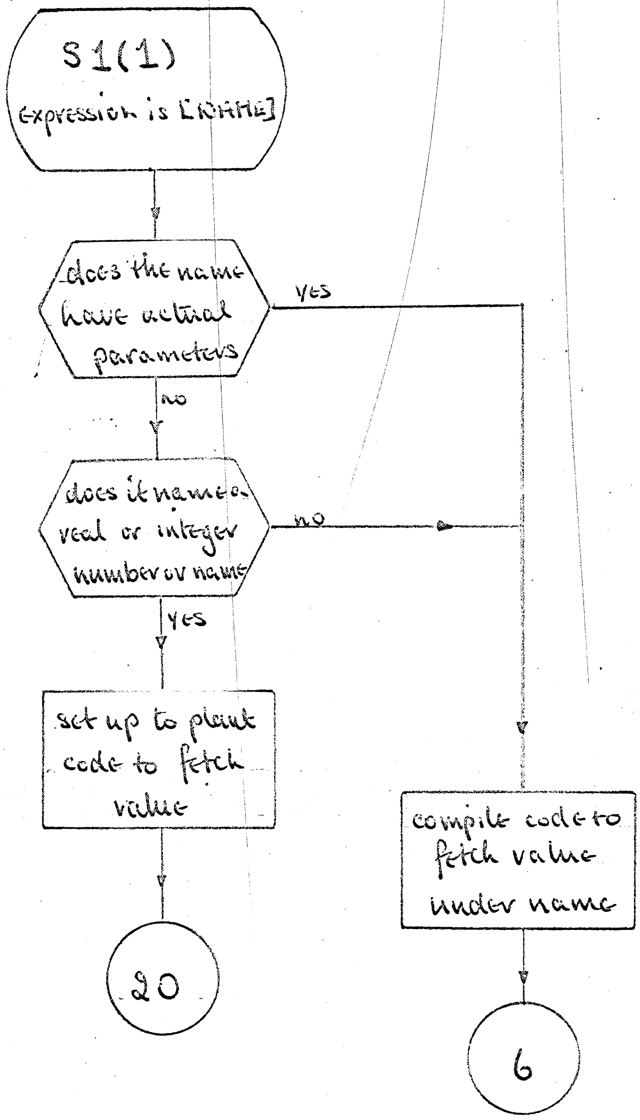

s1(3)
s1(4)
3 → z'
1 → m
z = 2 or cp(q'-1) = 15
no
yes
2 → z
cSEXP (z')
type = 0
no
yes
2 → type
m = 1
yes
no
pN(26-2*type)
0 → m
6

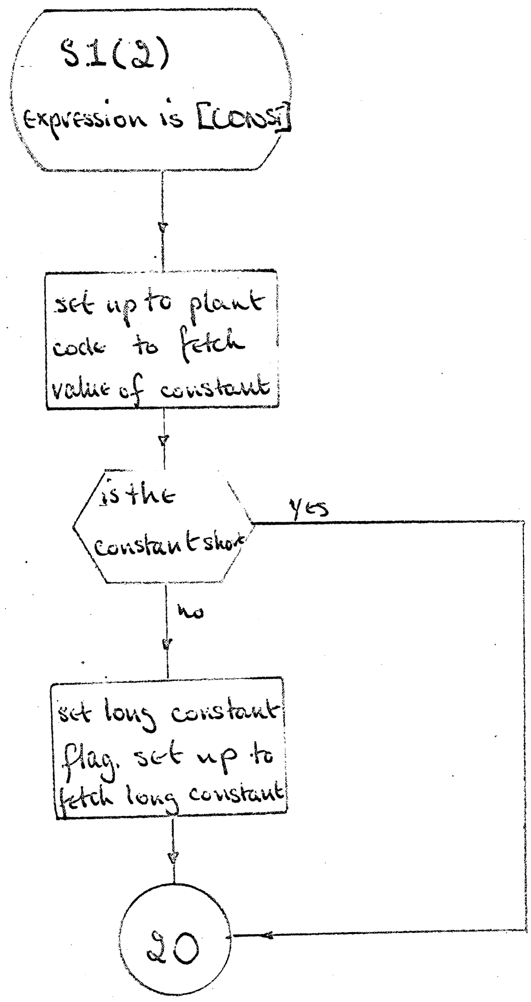

s1(1)
A(p+1) = 1
yes
no
copy tag (A(p))
type = 1 or 2 or 11 or 12
no
yes
type → ST(n0)
i → ST(n0+1)
k → ST(n0+2)
n0+3 → n0
p+3 → p
cNAME (2)
6
20

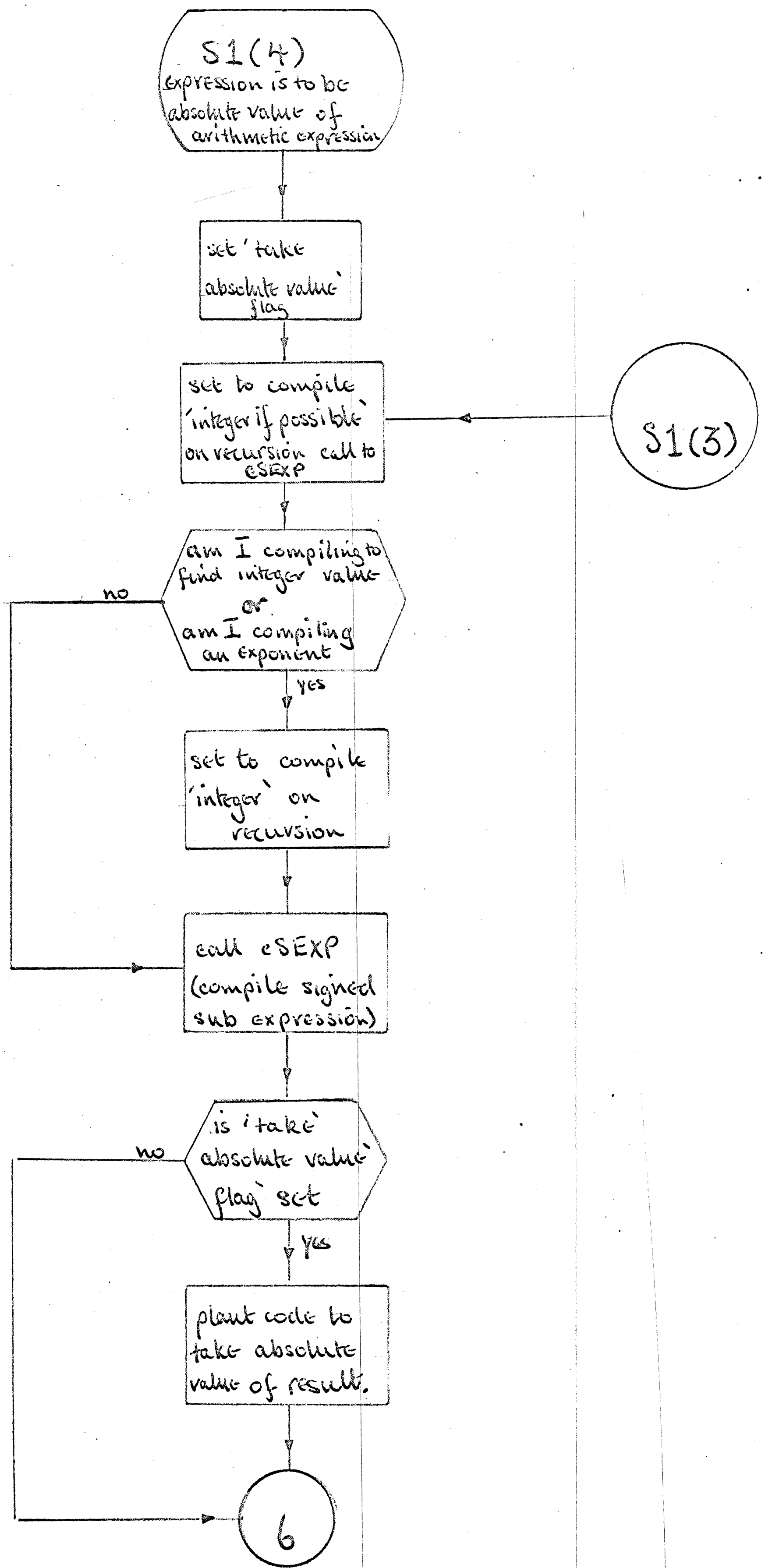

s1(2)
A(p)-1 → type
type → ST(n0)
A(p+1) → ST(n0+2)
type = 0
yes
no
13 → ST(n0+1)
1 → k
5
n0+3 → n0
p+3 → p
20

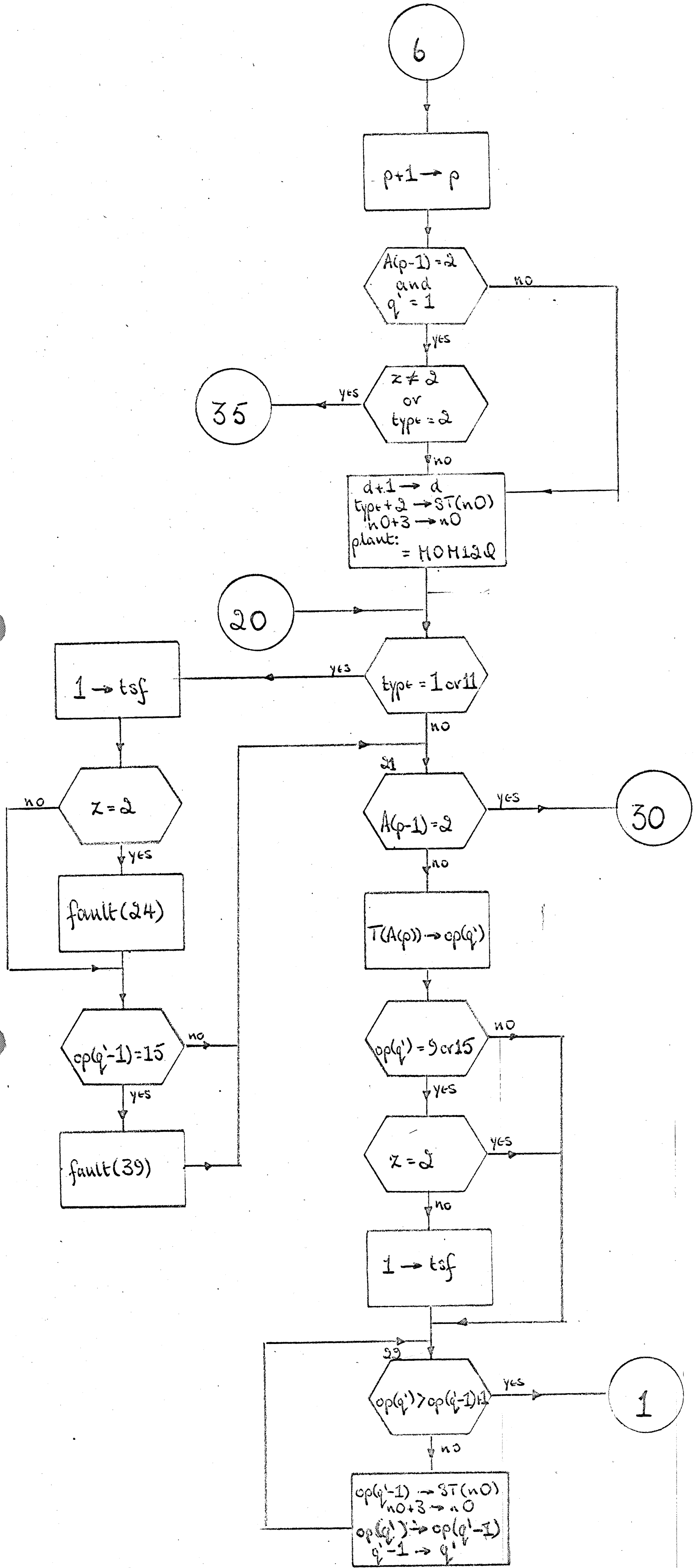
cSEXP

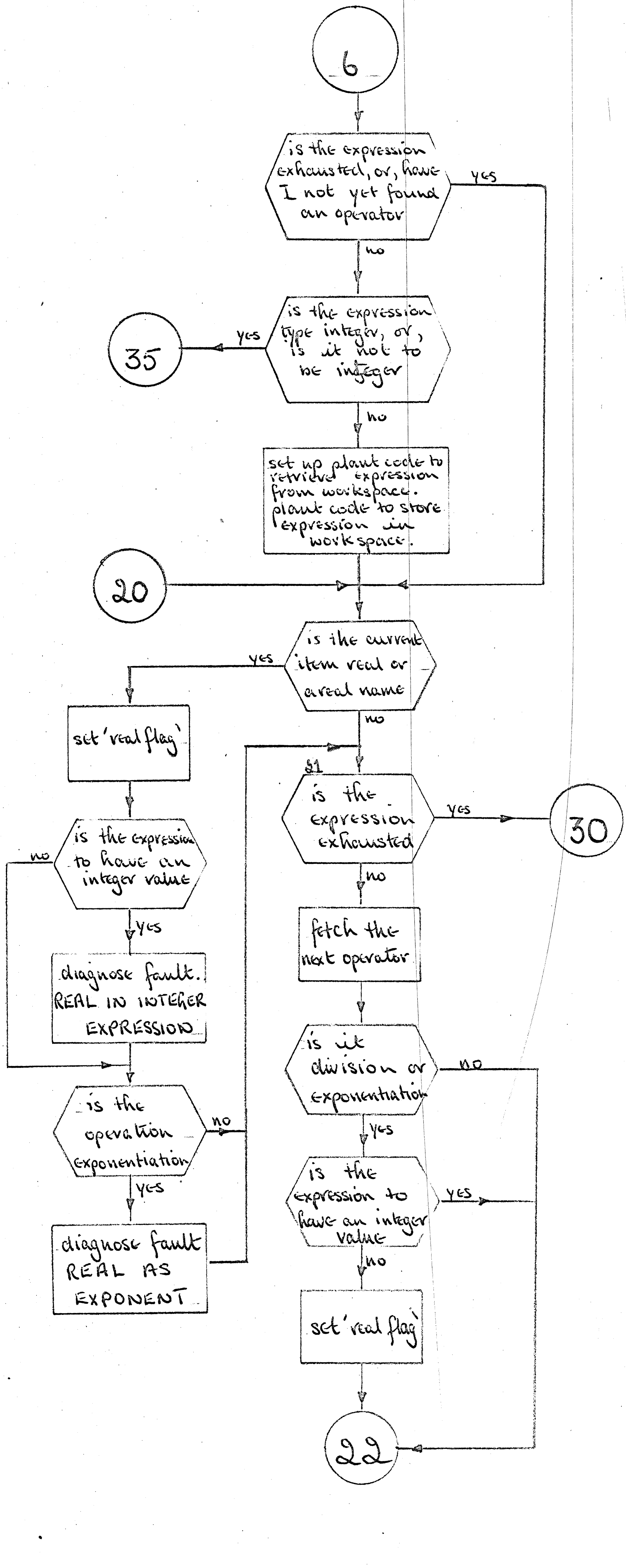
English

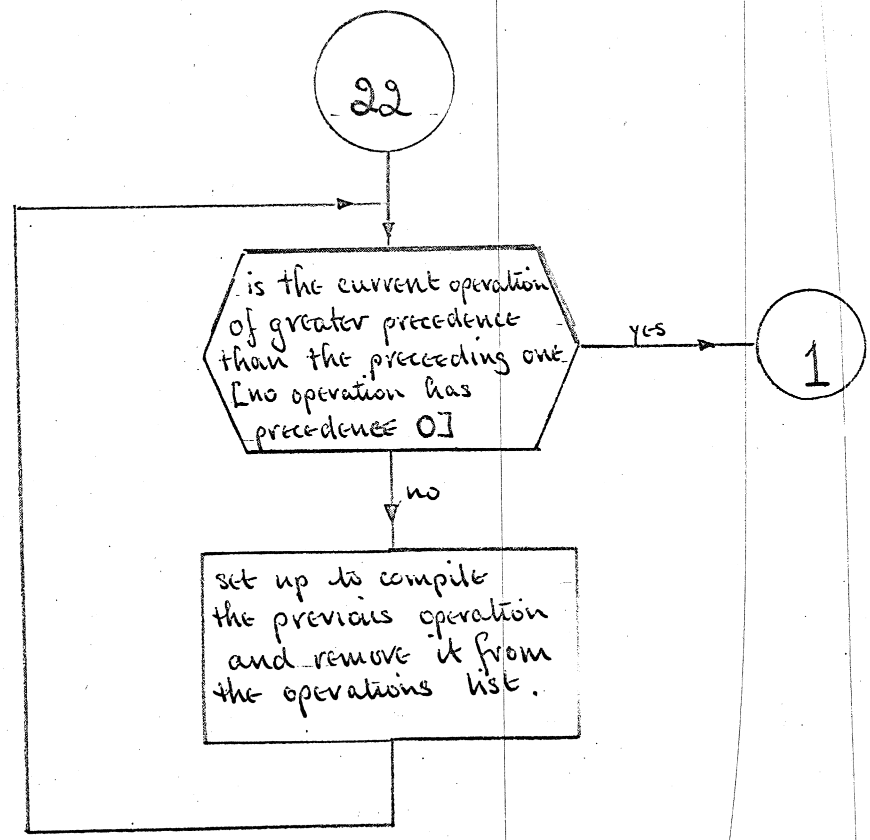


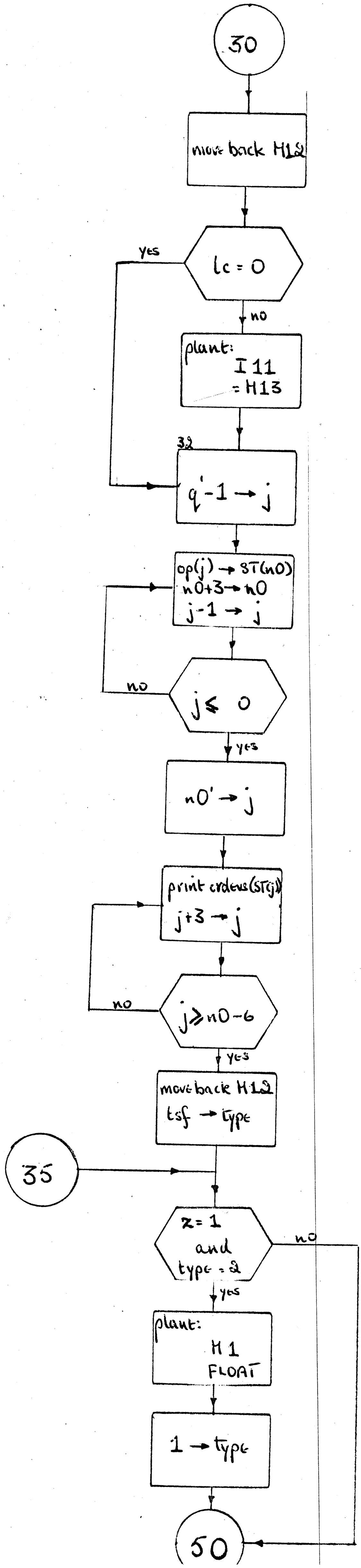



S1(4)
expression is to be absolute value of arithmetic expression
set 'take absolute value' flag
set to compile 'integer if possible' on recursion call to cSEXP
S1(3)
am I compiling to find integer value or am I compiling an exponent
no
yes
set to compile 'integer' on recursion
call cSEXP (compile signed sub expression)
is 'take absolute value' flag set
no
yes
plant code to take absolute value of result.
6


6
p+1 → p
A(p-1) = 2 and q' = 1
no
yes
z ≠ 2 or type = 2
yes
35
no
d+1 → d
type+2 → ST(n0)
n0+3 → n0
plant: = HOH12Q
20
type = 1 or 11
yes
1 → tsf
no
z = 2
no
yes
fault(24)
op(q'-1) = 15
no
yes
fault(39)
21
A(p-1) = 2
yes
30
no
T(A(p)) → op(q')
op(q') = 9 or 15
no
yes
z = 2
yes
no
1 → tsf
99
op(q') > op(q'-1)+1
yes
1
no
op(q'-1) → ST(n0)
n0+3 → n0
op(q') → op(q'-1)
q'-1 → q'


6
is the expression exhausted, or, have I not yet found an operator
yes
no
is the expression type integer, or, is it not to be integer
yes
35
no
set up plant code to retrieve expression from workspace. plant code to store expression in workspace.
20
is the current item real or a real name
yes
no
set 'real flag'
31
is the expression exhausted
yes
30
no
is the expression to have an integer value
no
yes
diagnose fault. REAL IN INTEGER EXPRESSION
fetch the next operator
is it division or exponentiation
no
yes
is the operation exponentiation
no
yes
diagnose fault REAL AS EXPONENT
is the expression to have an integer value
yes
no
set 'real flag'
22


22
is the current operation of greater precedence than the preceeding one [no operation has precedence 0]
yes
1
no
set up to compile the previous operation and remove it from the operations list.


30
move back H12
lc = 0
yes
no
plant:
I 11
= H13
32
q'-1 → j
op(j) → ST(n0)
n0+3 → n0
j-1 → j
no
j ≤ 0
yes
n0' → j
print ordeus(ST(j))
j+3 → j
no
j ≥ n0-6
yes
move back H12
tsf → type
35
z = 1
and
type = 2
no
yes
plant:
H 1
FLOAT
1 → type
50

'Setting up' an expression for compilation consists in forming a list of operators and operands in the stack. The list is ordered in reverse polish, so that b+c * a ↑ b/e would become

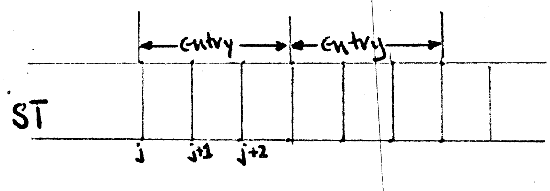
b c a b ↑ * e / +

The following code will be planted (assuming the variables to be integer)

```
** b
** c
** a
** b
*  JS  82P  (exponentiation subroutine)
*  XD
*  CONT
** e
*  ÷
*  +
```

Expressions may always be treated in this way since parenthetic sub-expressions are compiled by recursive calls to cSEXP, and the results stored in temporay workspace to be fetched when needed.

The list of operands and operations is kept on part of the ST array. Each entry, whether representing an operand or an operation consists of three words opp 6



The first word of each entry identifies the operation or the operand represented by the entry. The next two words give other information about the entry. The entry codes are as follows:

ST(j)	ST(j+1)	ST(j+2)
0 short integer	value	sign: 5 + 6 -
1, 2, real or integer	address of value	...
3, 4, array elements	address of value	...
5 add operator	...	...
6 subtract operator	...	...
8 multiply operator	...	...
9 divide operator	...	...
13 negate operator	...	...
15 exponentiate operator	...	...
11, 12 real or integer name	address of name	...
14 address	...	...

The stack in this form is interpreted by routine print orders, and the corresponding code is planted.

The compiler itself is phrase structure oriented and compiles individual source statements one at a time. Compiler operates in two phases, a recognition phase and a compilation phase.

During the recognition phase, a source statement is compared with a list of permitted statement forms. If the statement matches a particular form, it is said to be syntactically correct and therefore ready for compilation. As a statement is being matched, a list of pointers is made which describe the ''path'' through the phrase structure which resulted in a match. The phrase structure itself is a recursive tree structure whose roots are either text literals or ''built-in-phrases''. Phrase structure components are listed on the next several pages.

Built in phrases (or b.i.p.s) are phrases defined by pieces of compiler program. Recognizing a built in phrase in a source statement involves executing one of these pieces of program. The built in phrases are

[NAME]	all program names
[CONST]	decimal constants
[N]	labels and short decimal constants
[OCTAL]	octal constants
[TEXT]	comment text
[CAPTION TEXT]	
[S]	end-of-statement:[redacted]semicolon or newline
[SET MARKER 1]	to indicate break between conditional and unconditional statement parts
[SET MARKER 2]	to indicate break between L.H.S. and R.H.S. of expression statements.

Each phrase, or P-word(symbolically 'P[...]') is either defined in terms of P-words and text literals, or is a built in phrase. The definitions, which are actually stored in the compiler, are written

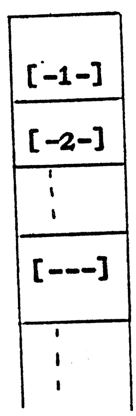
P[---] = ..., ..., ... ;

where '[---]' represents an identifier, and '...' may represent a <u>string</u> of P-word identifiers and text literals. * A phrase definition says that the phrase [---] may consist of ... exclusively, or ... exclusively, or ..., etc. A semicolon terminates the definition. The definition parts separated by commas are called alternatives, and, when a source statement is scanned, a part will be recognised as of the k <u>th</u> alternative of [---] if it matches the kth alternative ....

A null alternative (symbolically 'ø') is possible in many definitions. As A phrase definition is scanned from left to right, as it were, and when a match with the source statement part is found, scanning stops. 'ø' as the k<u>th</u> alternative acts as an instruction, ''match the source statement part as alternative k of this phrase.''

Phrase definitions are stored as a list, in the compiler, each definition occupying a contigmous part of the array < symbol (1300:2320).>

<symbol>



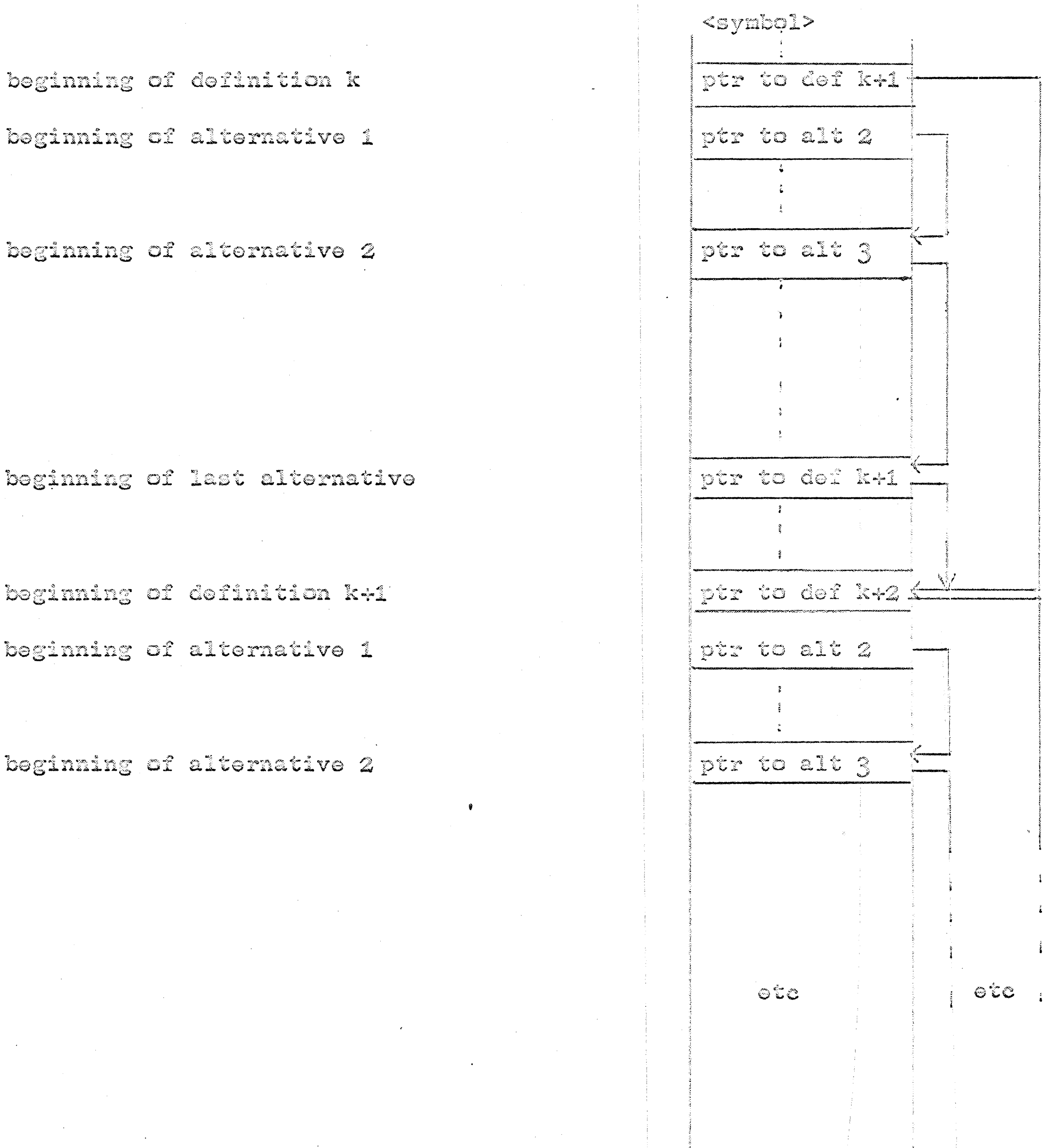
*Text literals are written between spiked brackets '⊏' and '⊐' . The desired text appears between them.

```
P[±']                    =   ⊦+⊣,⊦-⊣,ø;
P[OPERAND]               =   [NAME][APP],[CONST],⊦(⊣[±'][OPERAND][REST OF EXPR]⊦)⊣,
                             ⊦|⊣[±'][OPERAND][REST OF EXPR]⊦|⊣;
P[REST OF EXPR]          =   [OP][OPERAND][REST OF EXPR],ø;
P[APP]                   =   ⊦(⊣[±'][OPERAND][REST OF EXPR][REST OF EXPR-LIST]⊦)⊣,ø;
P[REST OF EXPR-LIST]     =   ⊦,⊣[±'][OPERAND][REST OF EXPR][REST OF EXPR-LIST],ø;
P[OP]                    =   ⊦+⊣⊦-⊣,⊦*⊣,⊦/⊣,⊦↑⊣,ø;
P[QUERY']                =   ⊦?⊣,ø;
P[,']                    =   ⊦,⊣,ø;
P[iu]                    =   ⊦if⊣,⊦unless⊣;
P[real']                 =   ⊦real⊣,ø;
P[TYPE]                  =   ⊦integer⊣,⊦real⊣;
P[TYPE']                 =   ⊦integer⊣,⊦real⊣ø;
P[RT]                    =   ⊦routine⊣,⊦real⊣⊦fn⊣,⊦integer⊣⊦fn⊣,⊦real⊣⊦map⊣,⊦integer⊣
                               ⊦map⊣;
P[FP-DELIMITER]          =   [RT],⊦integer⊣⊦array⊣⊦name⊣,⊦integer⊣⊦name⊣,⊦integer⊣,
                               [real']⊦array⊣⊦name⊣,⊦real⊣⊦name⊣,⊦real⊣,⊦addr⊣;
P[FPP]                   =   ⊦(⊣[FP-DELIMITER][NAME][REST OF NAME LIST][REST OF FP-LIST]
                               ⊦)⊣,ø;
P[REST OF FP-LIST]       =   [,'][FP-DELIMITER][NAME][REST OF NAME LIST][REST OF FP-LIST]
P[REST OF NAME LIST]     =   ⊦,⊣[NAME][REST OF NAME LIST],ø;
P[SC]                    =   [±'][OPERAND][REST OF EXPR][COMP][±'][OPERAND][REST OF EXPR]
                               [REST OF SC],
                               ⊦(⊣[SC][REST OF COND]⊦)⊣;
P[REST OF SC]            =   [COMP][±'][OPERAND][REST OF EXPR],ø;
P[REST OF COND]          =   ⊦and⊣[SC][REST OF AND-C],⊦or⊣[SC][REST OF OR-C],ø;
P[REST OF AND-C]         =   ⊦and⊣[SC][REST OF AND-C],ø;
P[REST OF OR-C]          =   ⊦or⊣[SC][REST OF OR-C],ø;
P[REST OF UI]            =   ⊦=⊣[±'][OPERAND][REST OF EXPR][QUERY'],ø;
P[spce']                 =   ⊦spec⊣,ø;
P[REST OF BP-LIST]       =   ⊦,⊣[±'][OPERAND][REST OF EXPR]⊦:⊣[±'][OPERAND][REST OF EXPR]
                               [REST OF BP-LIST],ø;
P[REST OF ARRAY LIST]    =   ⊦,⊣[NAME][REST OF NAME LIST]⊦(⊣[±'][OPERAND][REST OF EXPR]⊦:⊣
                               [±'][OPERAND][REST OF EXPR][REST OF BP-LIST]⊦)⊣[REST OF
                               ARRAY LIST],ø;
P[REST OF SWITCH LIST] = ⊦,⊣[NAME][REST OF NAME LIST]⊦(⊣[±'][CONST]⊦:⊣[±'][CONST]⊦:⊣
                               [±'][CONST]⊦)⊣[REST OF SWITCH LIST],ø;
P[COMP]                  =   ⊦=⊣,⊦≥⊣,⊦>⊣,⊦≠⊣,⊦<⊣,⊦≤⊣;
P[REST OF SS1]           =   [S],[iu][SC][REST OF COND][S];
P[REST OF N-LIST]        =   ⊦,⊣[N][REST OF N-LIST],ø;
P[REST OF FAULT LIST]    =   ⊦,⊣[N][REST OF N-LIST]⊦->⊣[N][REST OF FAULT LIST],ø;
```

```
P[UI]          =    [NAME][APP][SET MARKER 1][REST OF UI],
                    [->-][N],
                    [-caption-][CAPTION TEXT],
                    [-return-],
                    [-result=-][±'][OPERAND][REST OF EXPR],
                    [-stop-],
                    [->-][NAME][-(-][±'][OPERAND][REST OF EXPR][-)-],
                    [-monitor-][N];

P[SS]          =    [UI][SET MARKER 2][REST OF SS1],
                    [-cycle-][NAME][APP][-=-][±'][OPERAND][REST OF EXPR][-,-]
                        [±'][OPERAND]
                       [REST OF EXPR][-,-][±'][OPERAND][REST OF EXPR][S],
                    [-repeat-][S],
                    [N][-:-],
                    [iu][SC][REST OF COND][-then-][UI][S],
                    [-|-][TEXT],
                    [TYPE][NAME][REST OF NAME LIST][S],
                    [-end-][S],
                    [RT][spec'][NAME][FPP][S],
                    [-spec-][NAME][FPP][S],
                    [-comment-][TEXT],
                    [TYPE'][-array-][NAME][REST OF NAME LIST][-(-][±'][OPERANI
                        [REST OF EXPR][-:-]
                        [±'][OPERAND][REST OF EXPR][REST OF BP-LIST][-)-]
                        [REST OF ARRAY LIST][S],
                    [-*-][-*-][-*-][-A-][S],
                    [-begin-][S],
                    [-end-][-of-][-program-],
                    [-upper-][-case-][-delimiters-][S],
                    [NAME][-(-][±'][CONST][-)-][-:-],
                    [-switch-][NAME][REST OF NAME LIST][-(-][±'][CONST][-:-]
                        [±'][CONST][-)-][REST OF SWITCH LIST][S],
                    [-compile-][-queries-][S],
                    [-ignore-][-queries-][S],
                    [-mcode-][S],
                    [N][-P-][-:-],
                    [-*-][UCI][S],
                    [-fault-][N][REST OF N-LIST][->-][N][REST OF FAULT LIST][
                    [-normal-][-delimiters-][S],
                    [-strings-][S],
                    [-end-][-of-][-perm-][S],
                    [-end-][-of-][-mcode-][S],
                    [-define-][-compile-][-r-][S],
                    [S];
```

built in phrase	action	analysis record entry
[NAME]	File text of name in name list. n=ptr.to entry in name list	n
[CONST]	If constant is a short (≤15bits) integer, j=1, k=value If constant is long integer, j=3, value is stacked, k=address of value If constant is real, j=2, value is stacked k=address of value	j k
[OCTAL] or [N]	v=value	v
[TEXT]	(comment text ignored)	none
[SET MARKER 1] [SET MARKER 2]	internal pointers set to point to next empty element of analysis record; marker 1 if L.H.S. of expression to follow marker 2 if condition for previous part to follow	none
[CAPTION TEXT]	caption text is stacked. k=ptr to text in stack	k
[S]	none	none
∅	recognise this alternative	none

A definition is delimited by means of a pointer stored at the beginning of the array part for that definition. It points to the beginning of the next definition in the array. The alternatives of a definition are treated in the same manner, the first computer word of an alternative being a pointer to the first computer word of the next alternative:



An alternative is composed of a string of phrase identifiers and text literals. These are represented in <symbol> as a string of pointers. There are three kinds of these pointers: (1) to an entry in the dictionary of text literals, (2) to pieces of compiler code for built in phrases, and (3) to phrase definitions in <symbol> for phrases which are defined.

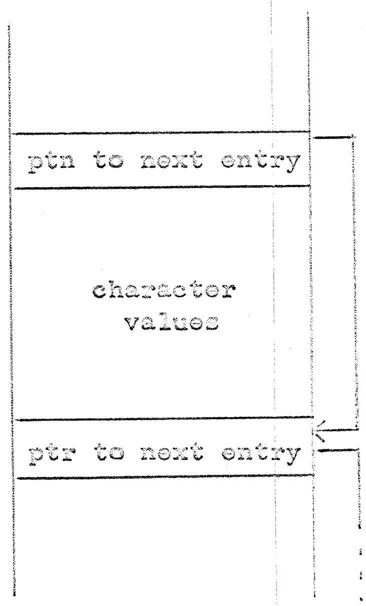
(In fact, the pointers are integer numbers rather than addresses. They are distinguished in the following way. If n is the value of an entry in symbol, then

(1) if $0 \leq n < 1000$, n points to the literal dictionary.

(2) if $1000 \leq n < 1300$, n points to a built in phrase

(3) if $n \geq 1300$, n points to a phrase definition in symbol

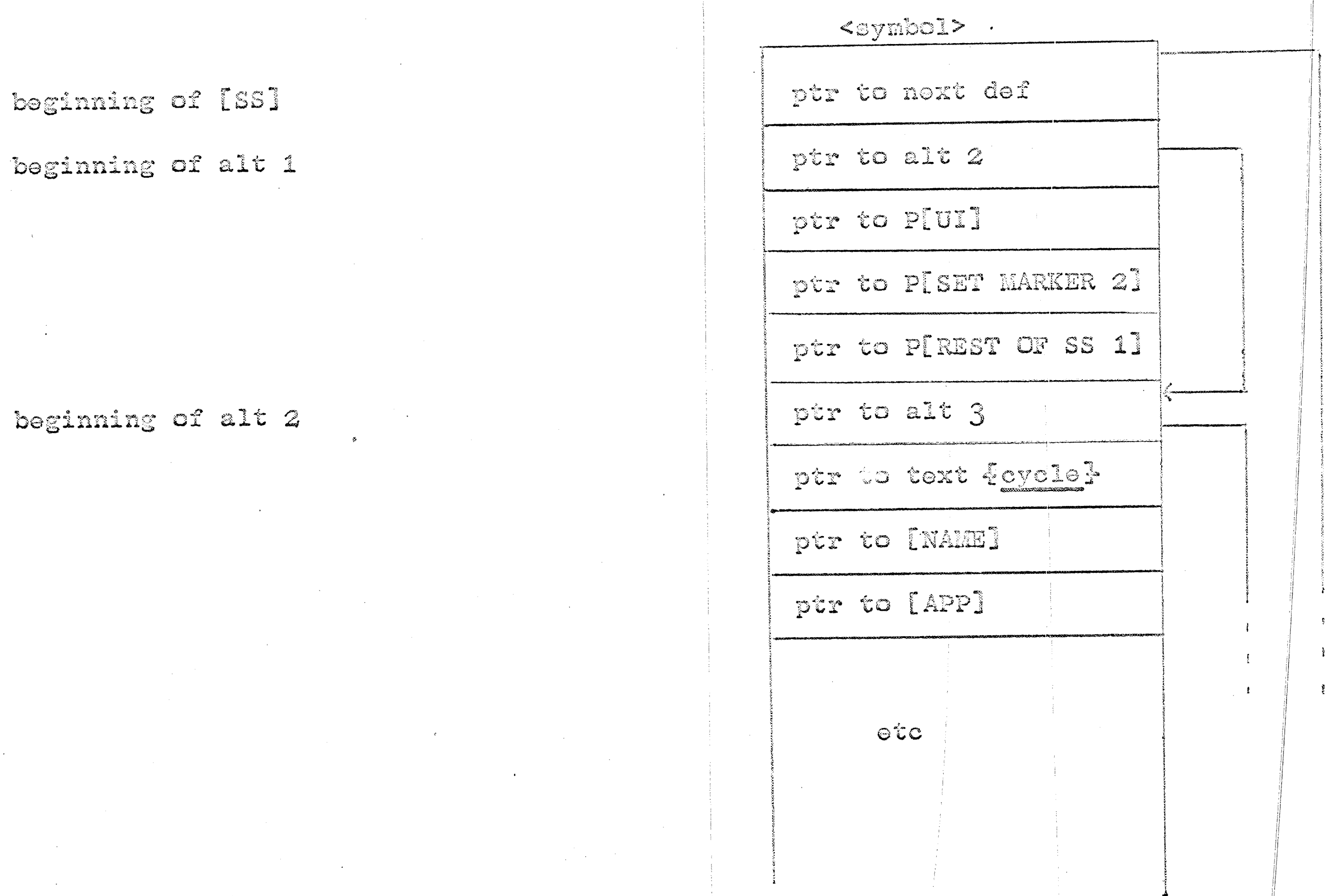
The literal dictionary <clett> is organised so that every entry of a literal begins at a new computer word which contains a pointer to the beginning of the next entry. This enables the



recognition phrase routine <compare> to mechanically test for end-of-literal. The literal dictionary, like the array <symbol> is filled in at compiler-define time described under the section ''Compiling a New Compiler''.

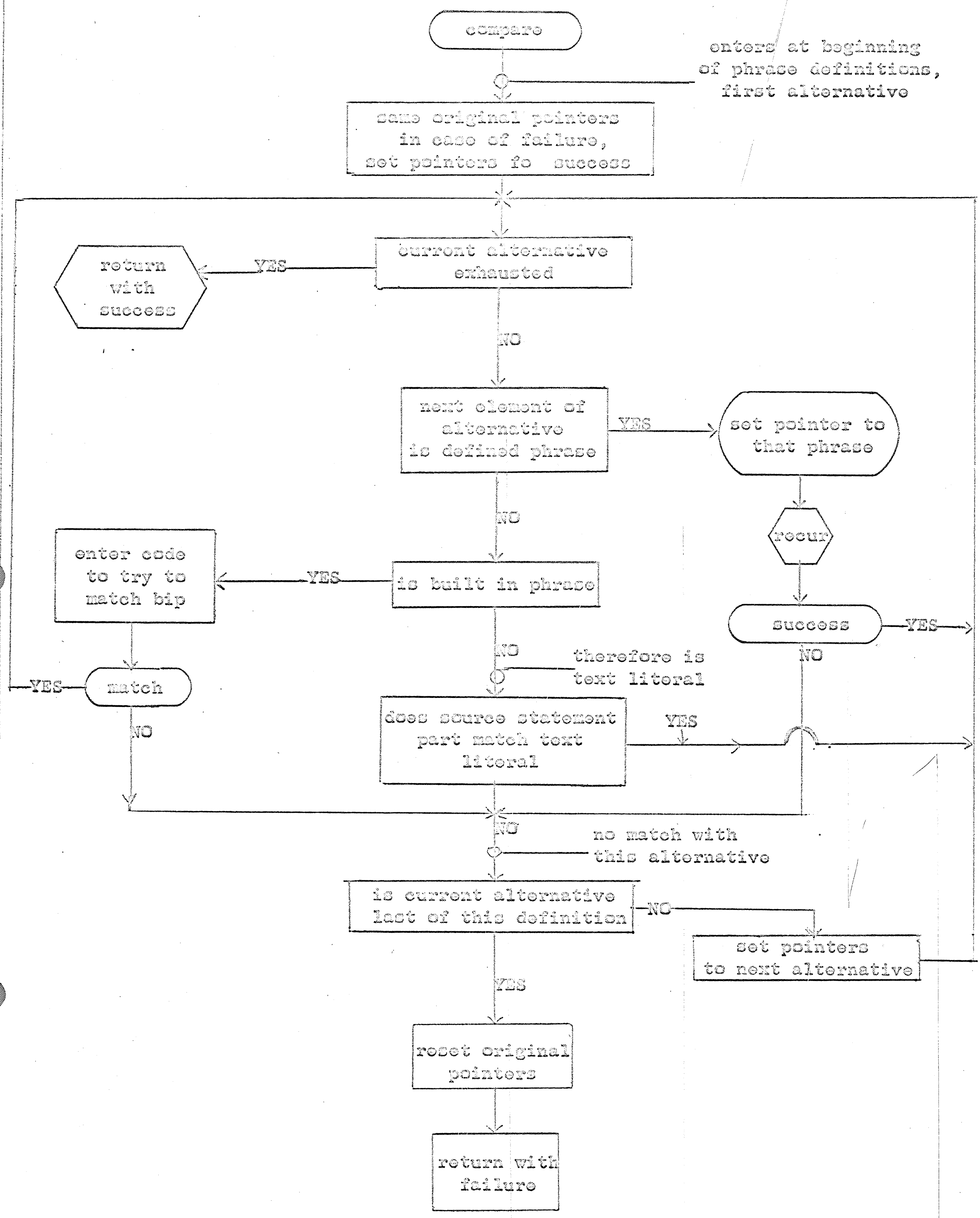
Pointers to built in phrases are used directly as switch indices in <compare> to jump to the coded parts.)

In this fashion, the whole tree of phrase definitions is stored for use by the recognition routine. As a whole, the phrase structure tree (for version I) has thirty ''tips'' and a number of ''roots''. The roots are text literals and the nine built in phrases. The tips are the thirty alternatives of phrase [SS]. Every legal AA source statement corresponds to a path beginning at one of the tips and ending at a prescribed root: ([S] or for labels [:].) Routine compare thus begins scanning from the beginning of the definition of [SS] in <symbol>,



and compares each alternative of [SS] to the source statement until a match is found, or until [SS] is exhausted.

Outline of line recognition routine



The recognition routine is capable of dealing with one phrase definition only. In particular it is capable of processing built in phrases, matching source statement text to text literals, flagging the fact that it has succeeded or failed in matching an alternative and recording the number of the alternative matched.

The method of scanning this involves entering <compare> with <symbol> pointers pointing to the beginning of a phrase definition. <compare> will then try to match the first alternative to the source statement by examining the first entry in the first alternative, trying to match the source statement part if the first entry is a b.i.p. or text literal, or setting pointers for the referenced definition and calling itself if the entry points to a phrase definition.

The object of recognising a source statement is to translate it into a form convenient for compilation. Basically this consists in forming a list of numbers which can be interpreted by the compilation routines as switches to particular routines or as flags. Not surprisingly the compilation routines are organised to reflect the structure of [SS].

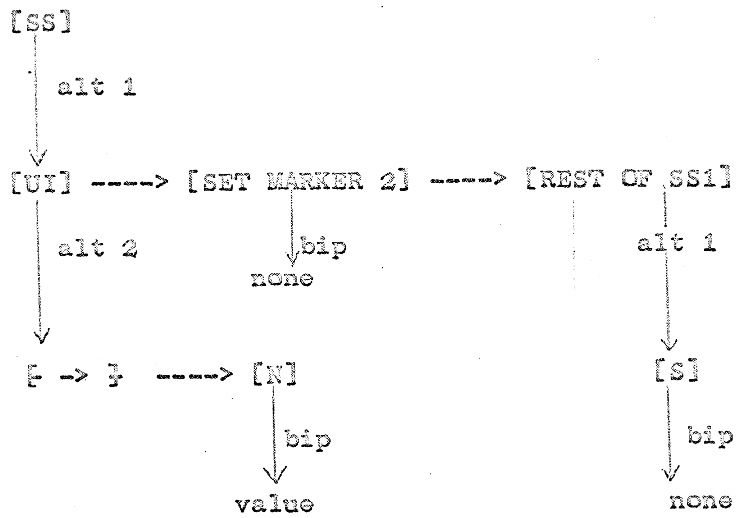
The compilation list or analysis record is written by <compare> as a source statement is matched. The majority of entries consist of alternative numbers of phrase parts matched with source statement parts. Each time compare cannot match an alternative, it goes on to try the next (unless the definition is exhausted, in which case it sets a 'failure' flag and returns.) For details of how the alternative numbers are planted, see the notes accompaning the flowcharts.

Entries in the analysis record other than alternative numbers are deposited for certain built in phrases, as follows;

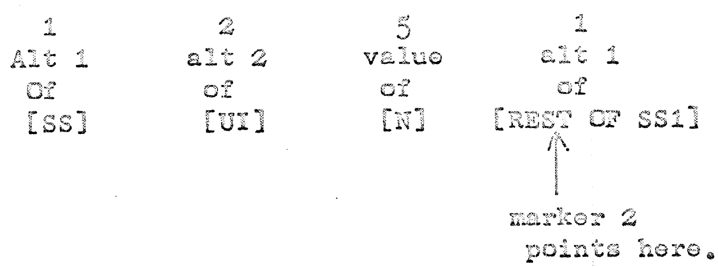
The formation of an analysis record is illustrated for two examples below. The source statement is presented along with its tree structure, then the corresponding analysis record is given.

Source statement: ->5

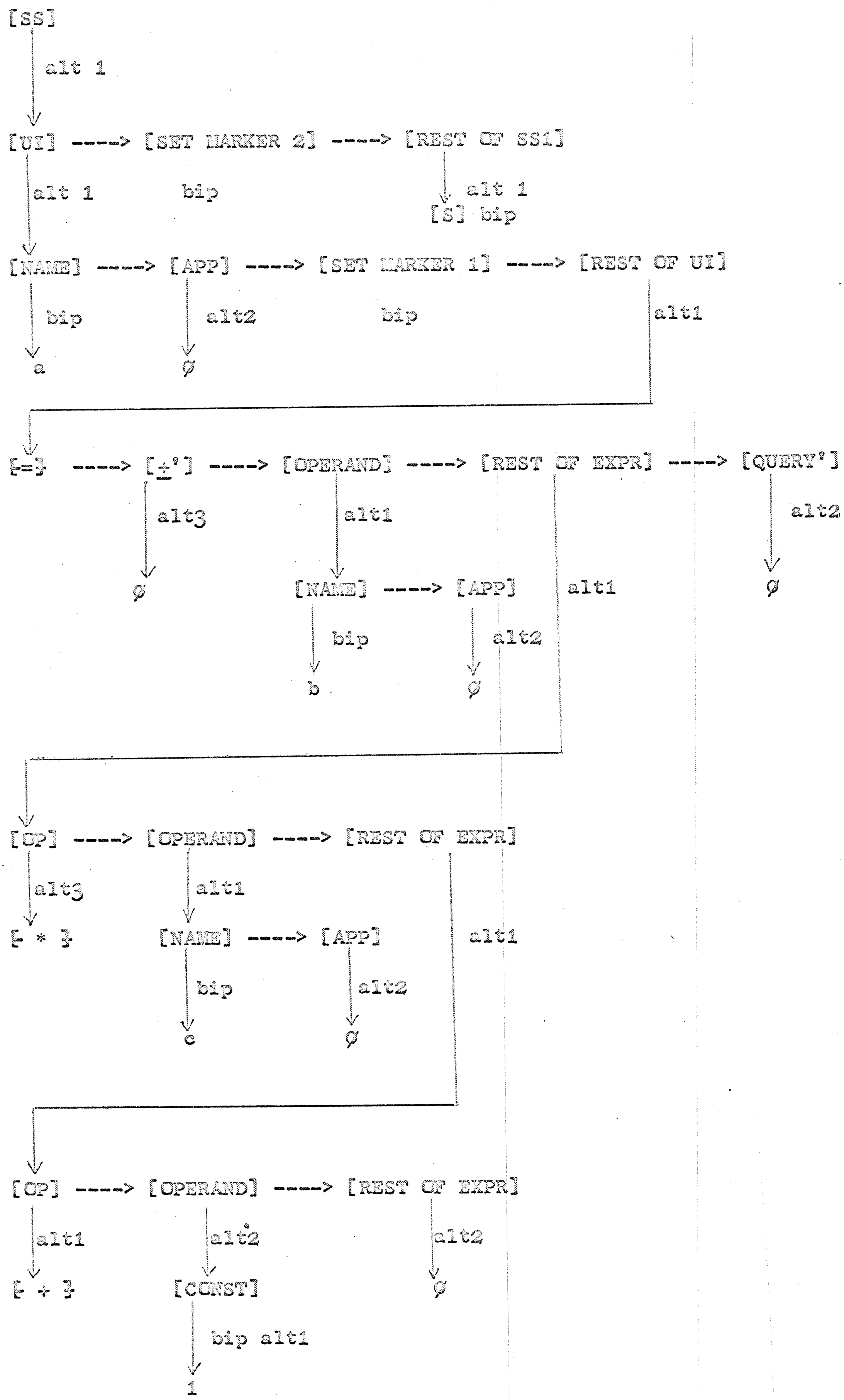
Tree:



Analysis record A

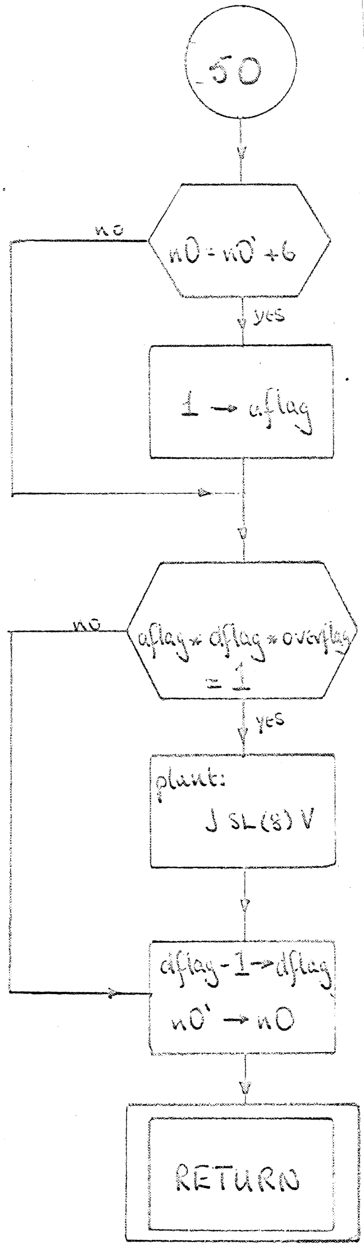


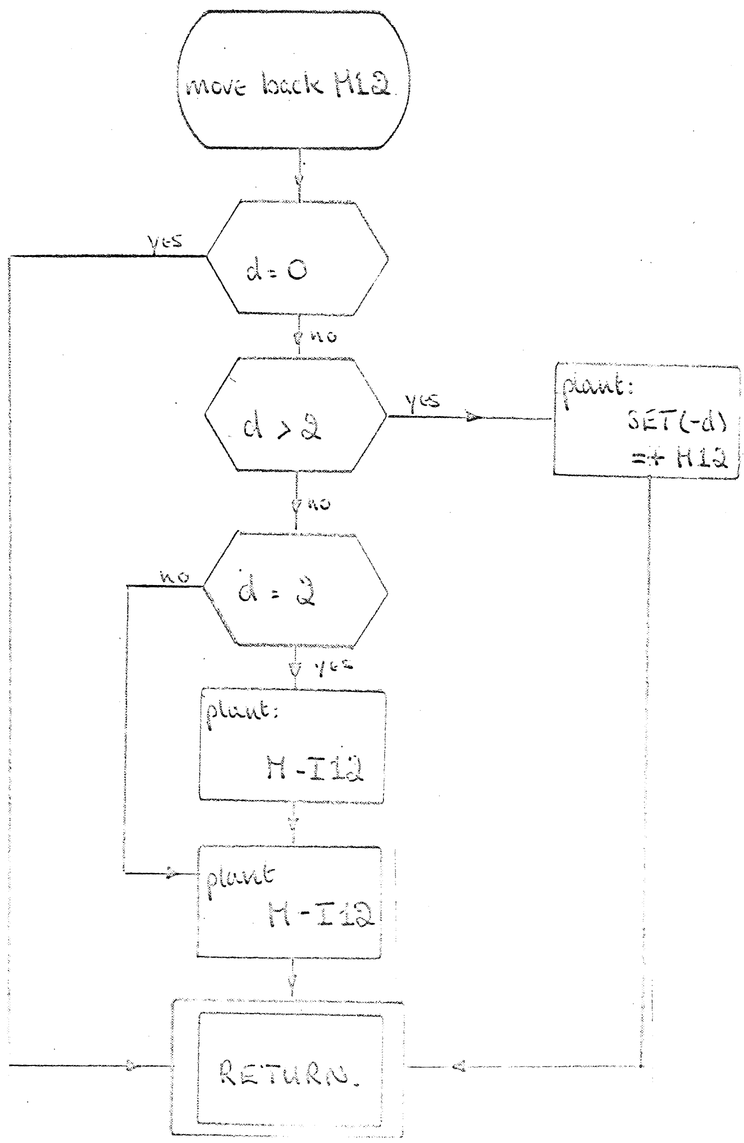
A more complicated example,
Source Statement a=b*c+1
Tree:

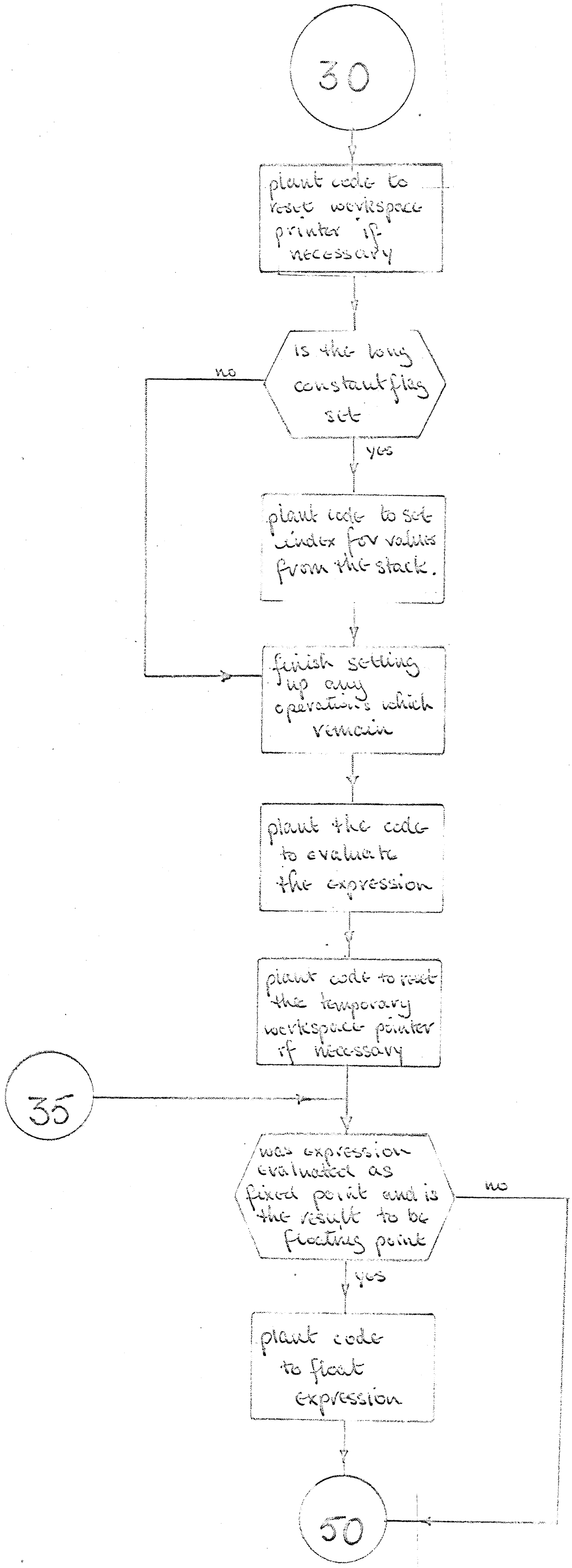


The resulting analysis record would be

```
1,1, a,2,1,3,1,b,2,1,3,1,C,2,1,1,2,1,1,2,2,2
       = †           *          +     1      †
        marker1                             marker 2
```


50
n0 = n0' + 6
no
yes
1 → aflag
no
aflag * dflag * overflag = 1
yes
plant:
J SL(8) V
dflag - 1 → dflag
n0' → n0
RETURN


move back M12
yes
d = 0
no
d > 2
yes
plant:
SET(-d)
=+ M12
no
no
d = 2
yes
plant:
M - I12
plant
M - I12
RETURN.


30
plant code to reset workspace printer if necessary
is the long constant flag set
no
yes
plant code to set index for values from the stack.
finish setting up any operations which remain
plant the code to evaluate the expression
plant code to reset the temporary workspace pointer if necessary
35
was expression evaluated as fixed point and is the result to be floating point
no
yes
plant code to float expression
50

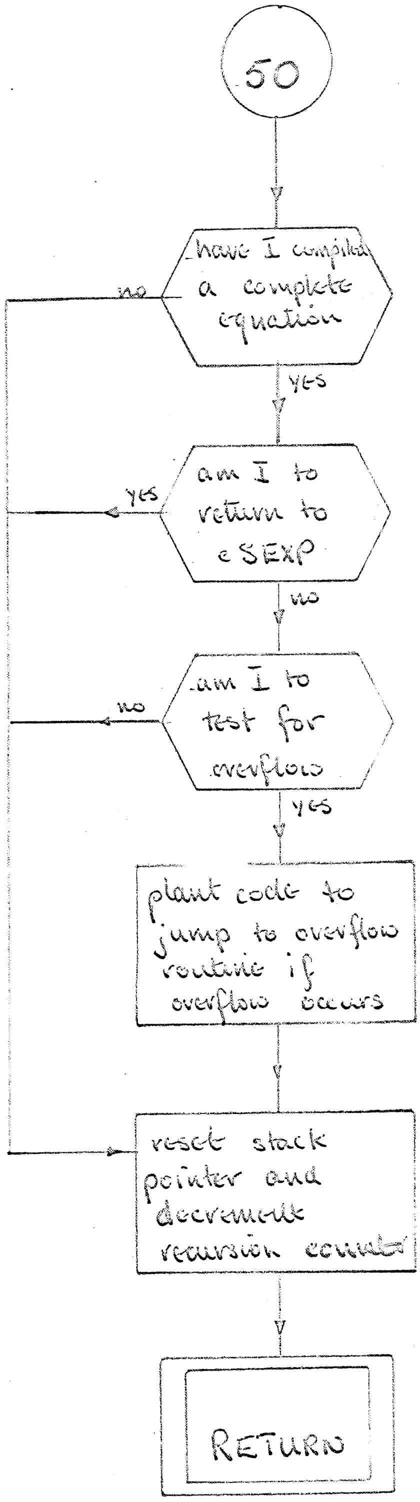
* Special codes are:

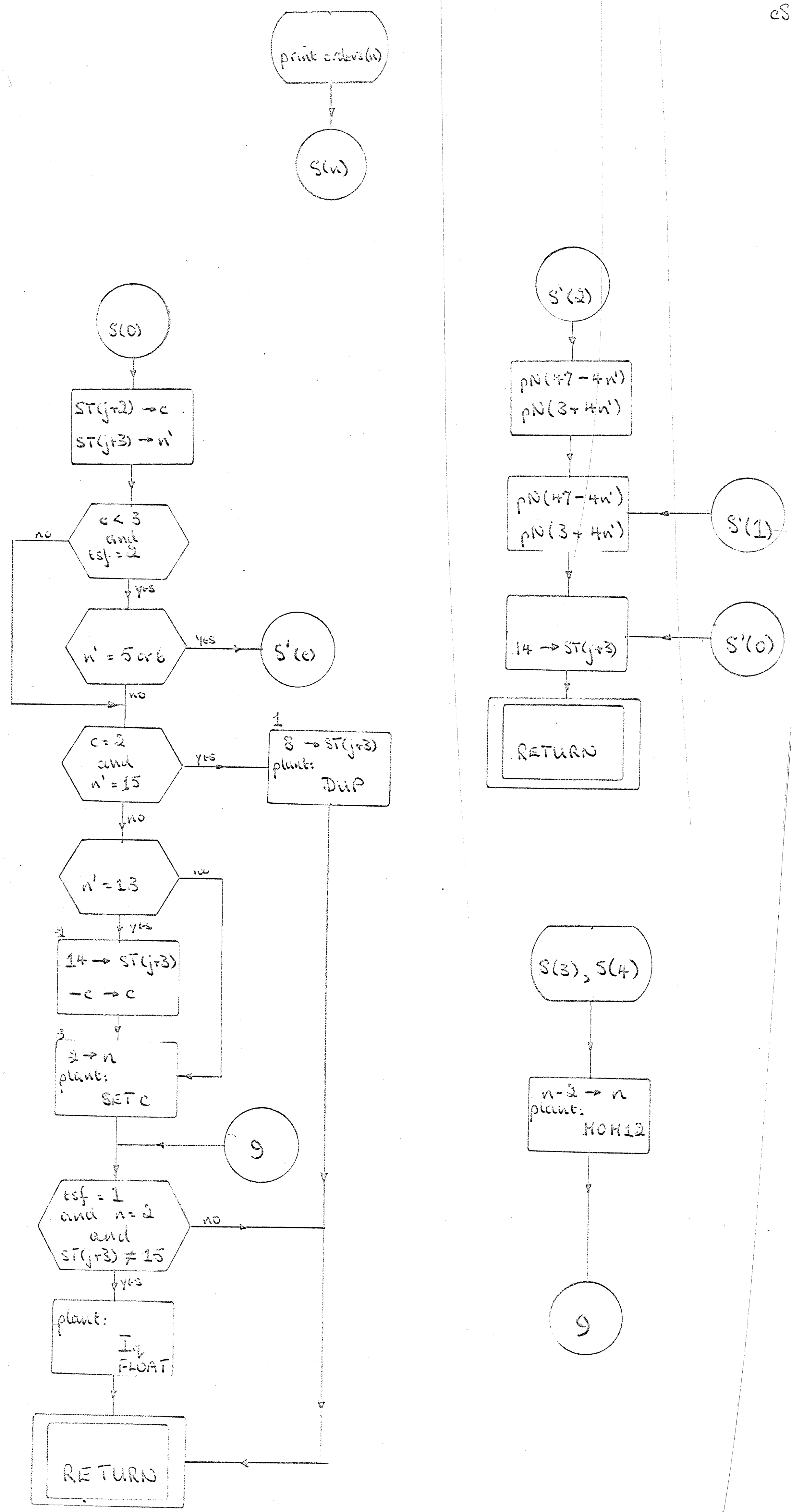
NOT; NEG	to add one to N1
NEG; NOT	to subtract one from N1

This is a bit of optimisation and the reason for it lies in the comparative execution times of the following code sequences

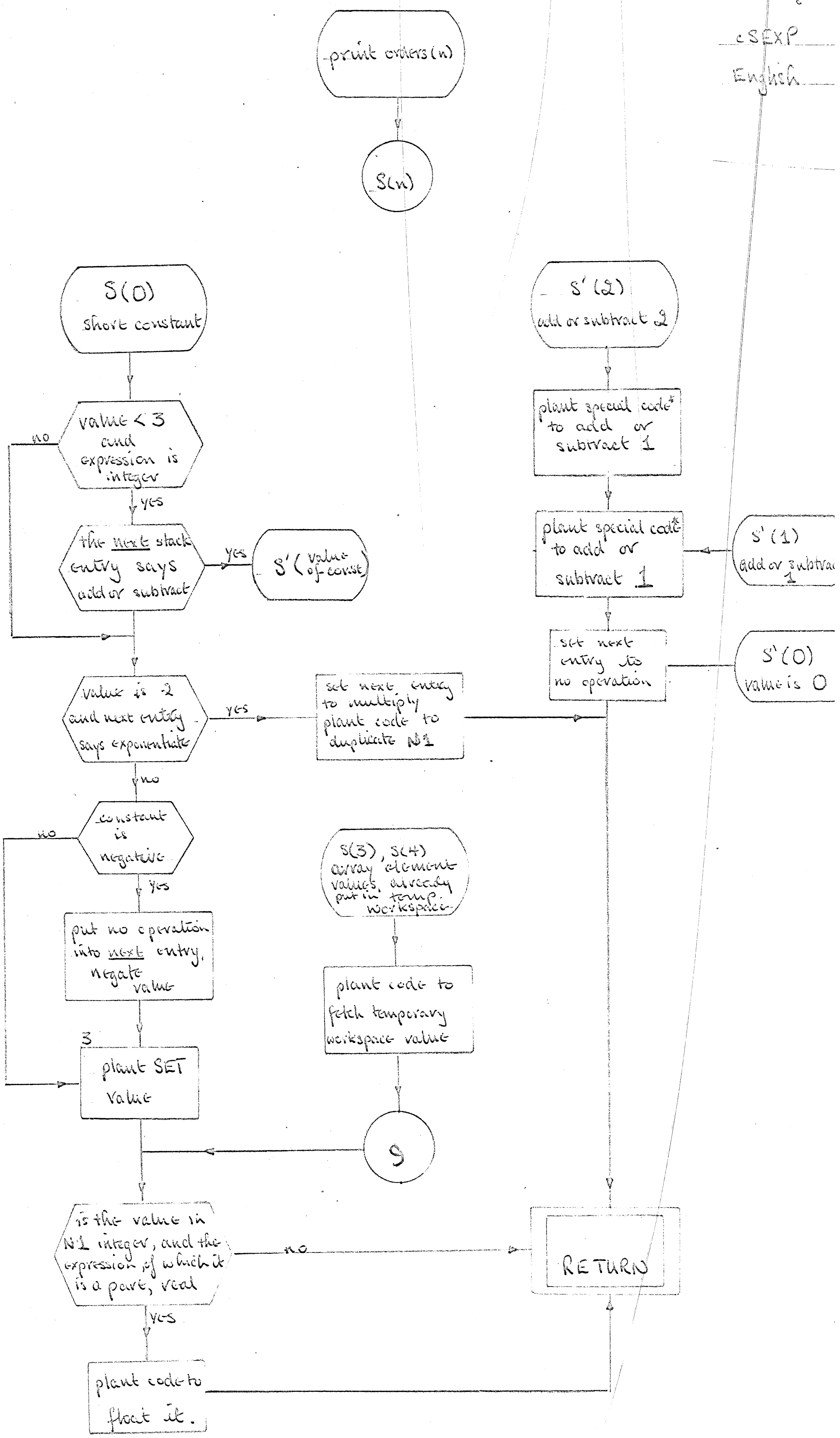
```
SET 1                    NOT
+       = 6 μsec         NEG    = 2 μsec

SET 2                    NOT
+       = 6 μsec         NEG
                         NOT    = 4 μsec
                         NEG
```

and the fact that the statement forms $x = y + 1$ and $x = y \pm 2$ are quite common.

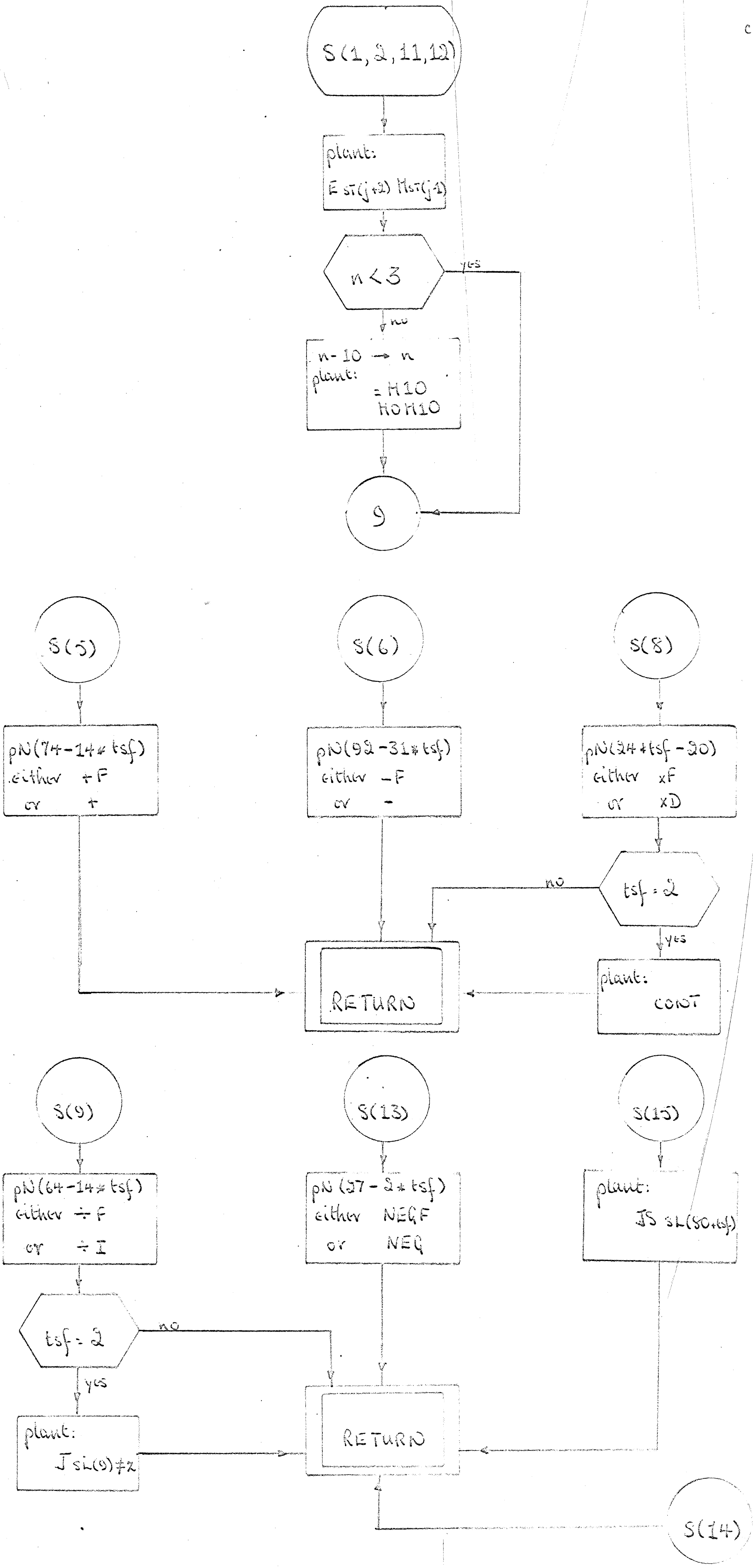

50
have I compiled a complete equation
no
yes
am I to return to eSEXP
yes
no
am I to test for overflow
no
yes
plant code to jump to overflow routine if overflow occurs
reset stack pointer and decrement recursion counter
RETURN


print orders(n)
S(n)
S(0)
ST(j+2) → c
ST(j+3) → n'
c < 3 and tsf = 2
no
yes
n' = 5 or 6
yes
S'(c)
no
c = 2 and n' = 15
yes
1
8 → ST(j+3)
plant:
DUP
no
n' = 13
no
yes
2
14 → ST(j+3)
-c → c
3
2 → n
plant:
SET c
9
tsf = 1 and n = 2 and ST(j+3) ≠ 15
no
yes
plant:
Iq
FLOAT
RETURN
S'(2)
pN(47 - 4n')
pN(3 + 4n')
pN(47 - 4n')
pN(3 + 4n')
S'(1)
14 → ST(j+3)
S'(0)
RETURN
S(3), S(4)
n-2 → n
plant:
MOM12
9

n = ST(j)


print orders (n)
S(n)
S(0)
short constant
value < 3
and
expression is
integer
no
yes
the next stack
entry says
add or subtract
yes
S'(value of const)
value is -2
and next entry
says exponentiate
yes
set next entry
to multiply
plant code to
duplicate N1
no
constant
is
negative
no
yes
put no operation
into next entry,
negate
value
3
plant SET
value
S(3), S(4)
array element
values, already
put in temp.
workspace
plant code to
fetch temporary
workspace value
9
is the value in
N1 integer, and the
expression of which it
is a part, real
no
yes
plant code to
float it.
S'(2)
add or subtract 2
plant special code*
to add or
subtract 1
plant special code*
to add or
subtract 1
S'(1)
add or subtract
1
set next
entry to
no operation
S'(0)
value is 0
RETURN

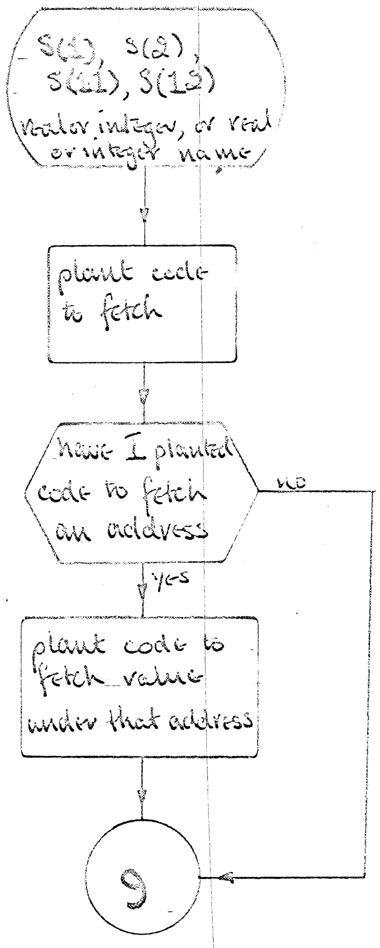
cSEXP

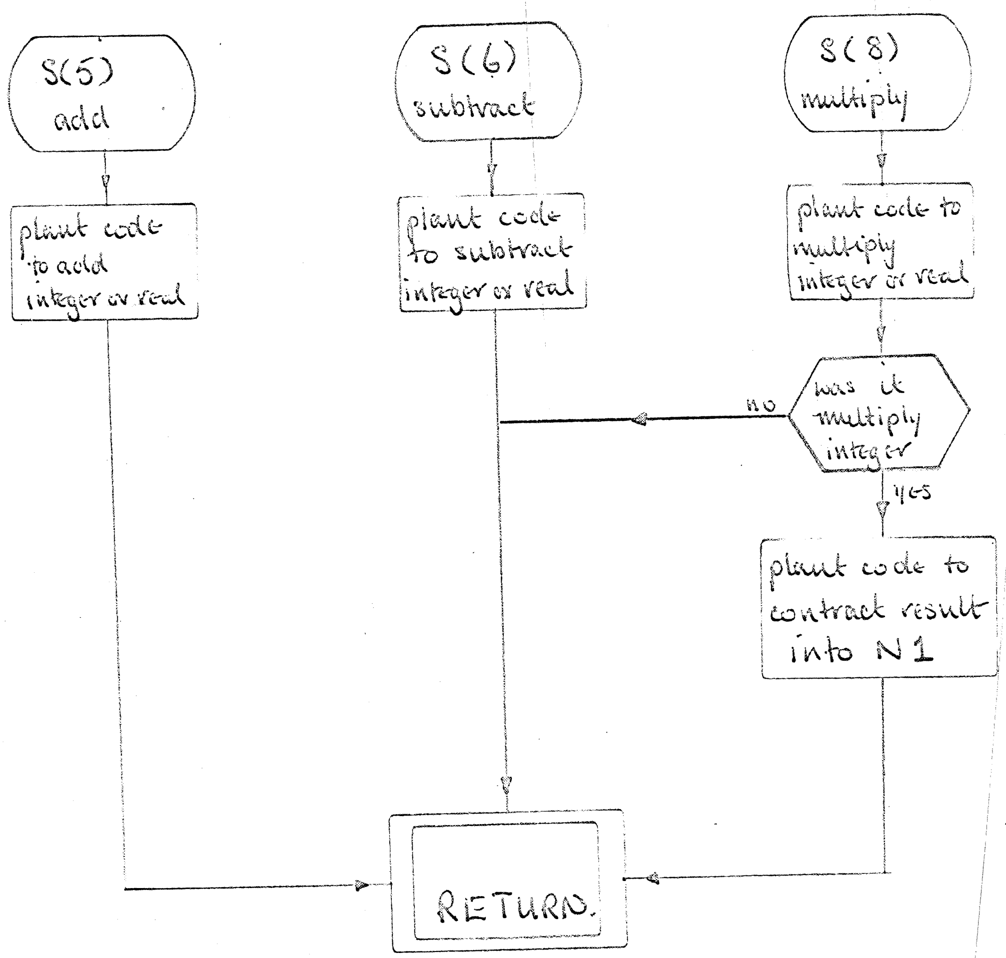


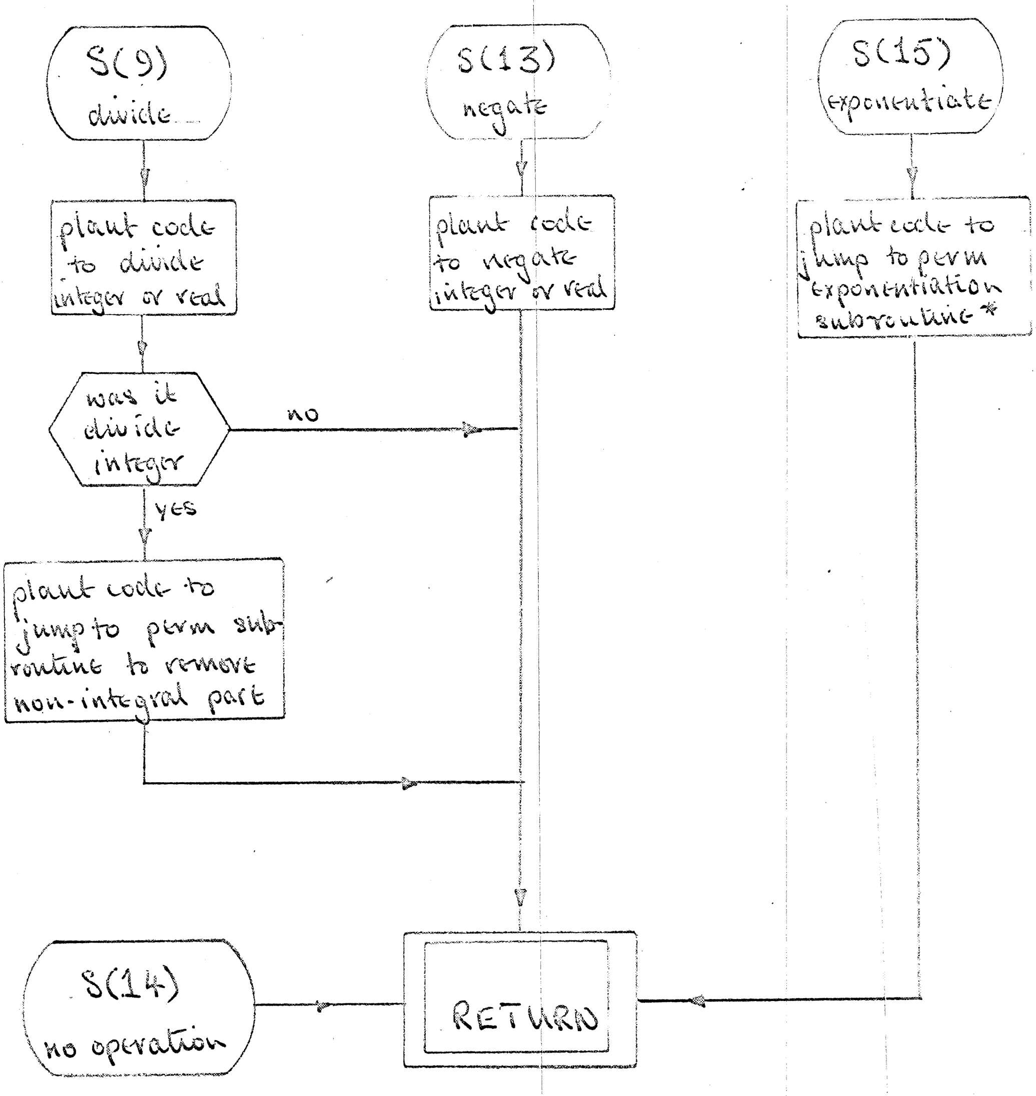
S(1, 2, 11, 12)
plant:
E ST(j+2) MST(j-1)
n < 3
yes
no
n - 10 → n
plant:
= H10
HOH10
9
S(5)
pN(74-14*tsf)
either +F
or +
S(6)
pN(92-31*tsf)
either -F
or -
S(8)
pN(24*tsf-20)
either xF
or xD
tsf = 2
no
yes
plant:
CONT
RETURN
S(9)
pN(64-14*tsf)
either ÷F
or ÷I
tsf = 2
no
yes
plant:
J SL(9)#2
S(13)
pN(27-2*tsf)
either NEGF
or NEG
S(15)
plant:
JS SL(80+tsf)
RETURN
S(14)

cSEXP

English







* Note, for a^b, $N1 = b$, $N2 = a$ has been compiled at this point.

The following pages list the steps in the recognition and compilation of

if (p=q and q > r) then p=r

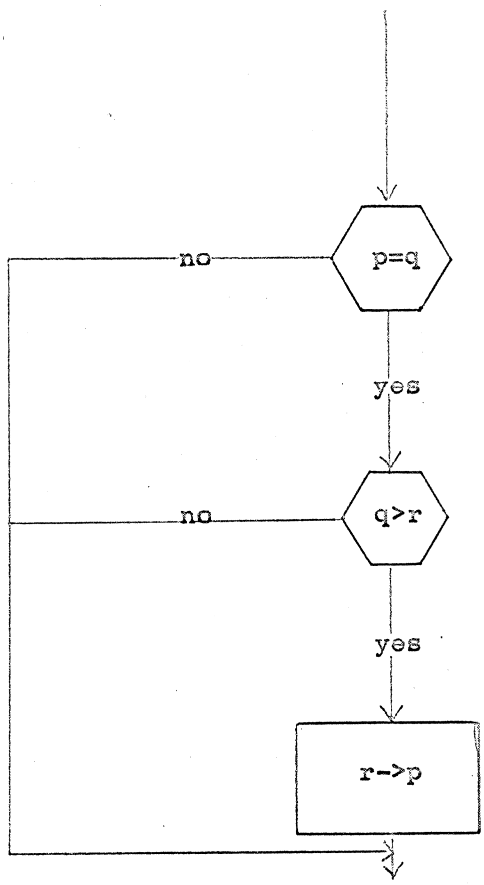
(P[SS]alt=[iu][SC][REST OF COND] then [UI][S].)

The left hand pages show the phrases recognised, in order downwards, with indenting to show the logical structure. On the right hand pages, the routines called in compiling the statement are listed in order downwards with the logical and recursive structure shown by indenting. The analysis record and its pointer <p> are listed in the middle.

The machine code finally planted is as follows

```
SET line no.
=I2                     ;|update line count
p
q
SIGN                    ;|N1=0 if P=Q,=1 if p>q,=-1 if p<q
DUP
J a ≠ Z                 ;| go to a if p≠ q
ERASE
q
r
SIGN                    ;|N1=0 if q=r, =1 if q>r, =-1 if q<r
NEG
NOT                     ;|N1=0 if q>r, ≠0 otherwise
a:  Jb≠Z                ;|go to b if p≠q or q≤r
r
=p
b:  ...
```

Notice that this is extraordinarily efficient code.



The compilation of this example is completely listed below.

Broadly speaking, the four routines do the following.

<cCOND> simply calls <cCC>, and plants overflow checks if required.

<cCC> (''compile compound condition'') deals with

conditional <u>and</u>/<u>or</u> ...

using <cSC> to compile the simple conditional, planting a test,

and calling <cCC> to handle the right conditional part ... .

<cSC> (''compile simple condition'')

deals with X condition y, by compiling x and y

into the nest, and calling <cCOMP> to make the comparison

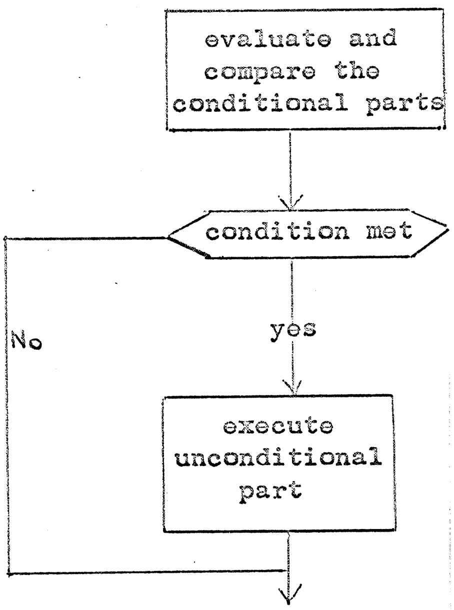
<cCOMP> plants code to compare N1 and N2 and set N1=0 if the condition is met.

All legal conditionals have one of two forms.

if or unless conditional part then execute unconditional part

execute unconditional part if or unless conditional part.

A conditional source statement is compiled by five routines controled by <cSS.> <cUI> compiles the unconditional part, and <cCOND,cCC,cSC,cCOMP> compile the conditional part. <cSS> first moves the analysis pointer to point at the conditional part of the statement, and calls <cCOND>. Upon return, <cSS> moves the pointer to the unconditional part and calls <cUI>. The resulting code will accomplish



This flow diagram is deceptive, though, because the code is heavily ''optimised'' to test whether the conditional parts are met every time failing such a test would mean skipping the unconditional part. For example,

if (p=q and q>r) then p=r

compiles as

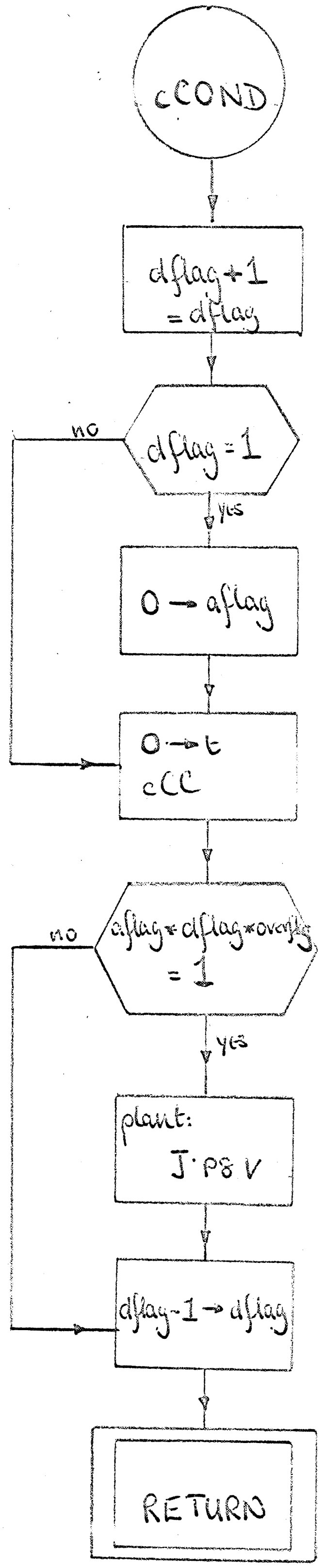
PHRASE PART	ALTERNATIVE
[SS]	5
[iu]	if
[SC]	PARENTHETIC
[(]	
[SC]	NON-PARENTHETIC
[±']	Ø
[OPERAND]	NAME
[NAME]	P
[APP]	Ø
[REST OF EXPR]	Ø
[COMP]	=
[±']	Ø
[OPERAND]	NAME
[NAME]	q
[APP]	Ø
[REST OF EXPR]	Ø
[REST OF SC]	Ø
[REST OF COND]	and
[and]	
[SC]	NON-PARENTHETIC
[±']	Ø
[OPERAND]	NAME
[NAME]	q
[APP]	Ø
[REST OF EXPR]	Ø

PHRASE PART	ALTERNATIVE
[ss]	5
[iu]	if
[sc]	PARENTHETIC
[(]	
[sc]	NON-PARENTHETIC
[±']	∅
[OPERAND]	NAME
[NAME]	P
[APP]	∅
[REST OF EXPR]	∅
[COMP]	=
[±']	∅
[OPERAND]	NAME
[NAME]	q
[APP]	∅
[REST OF EXPR]	∅
[REST OF SC]	∅
[REST OF COND]	and
[and]	
[sc]	NON-PARENTHETIC
[±']	∅
[OPERAND]	NAME
[NAME]	q
[APP]	∅
[REST OF EXPR]	∅

A(p)	P	ROUTINE EXAMING A(.)	CODE COMPILED
5	1	cSS	SET LINE
1	2		=I2
2	3	cCOND	
		cCC	
		cSC	
1	4	cCOND	
		cCC	
		cSC	
3	5	cSEXP(3)	fetch p
1	6		
p	7		
2	8		
2	9		
1	10	cCOMP(1)	
3	11	cSEXP(3)	fetch q
1	12		
q	13		
2	14		
2	15		
2	16	(cCOMP)	SIGN
1	17	(cSC)	
		(cCC)	DUP
1	18		ja≠Z
			ERASE
		cCC	
		cSC	
3	19	cSEXP(3)	fetch q
1	20		
q	21		
2	22		
2	23		

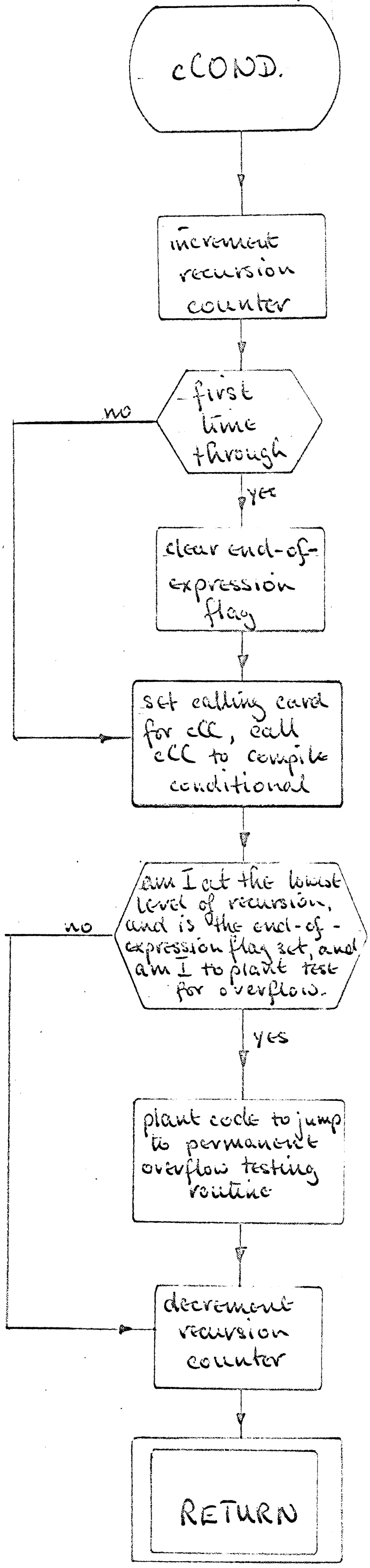
PHRASE PART							ALTERNATIVE
					[COMP]		>
					[±']		∅
					[OPERAND]		NAME
						[NAME]	r
						[APP]	∅
					[REST OF EXPR]		∅
					[REST OF SC]		∅
				[REST OF AND-C]			∅
	[)]						
[REST OF COND]							∅
[<u>Then</u>]							
[UI]							1
	[NAME]						P
	[APP]						∅
	[SET MARKER 1]						
	[REST OF UI]						not ∅
		[=]					
		[±']					∅
		[OPERAND]					NAME
			[NAME]				÷
			[APP]				∅
		[REST OF EXPR]					∅
		[QUERY']					∅
[S]							

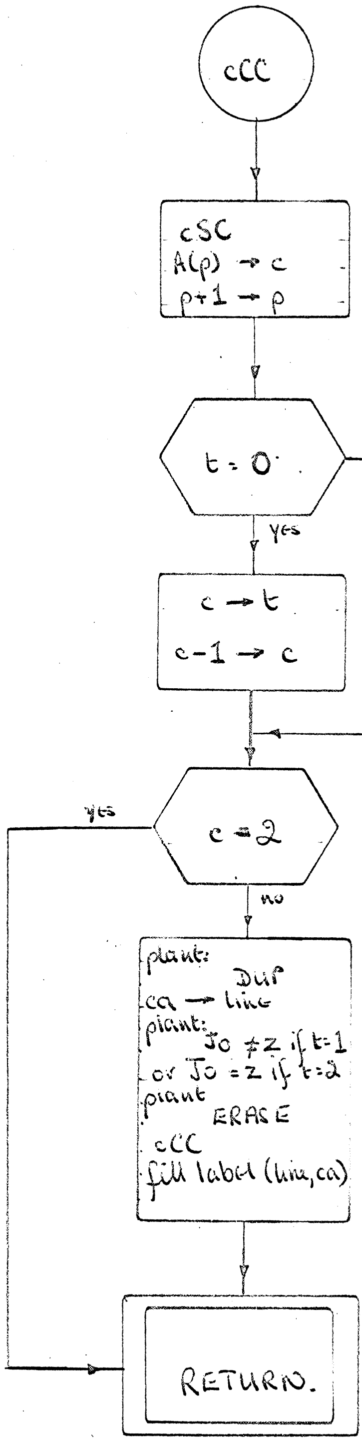
A(p)	P	ROUTINE EXAMINING A(.)	CODE COMPILED
3	24	cCOMP(1)	
3	25	cSEXP(3)	fetch r
1	26		
r	27		
2	28		
2	29	(cCOMP)	SIGN
			NEG
			NOT
2	30	(cSC)	
		(cCC)	
2	31	(cCC)	a:
		(cCOND)	
		(cCC)	
3	32	(cCOND)	
1	33	(cSS)	Jb ≠ Z
	34	cUI	
	35	{ cSEXP }	fetch r
		{ cNAME }	=p
1	36		
3	37		
1	38		
r	39		
2	40		
2	41		
2	42		
		(cSS)	
			(b:)


cCOND
dflag+1 = dflag
dflag = 1
no
yes
0 → aflag
0 → t
cCC
no
aflag * dflag * overflg = 1
yes
plant:
J' P8 V
dflag-1 → dflag
RETURN

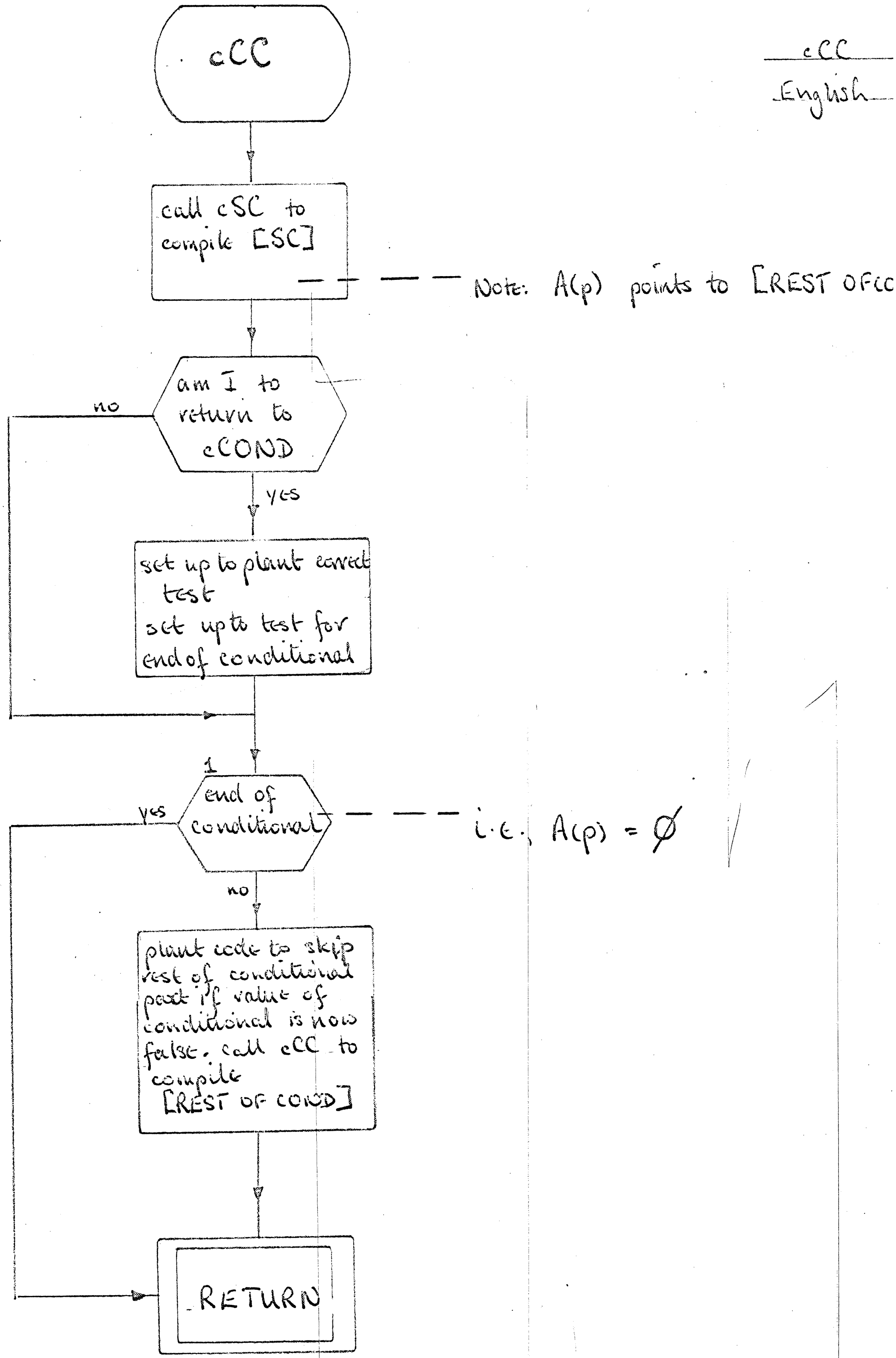
cCOND

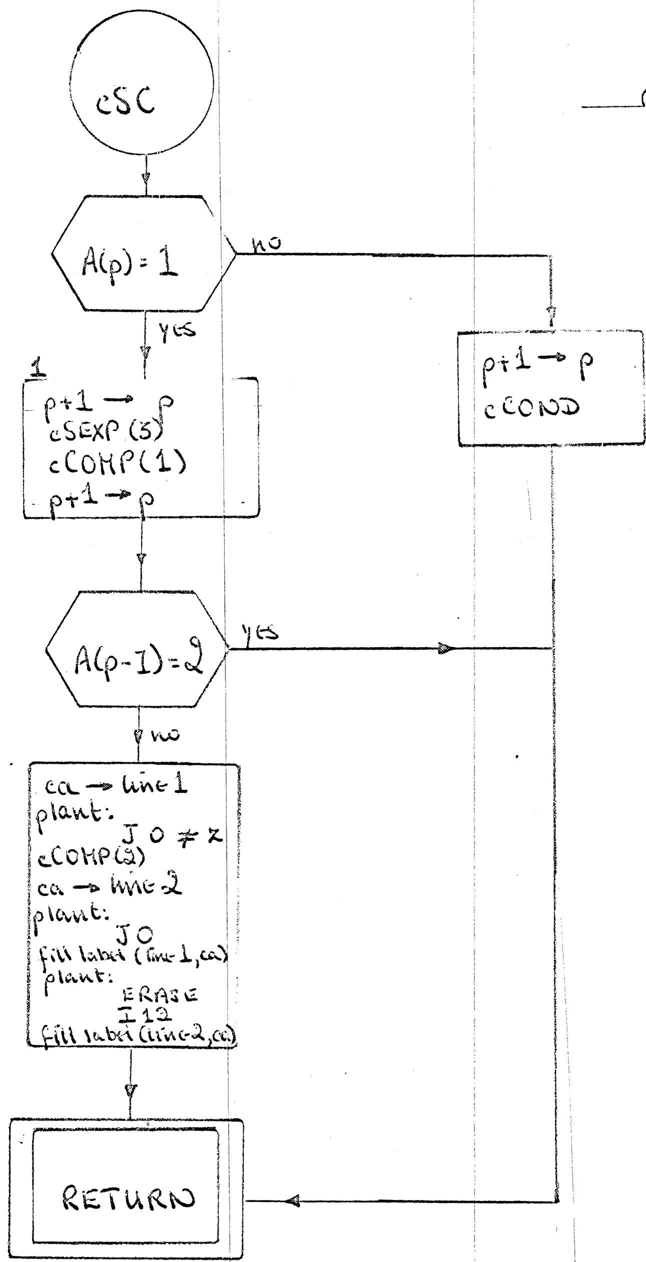
English



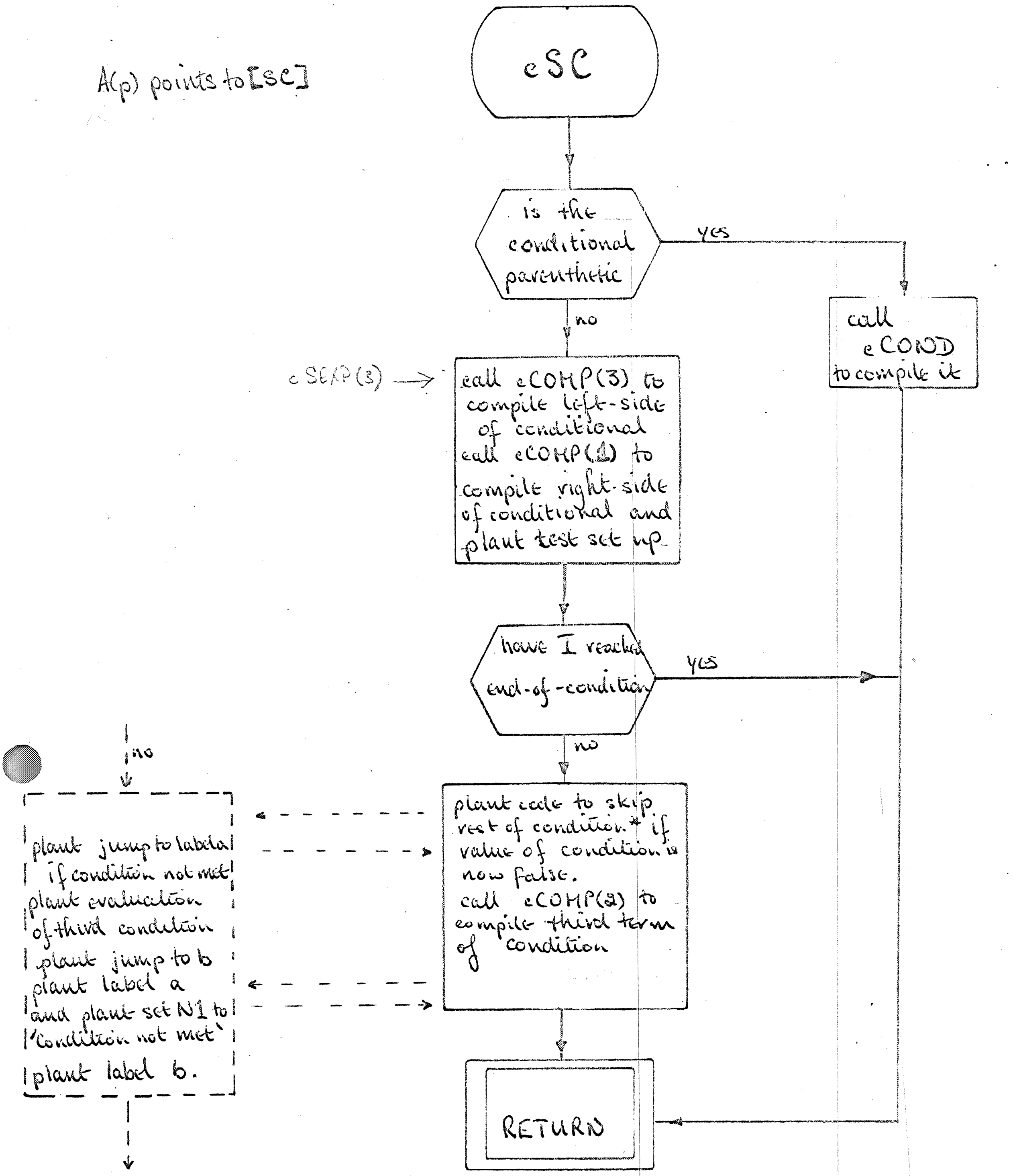

cCC
cSC
A(p) → c
p+1 → p
t = 0
no
yes
c → t
c−1 → c
c = 2
yes
no
plant:
DUP
ca → line
plant:
Jo ≠ z if t=1
or Jo = z if t=2
plant
ERASE
cCC
fill label (line, ca)
RETURN.

cCC
English


cCC
call cSC to compile [SC]
Note: A(p) points to [REST OF CC
am I to return to cCOND
no
yes
set up to plant correct test
set up to test for end of conditional
1
end of conditional
i.e. A(p) = ∅
yes
no
plant code to skip rest of conditional part if value of conditional is now false. call cCC to compile [REST OF COND]
RETURN

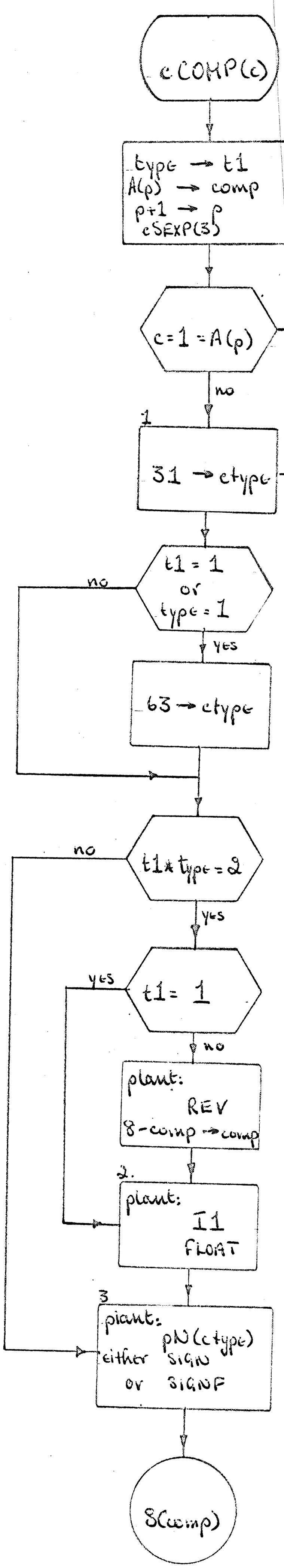

cSC
A(p) = 1
no
yes
1
p+1 → p
cSEXP (3)
cCOMP(1)
p+1 → p
p+1 → p
cCOND
A(p-1) = 2
yes
no
ca → line 1
plant:
J O ≠ z
cCOMP(2)
ca → line 2
plant:
J O
fill label (line-1, ca)
plant:
ERASE
I 12
fill label (line-2, ca)
RETURN

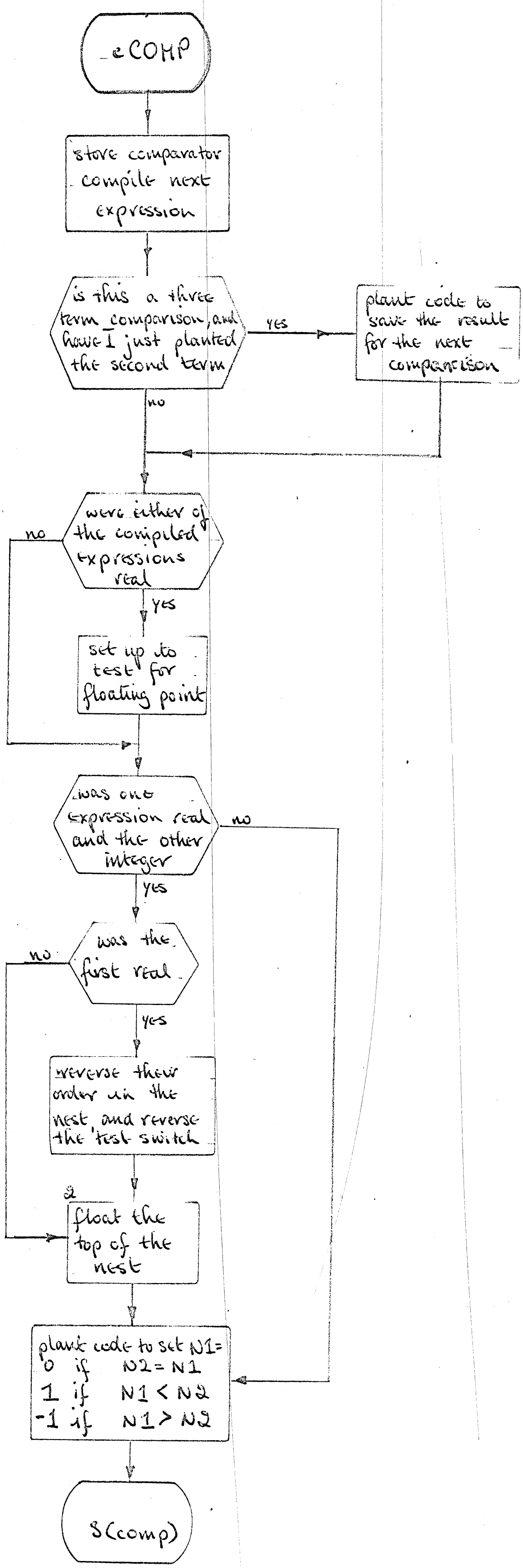
A(p) points to [SC]



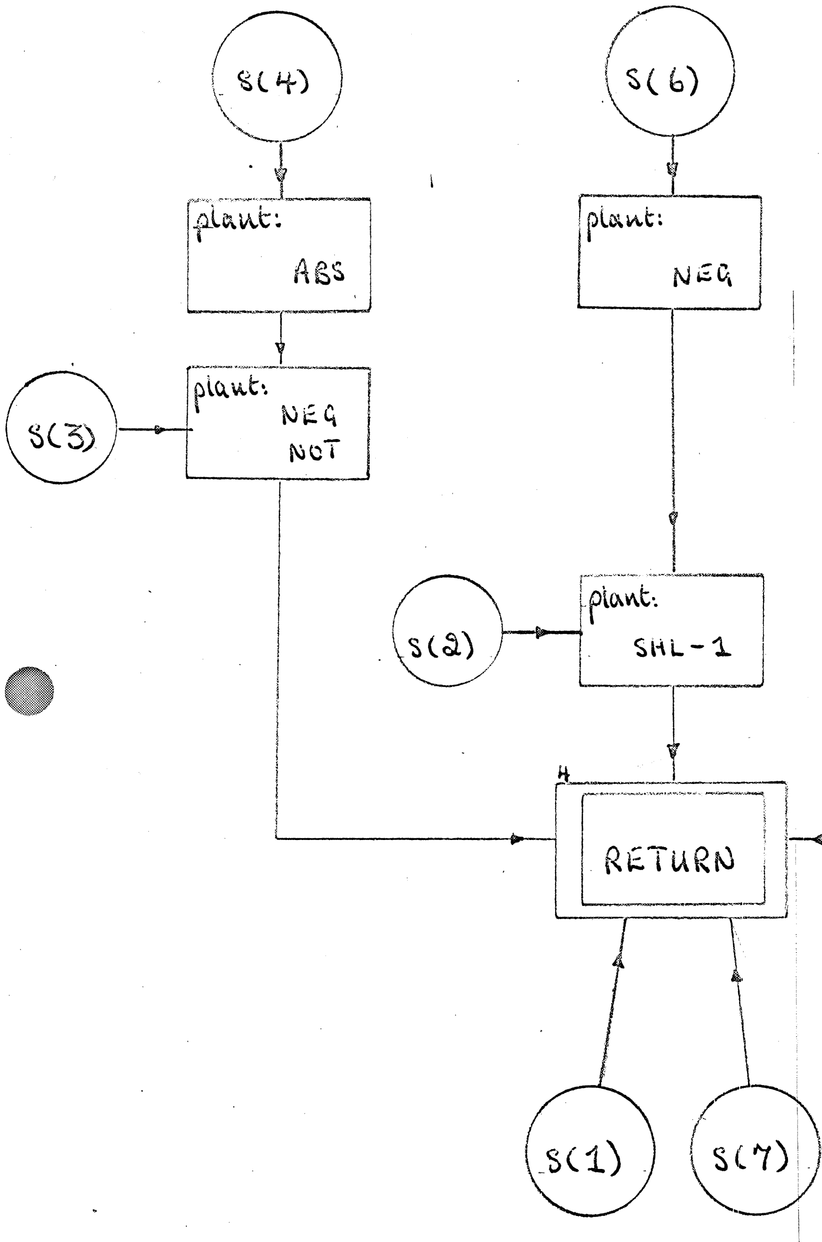
* This is to handle three termed conditionals (e.g. x > y > z)

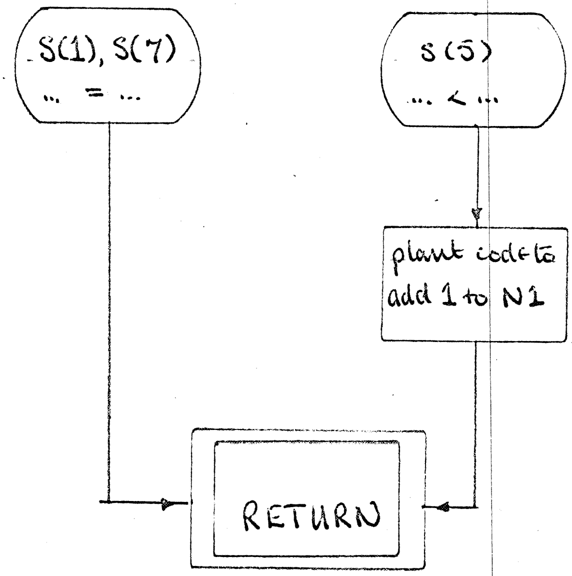
c = 1 on first call
= 2 on second call.

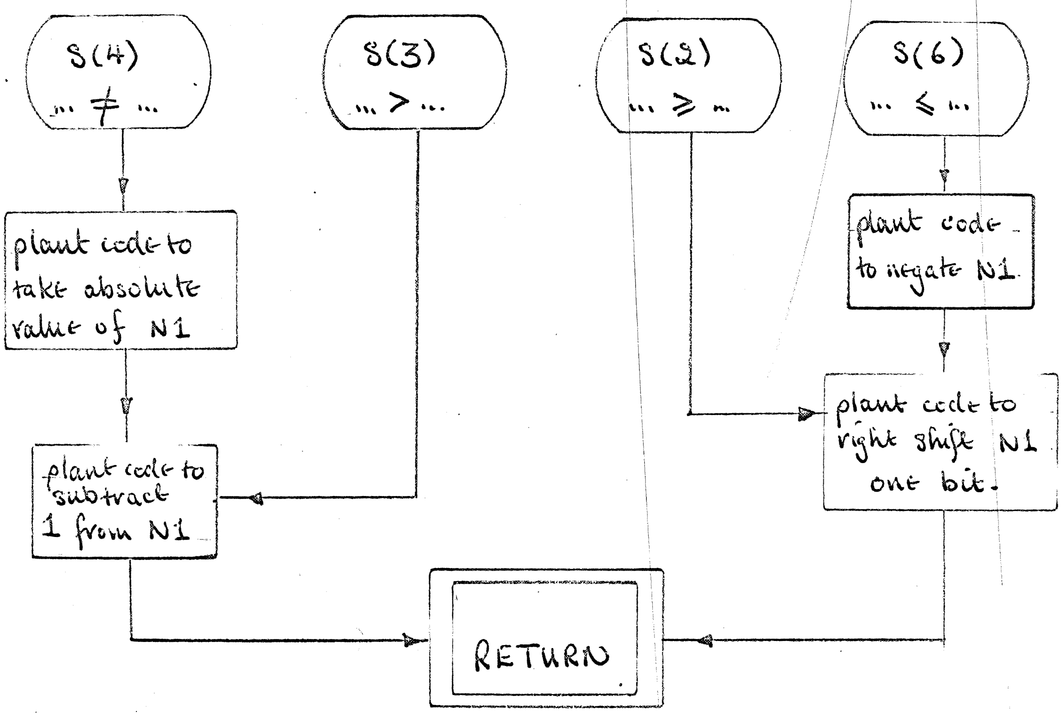



_cCOMP
store comparator compile next expression
is this a three term_comparison, and have I just planted the second term
yes
plant code to save the result for the next comparison
no
were either of the compiled expressions real
no
yes
set up to test for floating point
was one expression real and the other integer
no
yes
was the first real
no
yes
reverse their order in the nest and reverse the test switch
2
float the top of the nest
plant code to set N1=
0 if N2 = N1
1 if N1 < N2
-1 if N1 > N2
S(comp)

cCOMP.


S(4)
S(6)
S(5)
plant: ABS
plant: NEG
plant: NOT NEG
S(3)
plant: NEG NOT
S(2)
plant: SHL-1
4
RETURN
S(1)
S(7)





These codes set N1 = 0 if condition met, ≠ 0 if condition not met.