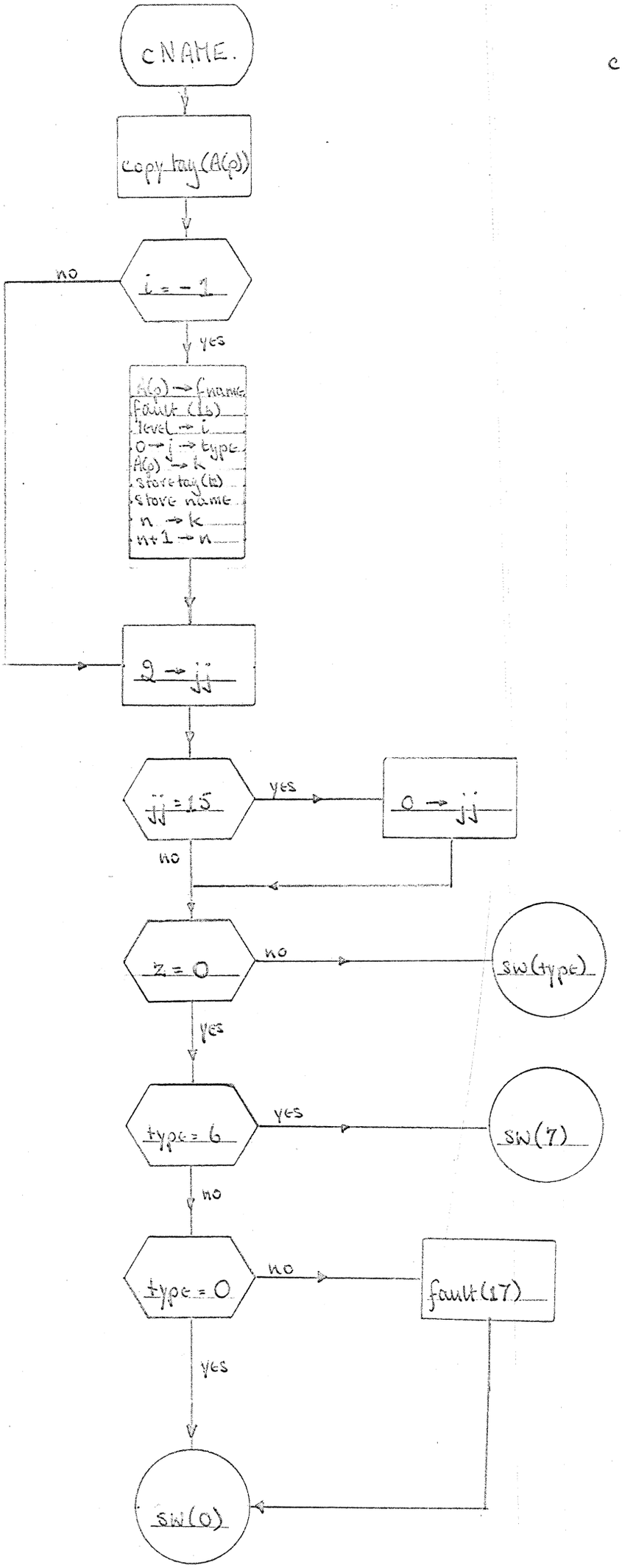
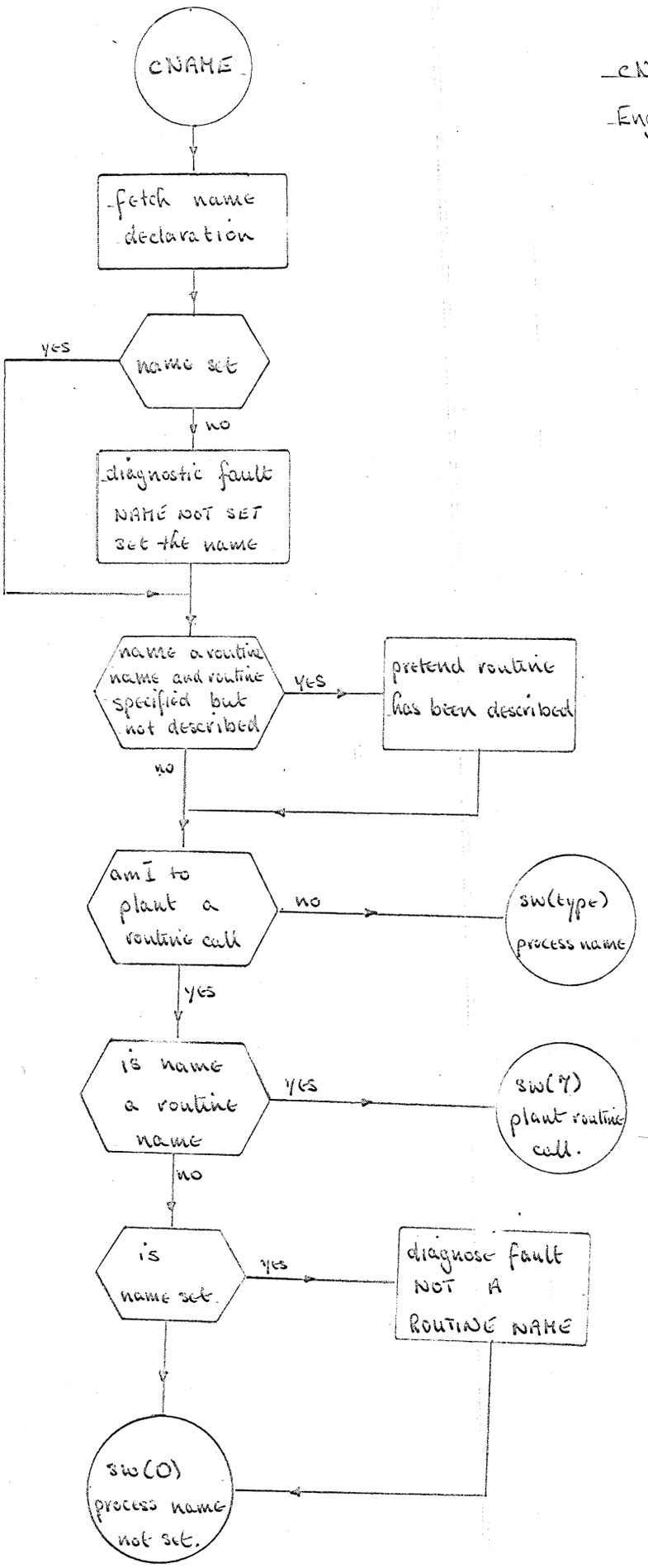
Meaning of calling parameter z

0	compile routine call
1	store into call whose name is referenced by A(p)
2	fetch from call whose name is referenced by A(p)
3	fetch address of call whose name is referenced by A(p)

Meaning of constants

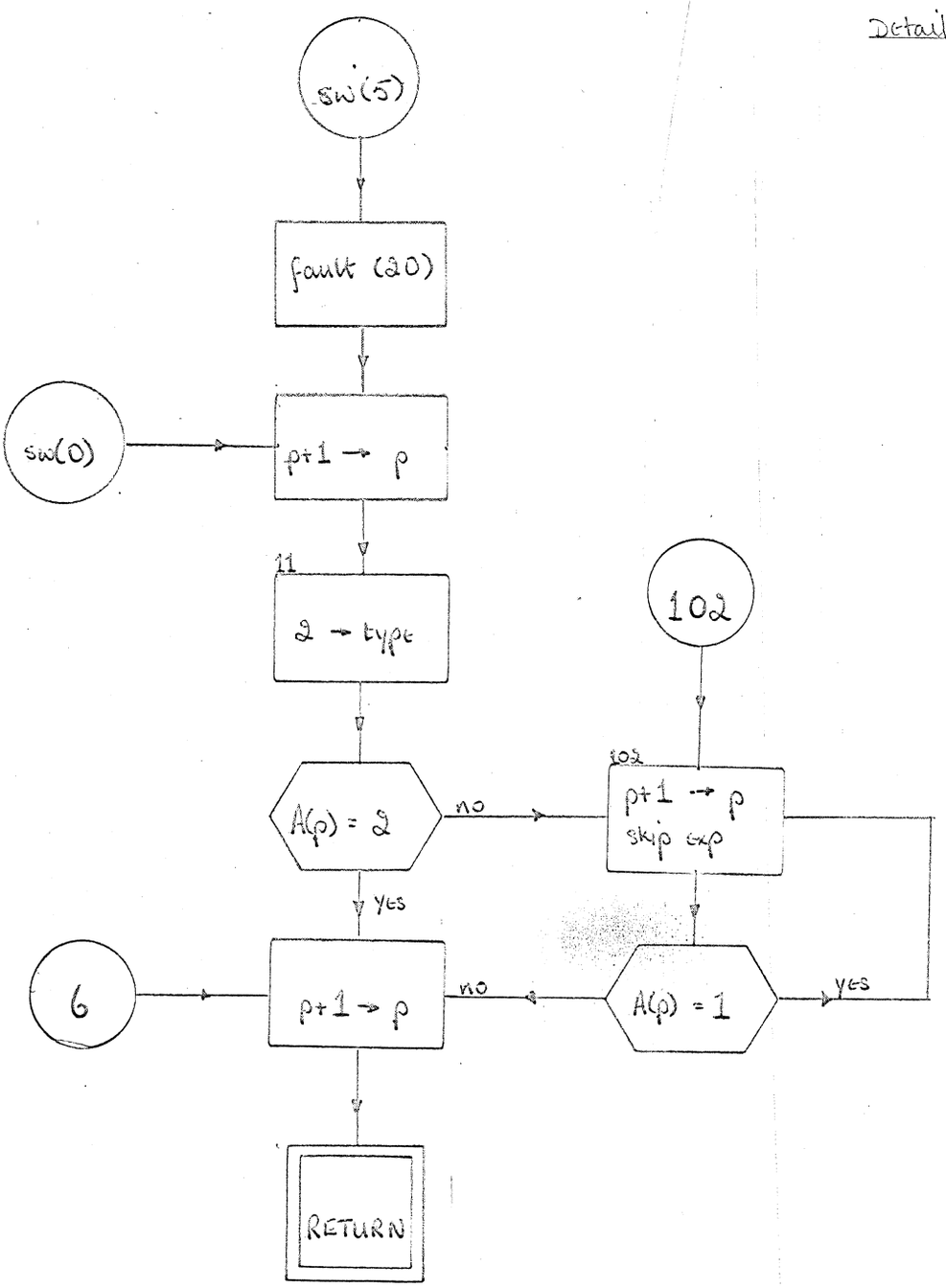
jj	holds the dimension of the name
q	number of parameters declared in array name declarations (in ②)


cNAME.
copy tag(A(p))
i = -1
no
yes
A(p) → fname
fault (fb)
level → i
0 → j → type
A(p) → k
store tag(k)
store name
n → k
n+1 → n
2 → jj
jj = 15
yes
0 → jj
no
z = 0
no
sw(type)
yes
type = 6
yes
sw(7)
no
type = 0
no
fault(17)
yes
sw(0)


cNAME
fetch name declaration
name set
yes
no
diagnostic fault
NAME NOT SET
set the name
name a routine name and routine specified but not described
yes
pretend routine has been described
no
am I to plant a routine call
no
sw(type) process name
yes
is name a routine name
yes
sw(7) plant routine call.
no
is name set
yes
diagnose fault NOT A ROUTINE NAME
sw(0) process name not set.

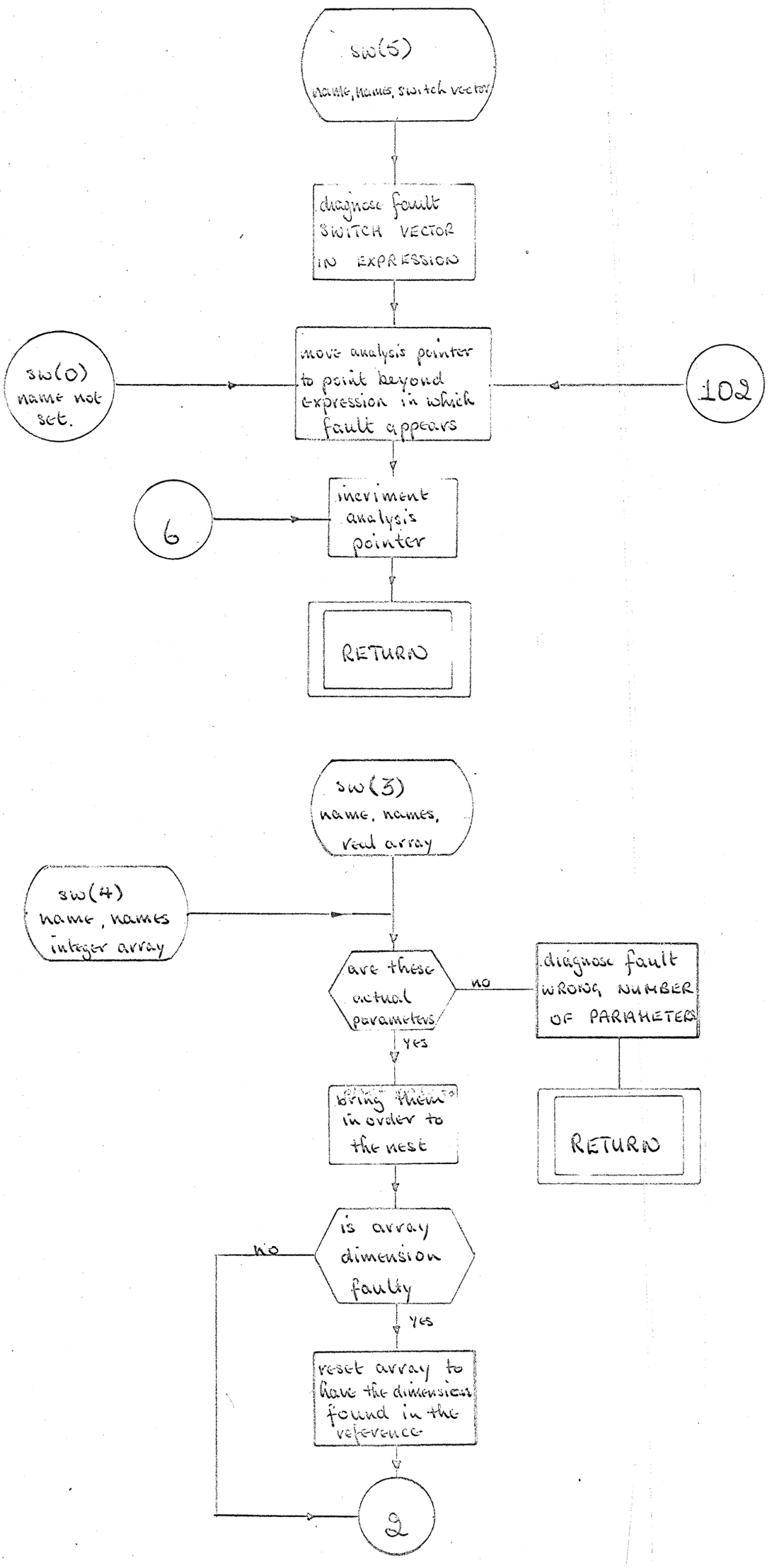
cNAME

Detail flow.

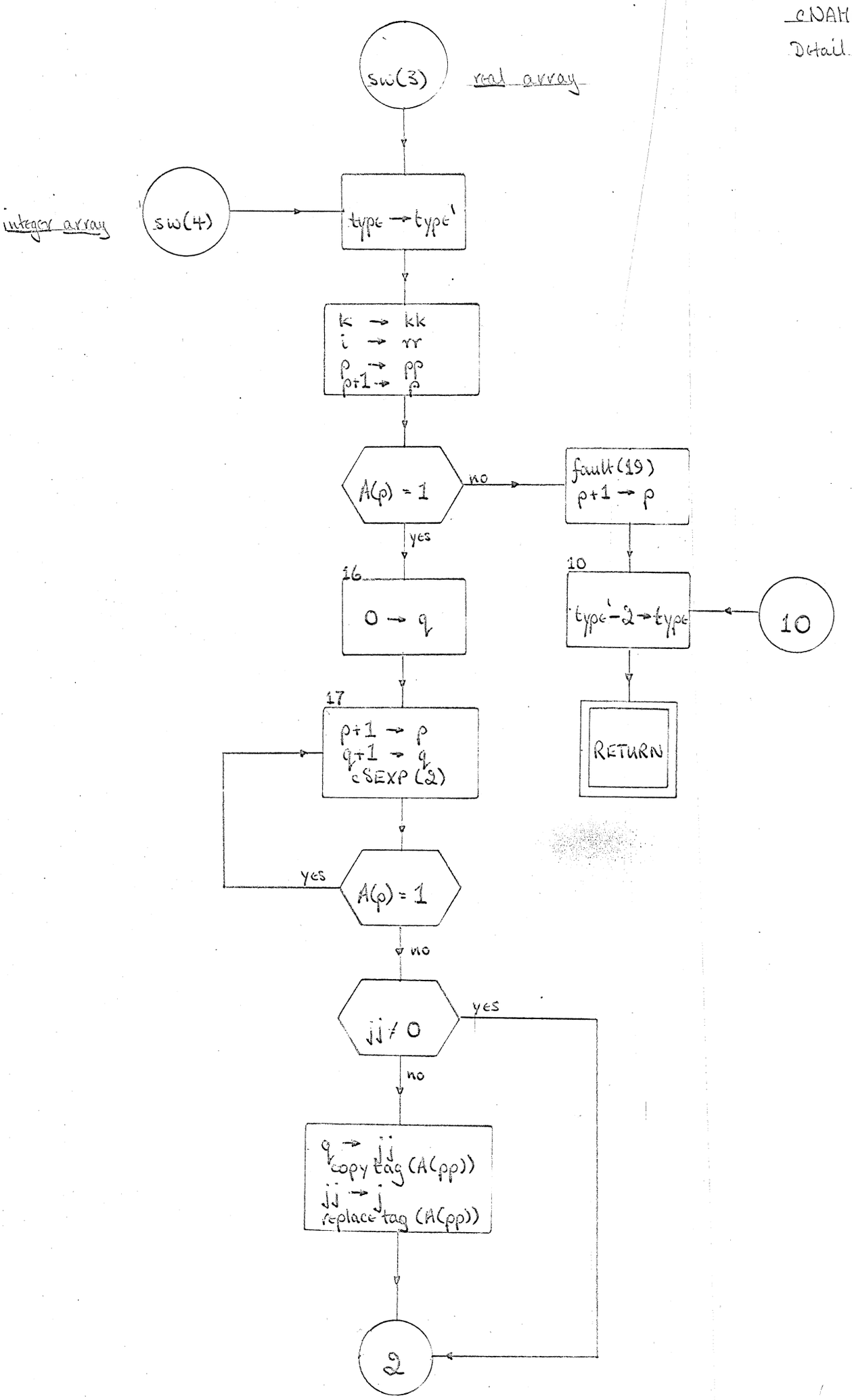


cNAME

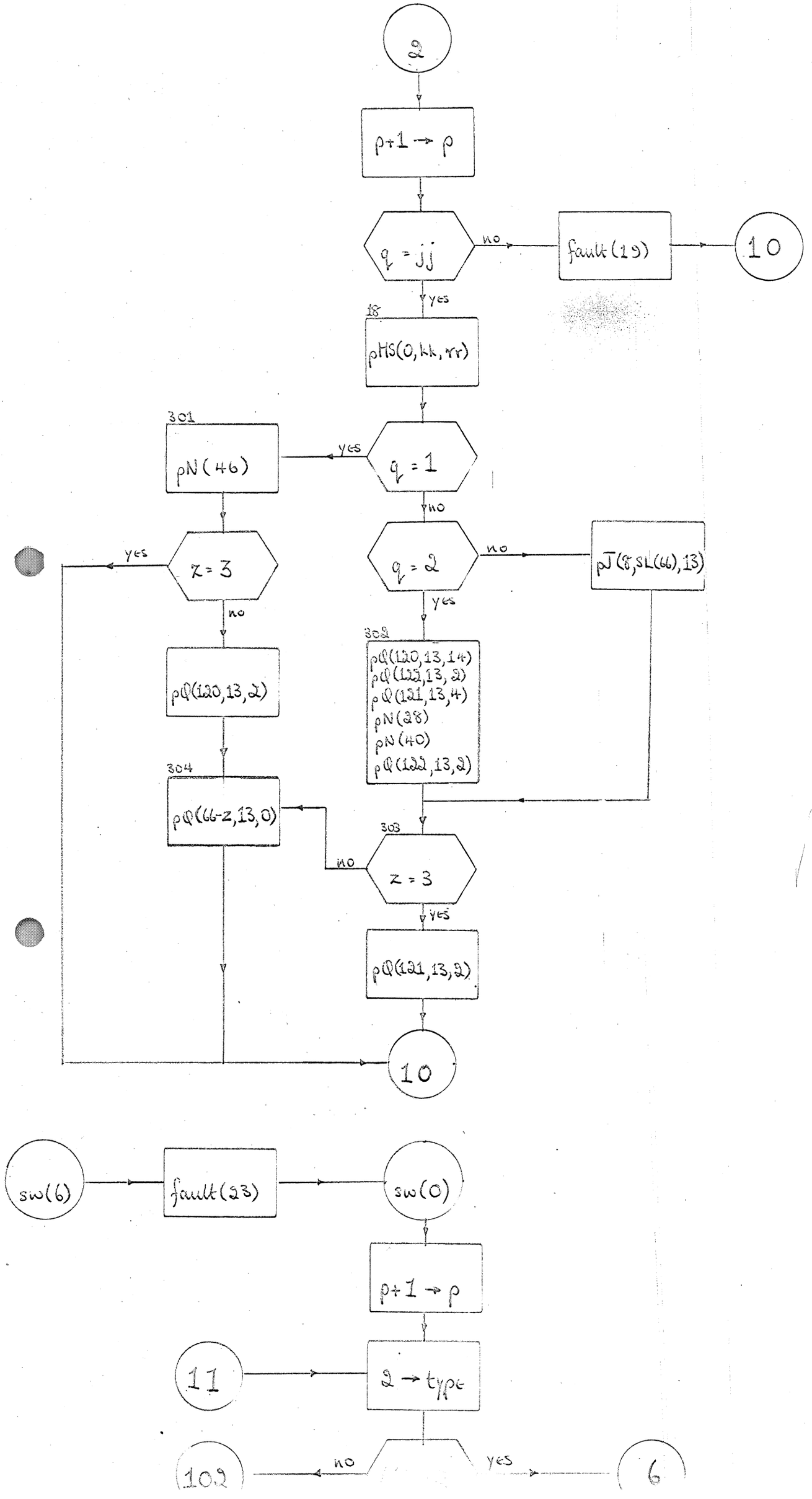
English



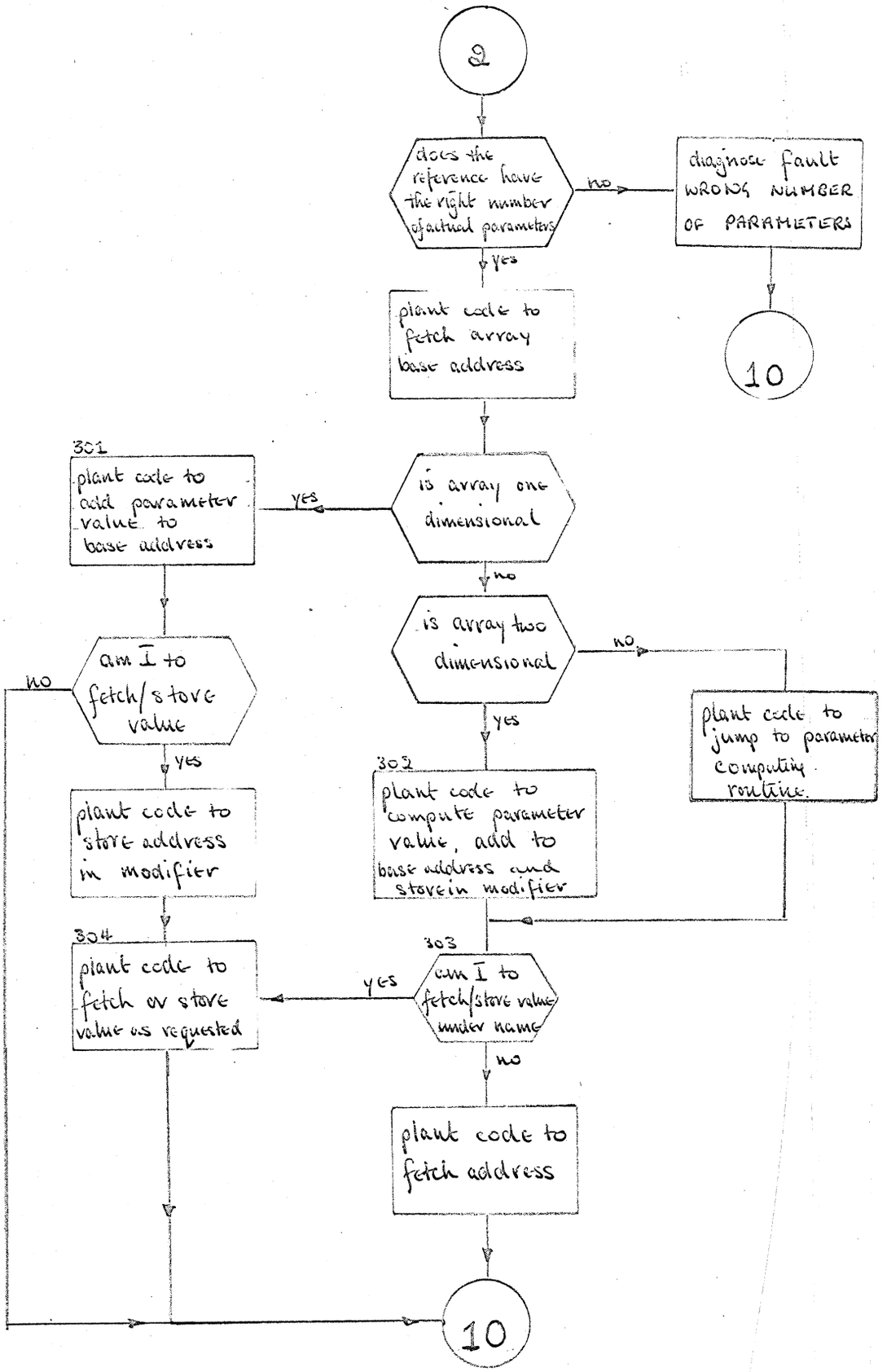
cNAME
Detail flow

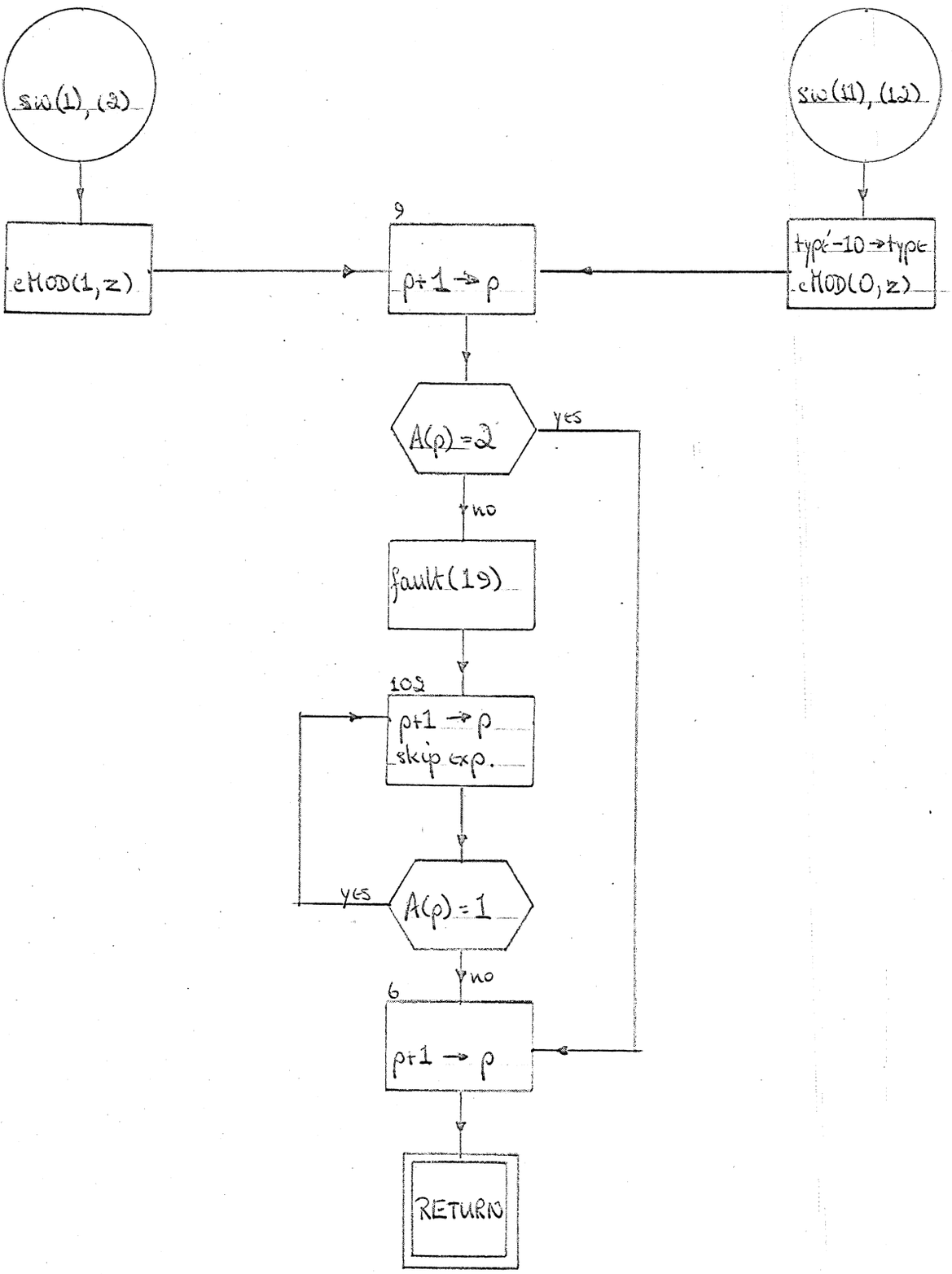


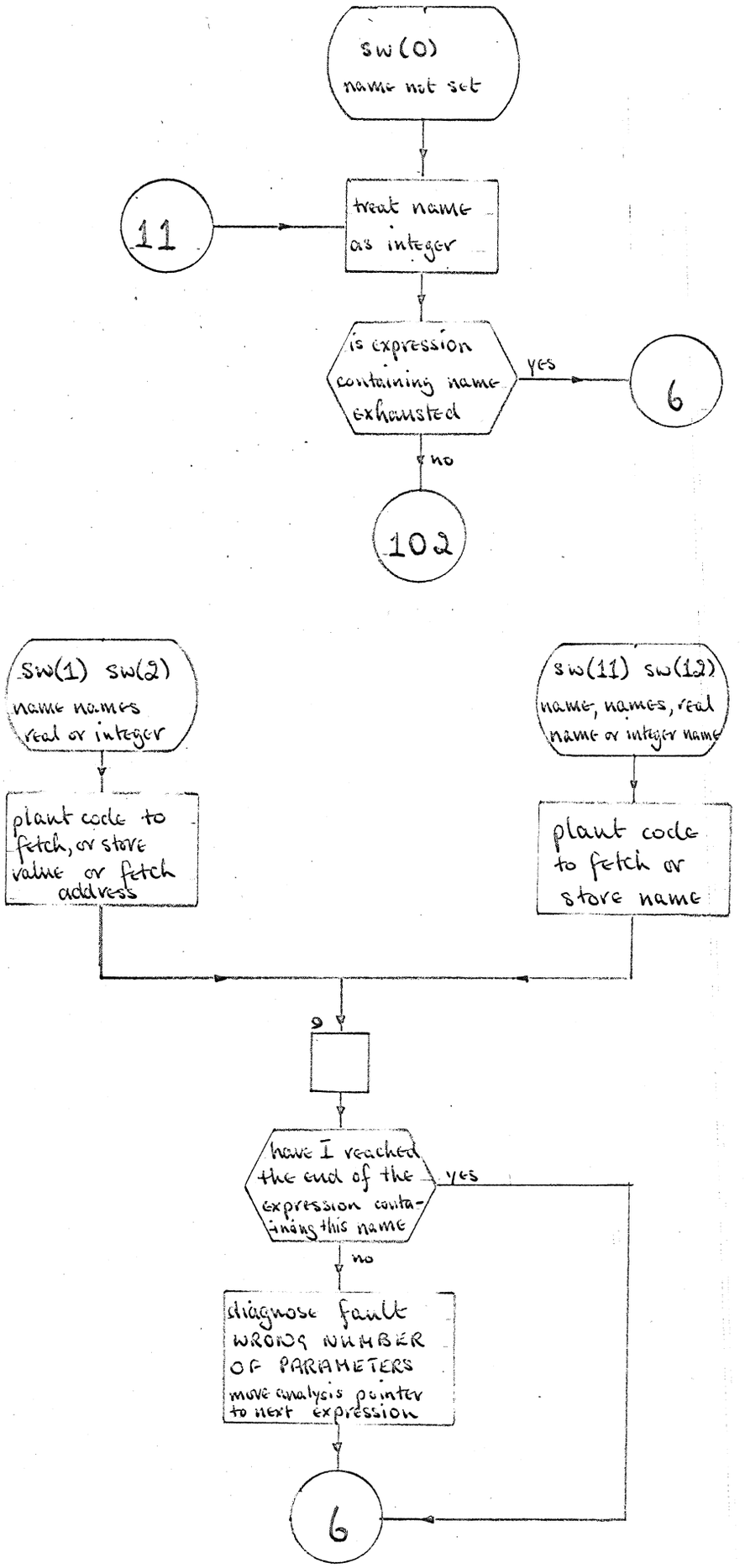
NAME

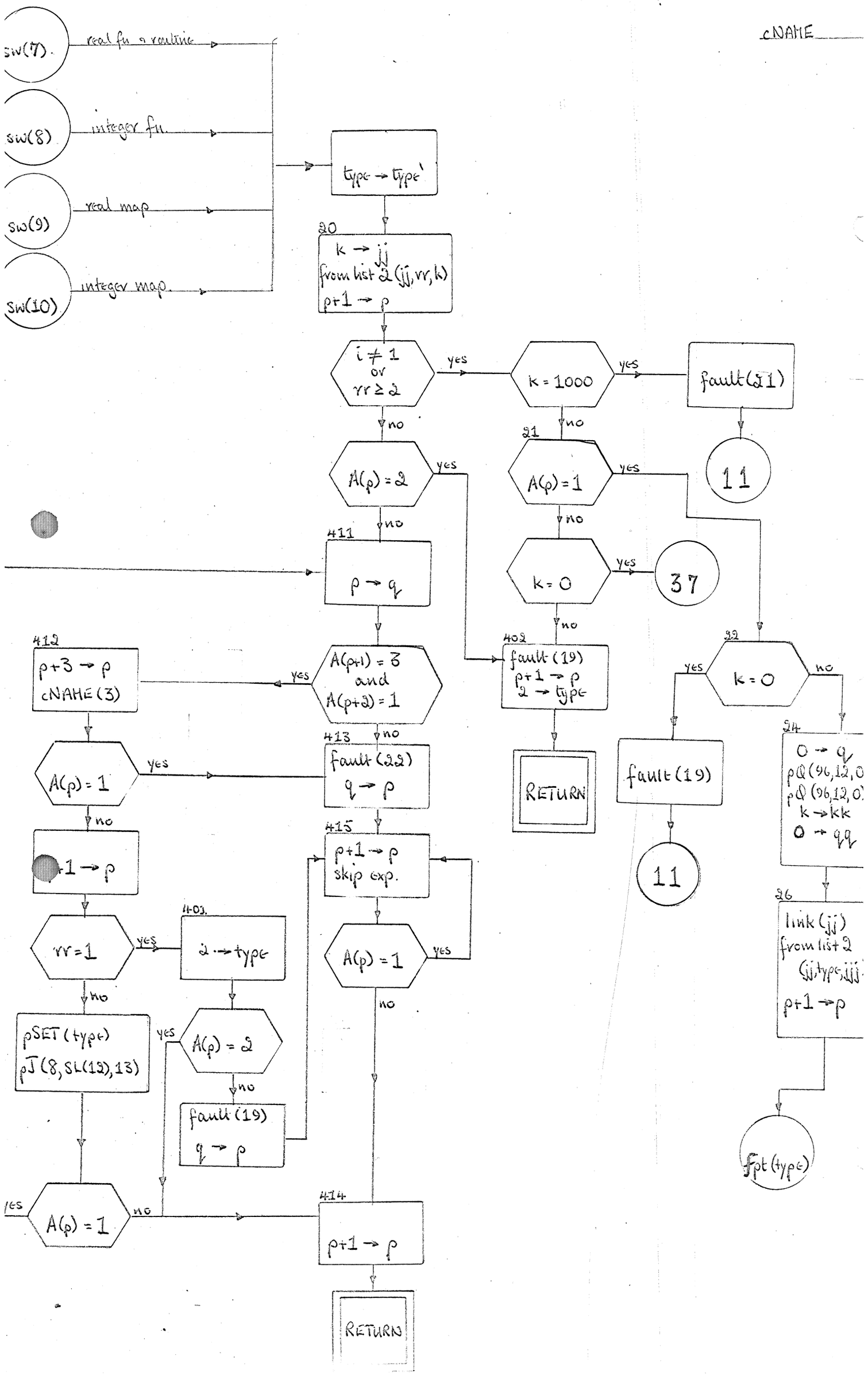

2
p+1 → p
q = jj
no
fault(19)
10
yes
18
pHS(0,kk,rr)
301
pN(46)
yes
q = 1
no
q = 2
no
pJ(8,SL(66),13)
yes
z = 3
yes
no
302
pQ(120,13,14)
pQ(122,13,2)
pQ(121,13,4)
pN(28)
pN(40)
pQ(122,13,2)
pQ(120,13,2)
304
pQ(66-z,13,0)
303
no
z = 3
yes
pQ(121,13,2)
10
sw(6)
fault(23)
sw(0)
p+1 → p
11
2 → type
10.2
no
yes
6

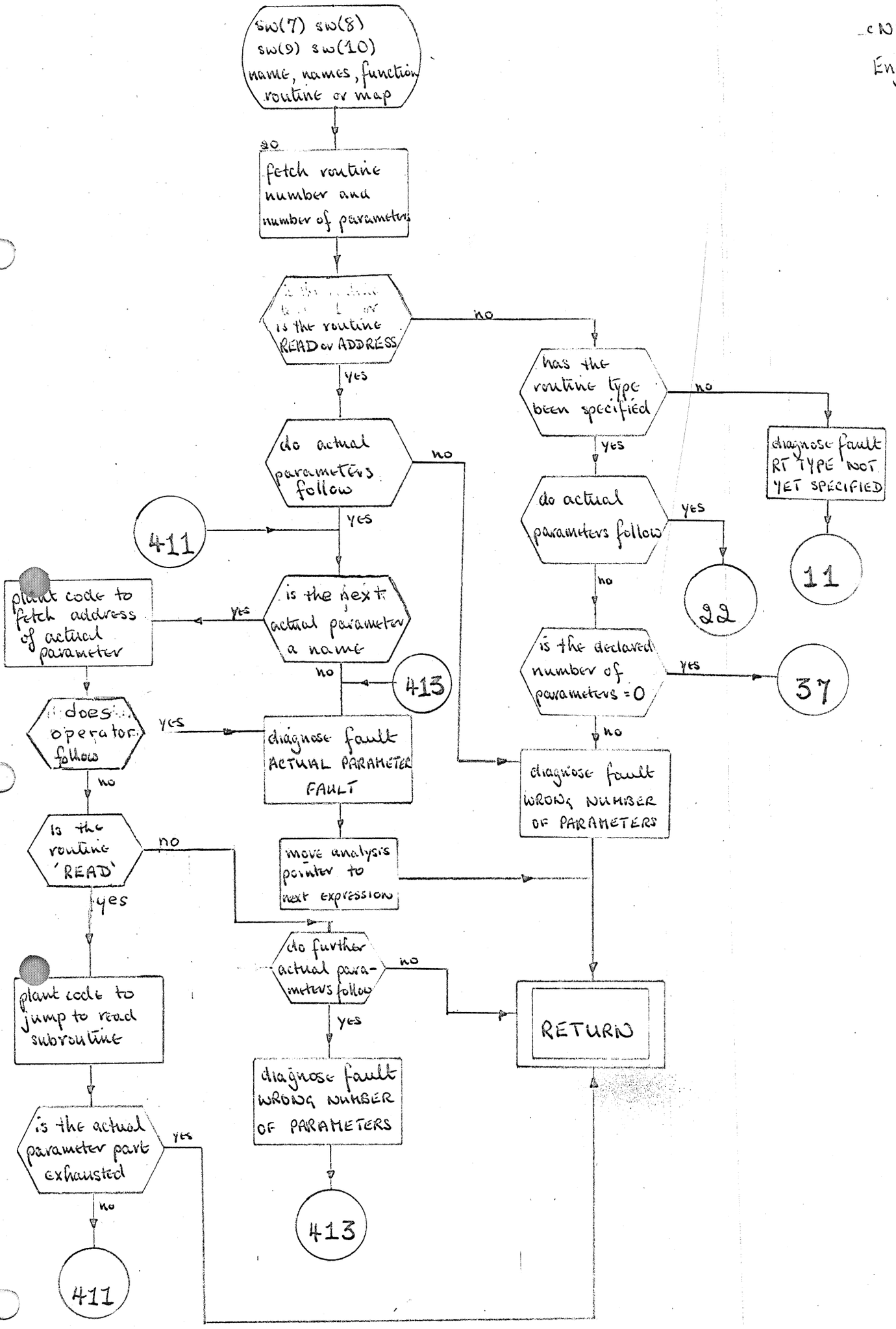
cNAME
English

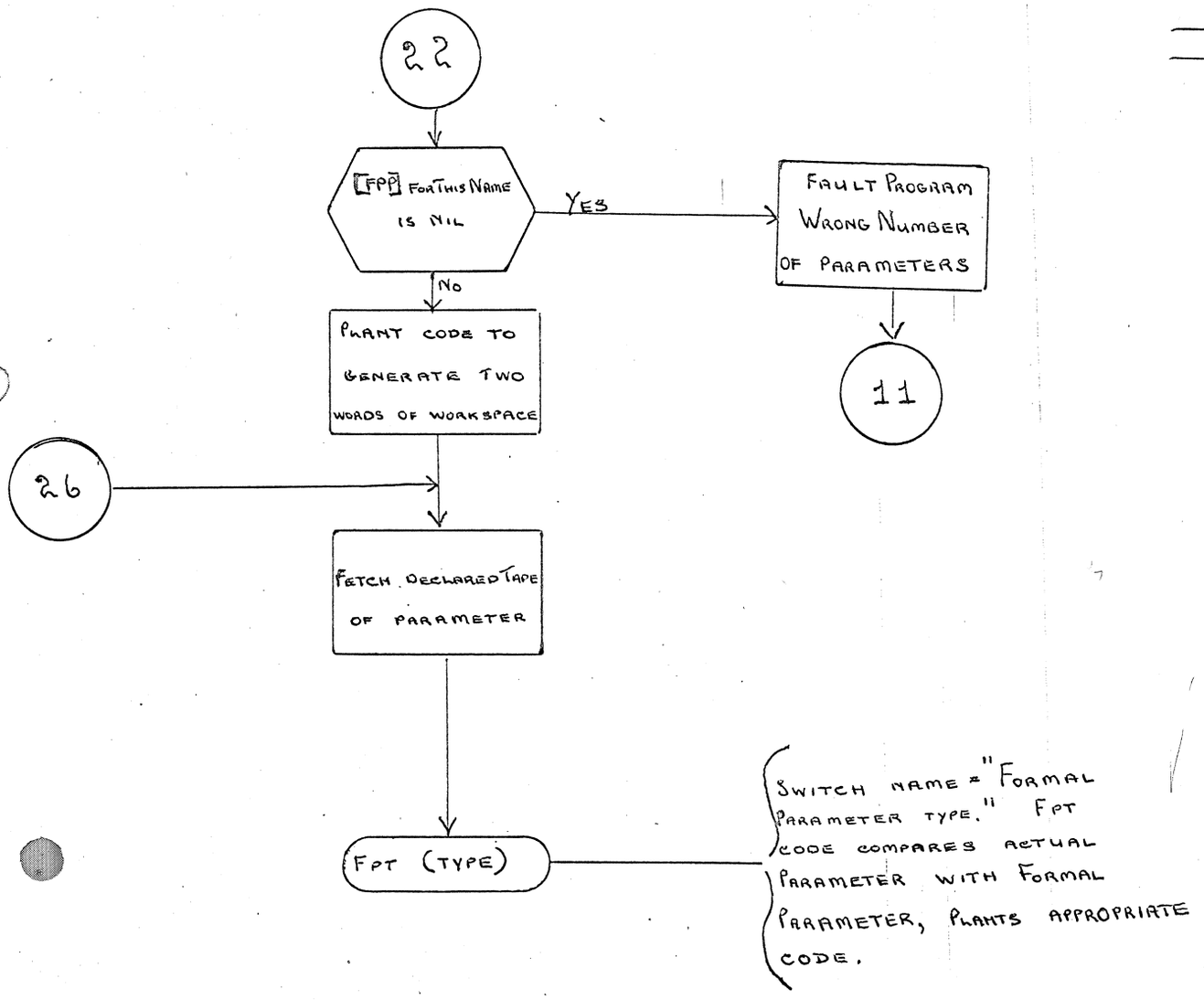


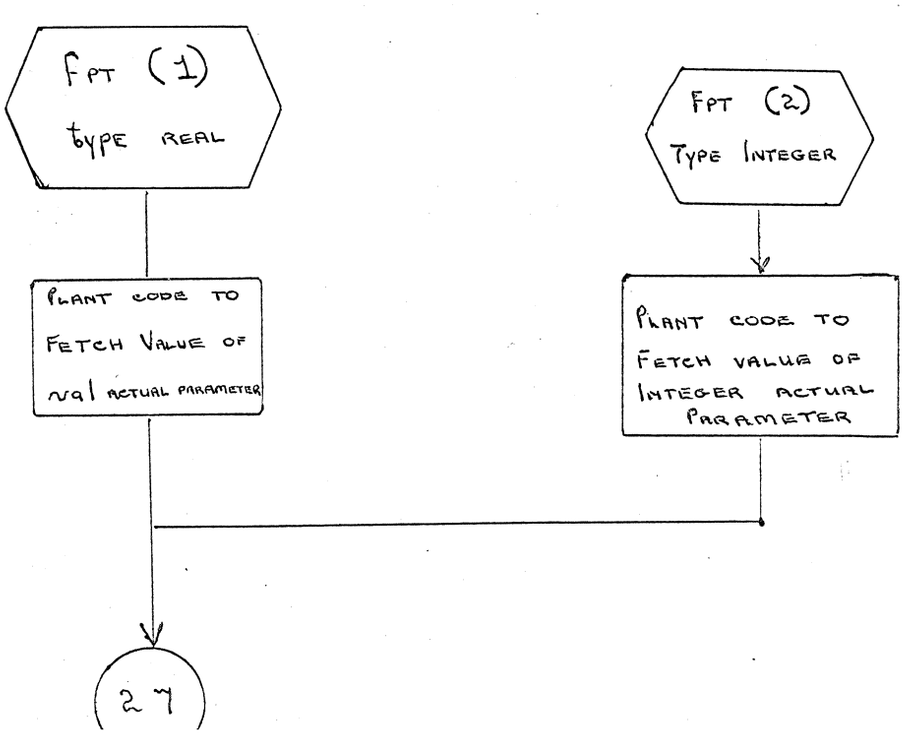

sw(1), (2)
sw(11), (12)
cMOD(1,z)
9
p+1 → p
type-10 → type
cMOD(0,z)
A(p)=2
yes
no
fault(19)
103
p+1 → p
skip exp.
yes
A(p)=1
no
6
p+1 → p
RETURN


sw(0)
name not set
11
treat name as integer
is expression containing name exhausted
yes
6
no
102
sw(1) sw(2)
name names real or integer
sw(11) sw(12)
name, names, real name or integer name
plant code to fetch, or store value or fetch address
plant code to fetch or store name
9
have I reached the end of the expression containing this name
yes
no
diagnose fault WRONG NUMBER OF PARAMETERS
move analysis pointer to next expression
6
sw(6)
diagnose fault ROUTINE NAME IN EXPRESSION
sw(0)

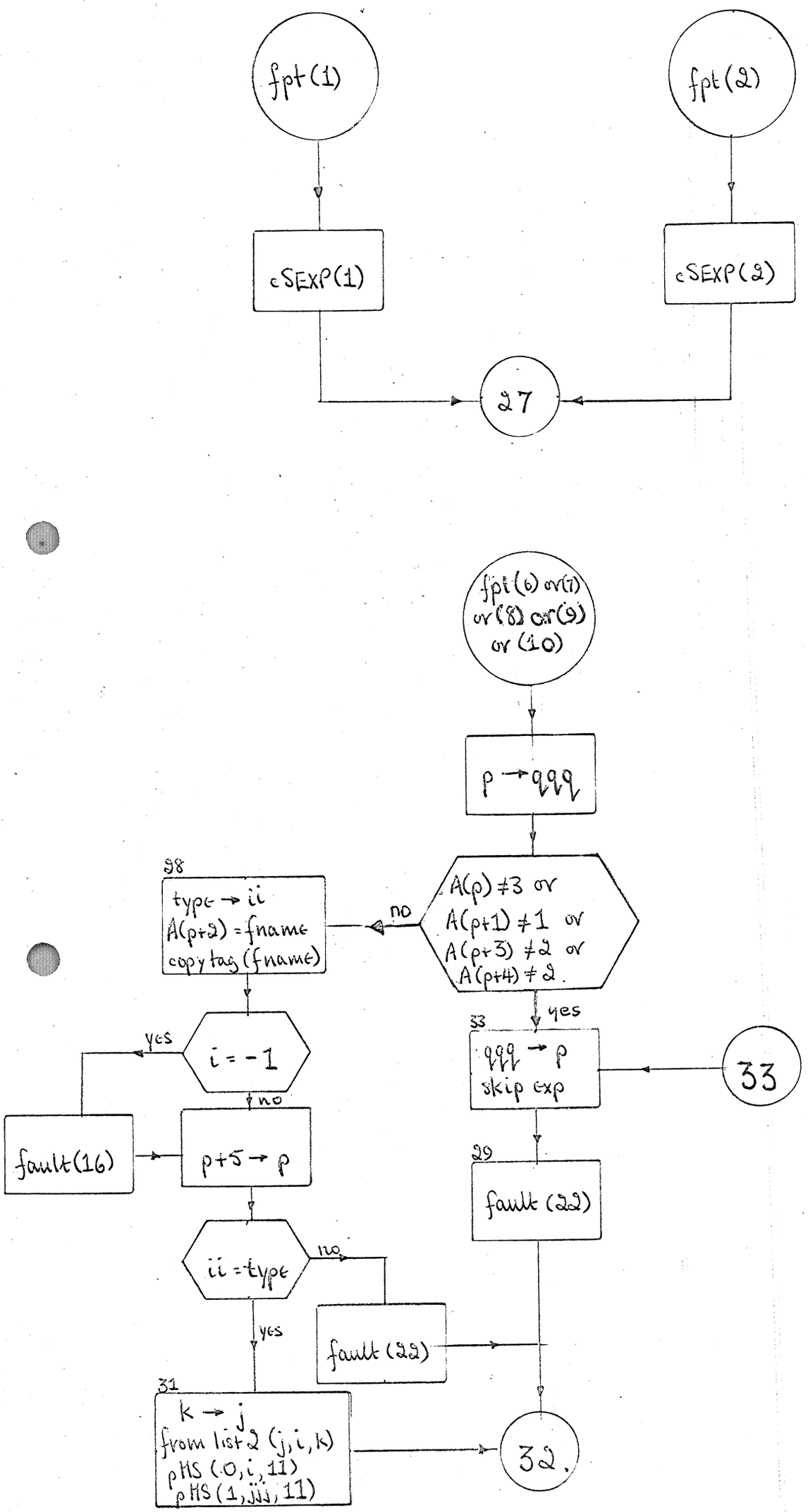

sw(7)
real fn a routine
sw(8)
integer fn
sw(9)
real map
sw(10)
integer map.
type → type'
20
k → jj
from list 2 (jj, rr, k)
p+1 → p
i ≠ 1 or rr ≥ 2
yes
k = 1000
yes
fault (21)
no
21
A(p) = 2
yes
A(p) = 1
yes
11
no
411
p → q
k = 0
yes
37
no
412
p+3 → p
cNAME (3)
A(p+1) = 3 and A(p+2) = 1
yes
402
fault (19)
p+1 → p
2 → type
RETURN
22
k = 0
yes
no
A(p) = 1
yes
413
fault (22)
q → p
fault (19)
11
24
0 → q
pQ (96,12,0
pQ (96,12,0
k → kk
0 → qq
no
p+1 → p
415
p+1 → p
skip exp.
26
link (jj)
from list 2
(jj, type, jjj
p+1 → p
rr = 1
yes
403
i → type
A(p) = 2
yes
fpt (type)
no
pSET (type)
pJ (8, SL(12), 13)
fault (19)
q → p
A(p) = 1
yes
no
414
p+1 → p
RETURN

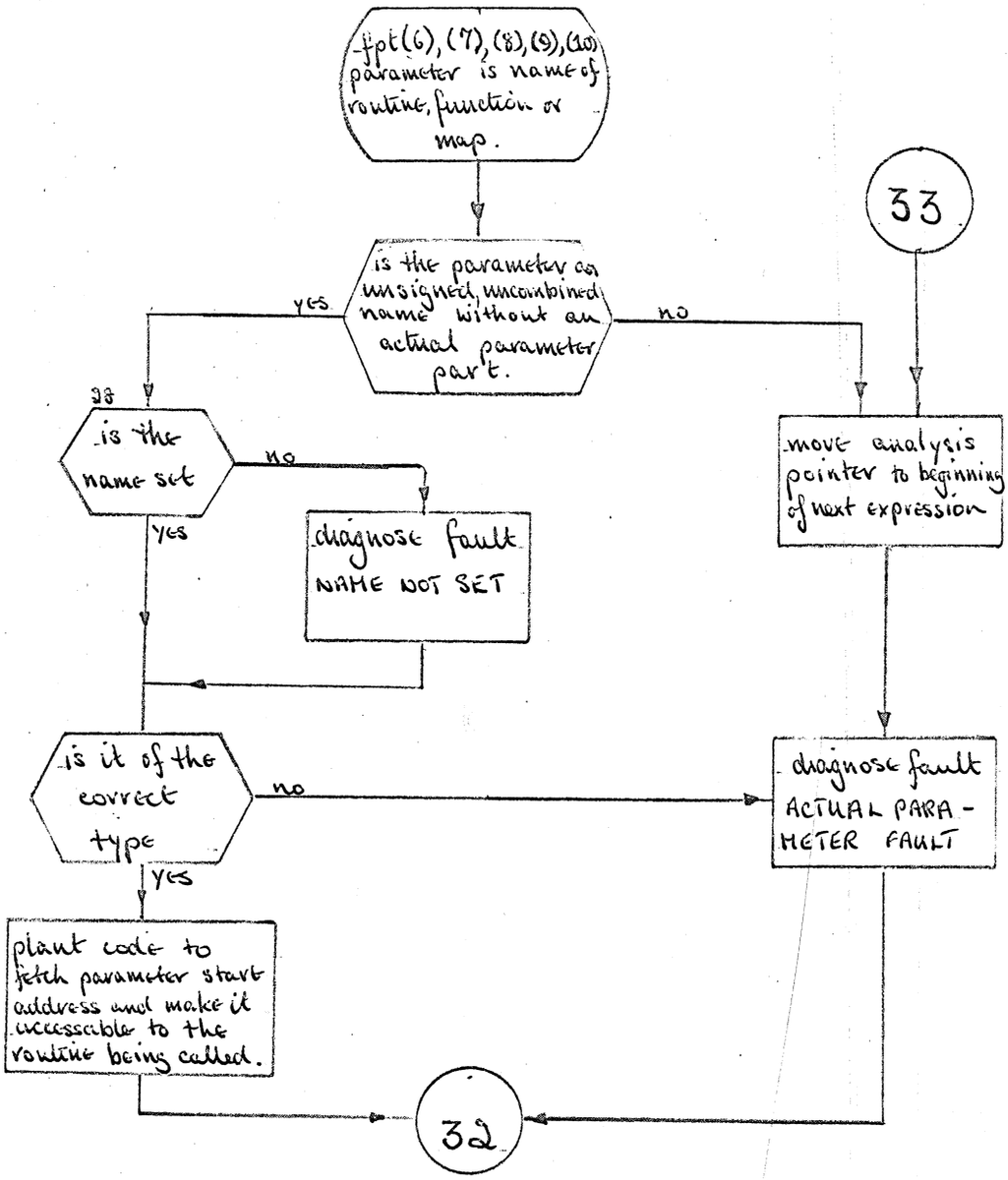

sw(7) sw(8)
sw(9) sw(10)
name, names, function routine or map
go
fetch routine number and number of parameters
is the routine READ or ADDRESS
no
yes
has the routine type been specified
no
yes
diagnose fault RT TYPE NOT YET SPECIFIED
11
do actual parameters follow
no
yes
411
do actual parameters follow
yes
22
no
plant code to fetch address of actual parameter
yes
is the next actual parameter a name
is the declared number of parameters = 0
yes
37
no
413
does operator follow
yes
diagnose fault ACTUAL PARAMETER FAULT
diagnose fault WRONG NUMBER OF PARAMETERS
no
is the routine 'READ'
no
move analysis pointer to next expression
yes
do further actual parameters follow
no
RETURN
plant code to jump to read subroutine
yes
diagnose fault WRONG NUMBER OF PARAMETERS
is the actual parameter part exhausted
yes
no
413
411

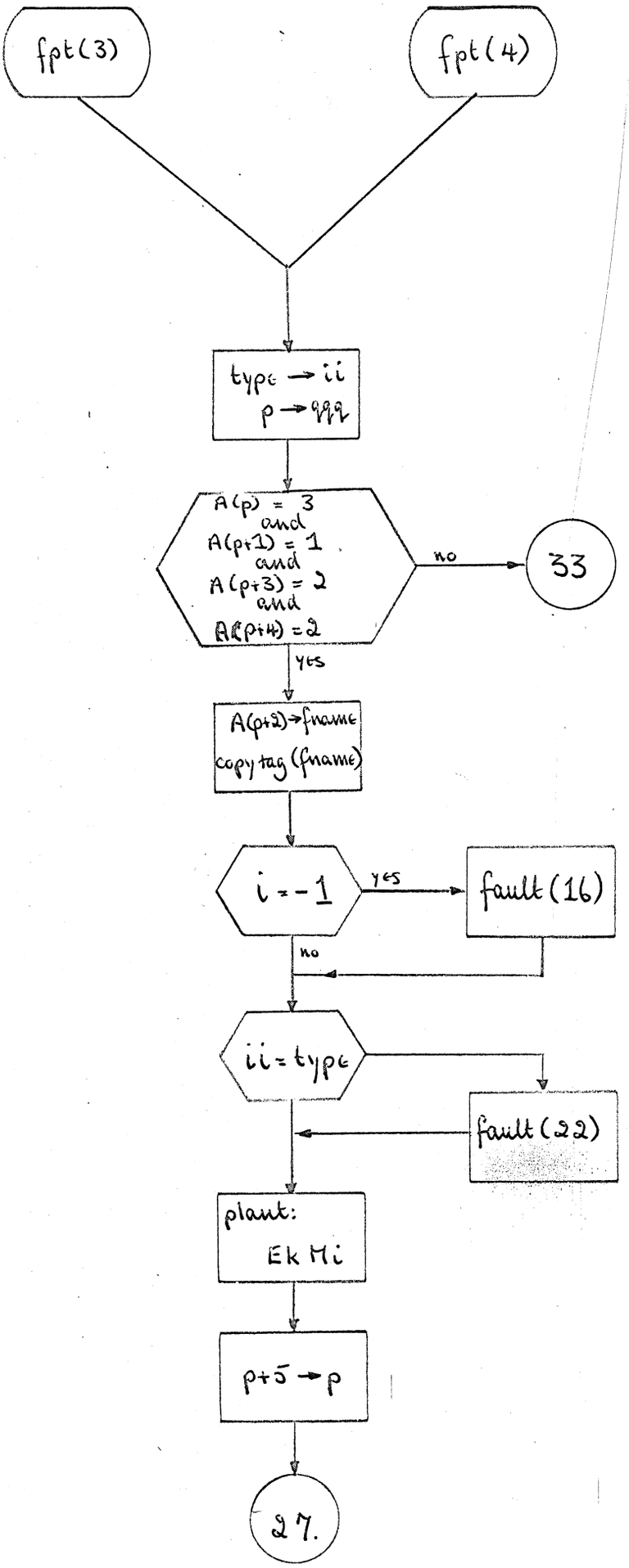

22
[FPP] FOR THIS NAME IS NIL
YES
FAULT PROGRAM WRONG NUMBER OF PARAMETERS
11
NO
PLANT CODE TO GENERATE TWO WORDS OF WORKSPACE
26
FETCH DECLARED TAPE OF PARAMETER
FPT (TYPE)
SWITCH NAME = "FORMAL PARAMETER TYPE." FPT CODE COMPARES ACTUAL PARAMETER WITH FORMAL PARAMETER, PLANTS APPROPRIATE CODE.

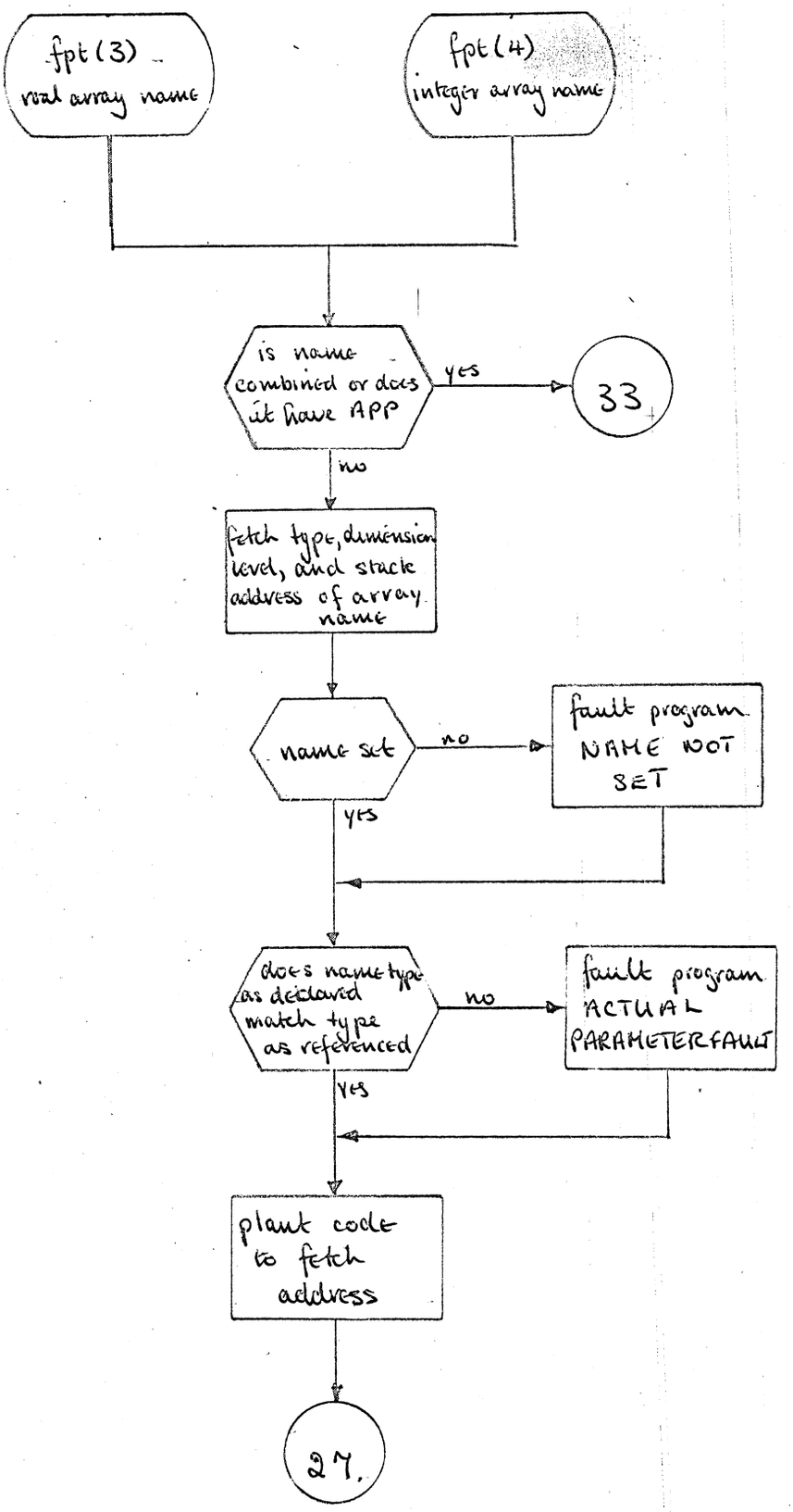

FPT (1)
type REAL
FPT (2)
Type INTEGER
PLANT CODE TO FETCH VALUE OF val ACTUAL PARAMETER
PLANT CODE TO FETCH VALUE OF INTEGER ACTUAL PARAMETER
27

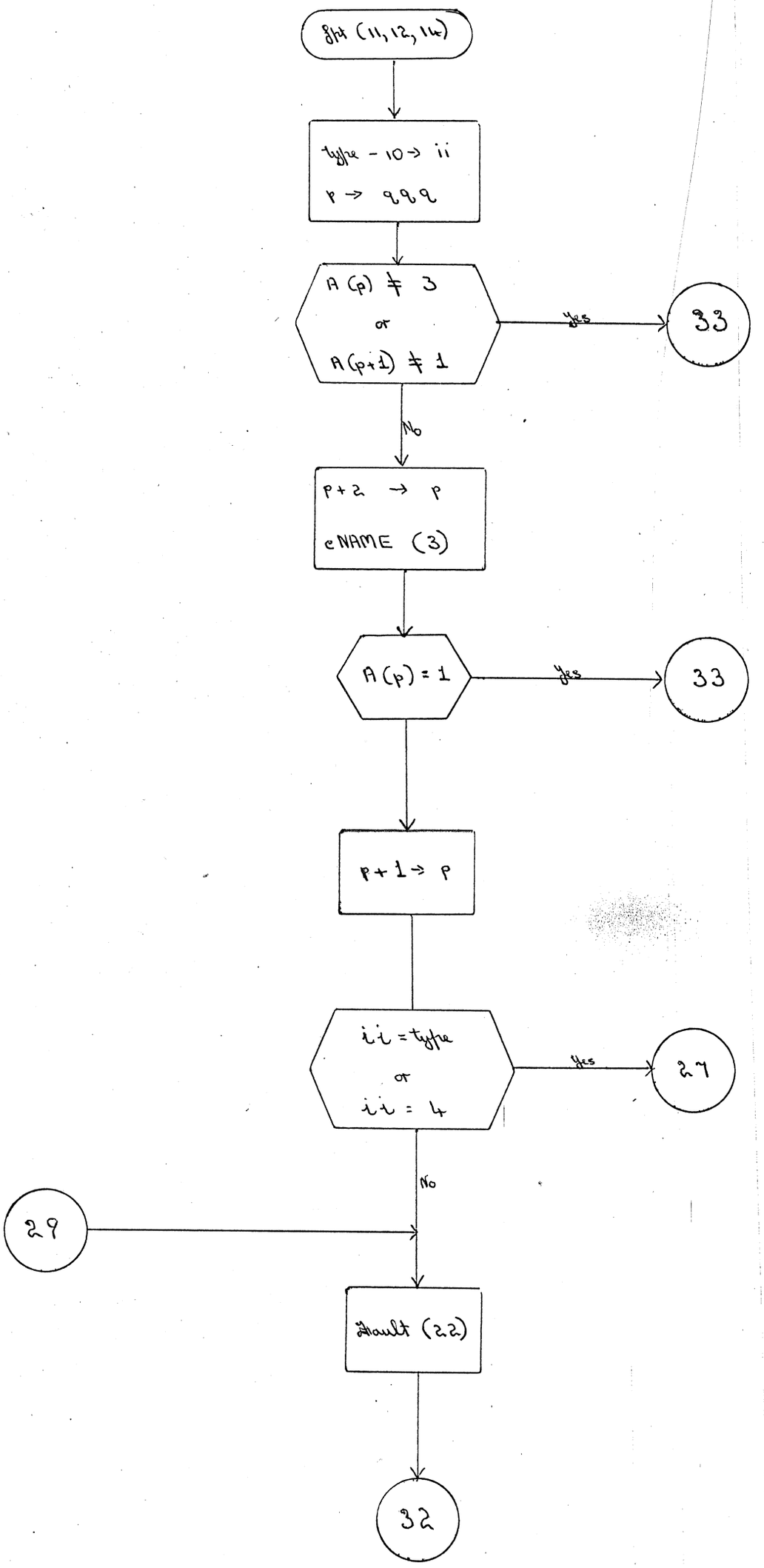
cNAME.


fpt(1)
fpt(2)
cSEXP(1)
cSEXP(2)
27
fpt(6) or(7) or(8) or(9) or(10)
p → 999
98
type → ii
A(p+2) = fname
copy tag (fname)
no
A(p) ≠3 or A(p+1) ≠1 or A(p+3) ≠2 or A(p+4) ≠2.
yes
33
999 → p
skip exp
33
yes
i = -1
no
fault(16)
p+5 → p
29
fault (22)
ii = type
no
yes
fault (22)
31
k → j
from list2 (j,i,k)
pHS (0,i,11)
pHS (1,jj,11)
32.

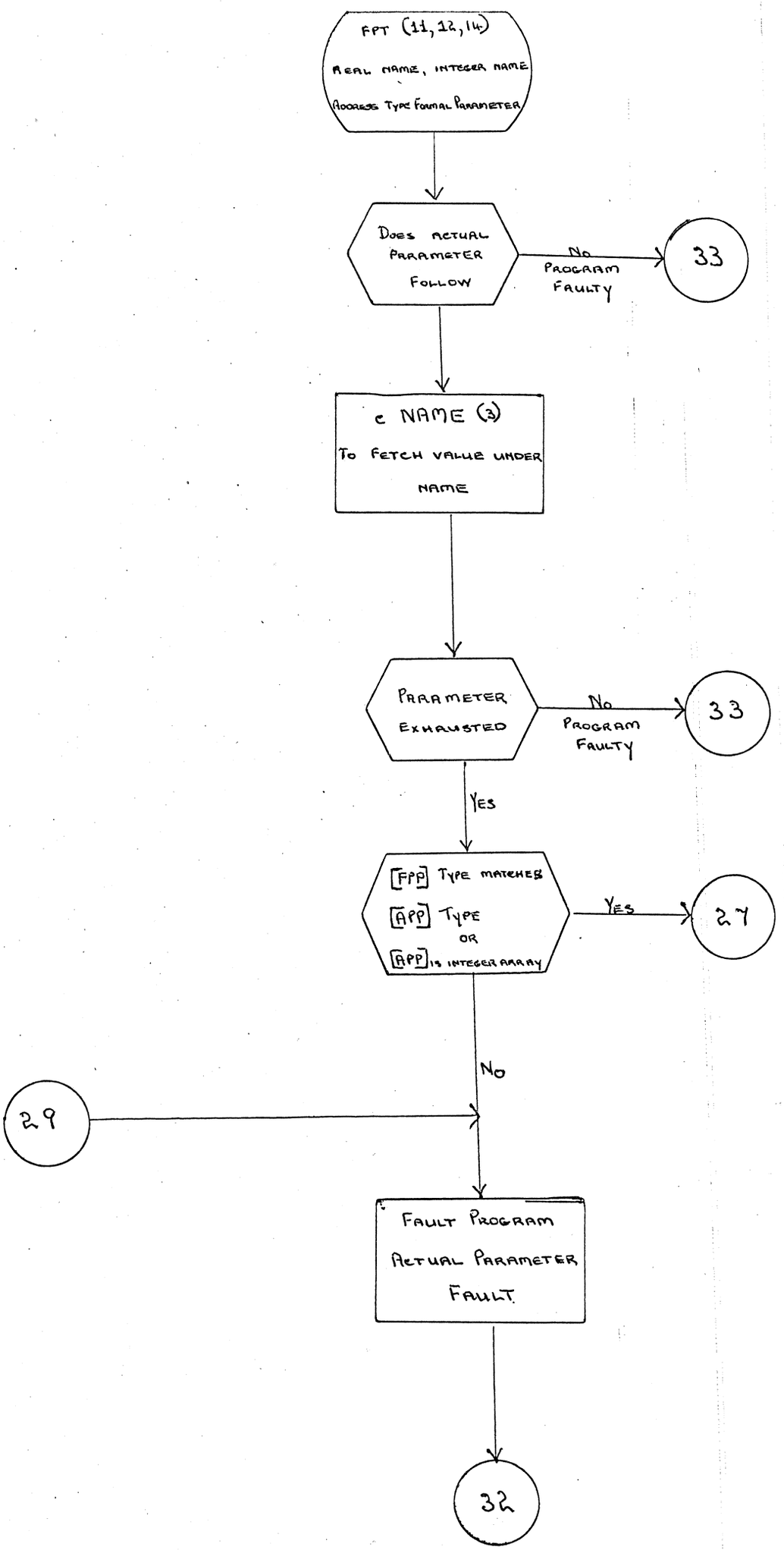

fpt(6),(7),(8),(9),(10) parameter is name of routine, function or map.
is the parameter an unsigned, uncombined name without an actual parameter part.
yes
no
33
33
is the name set
no
yes
diagnose fault NAME NOT SET
move analysis pointer to beginning of next expression
is it of the correct type
no
yes
diagnose fault ACTUAL PARA-METER FAULT
plant code to fetch parameter start address and make it accessible to the routine being called.
32

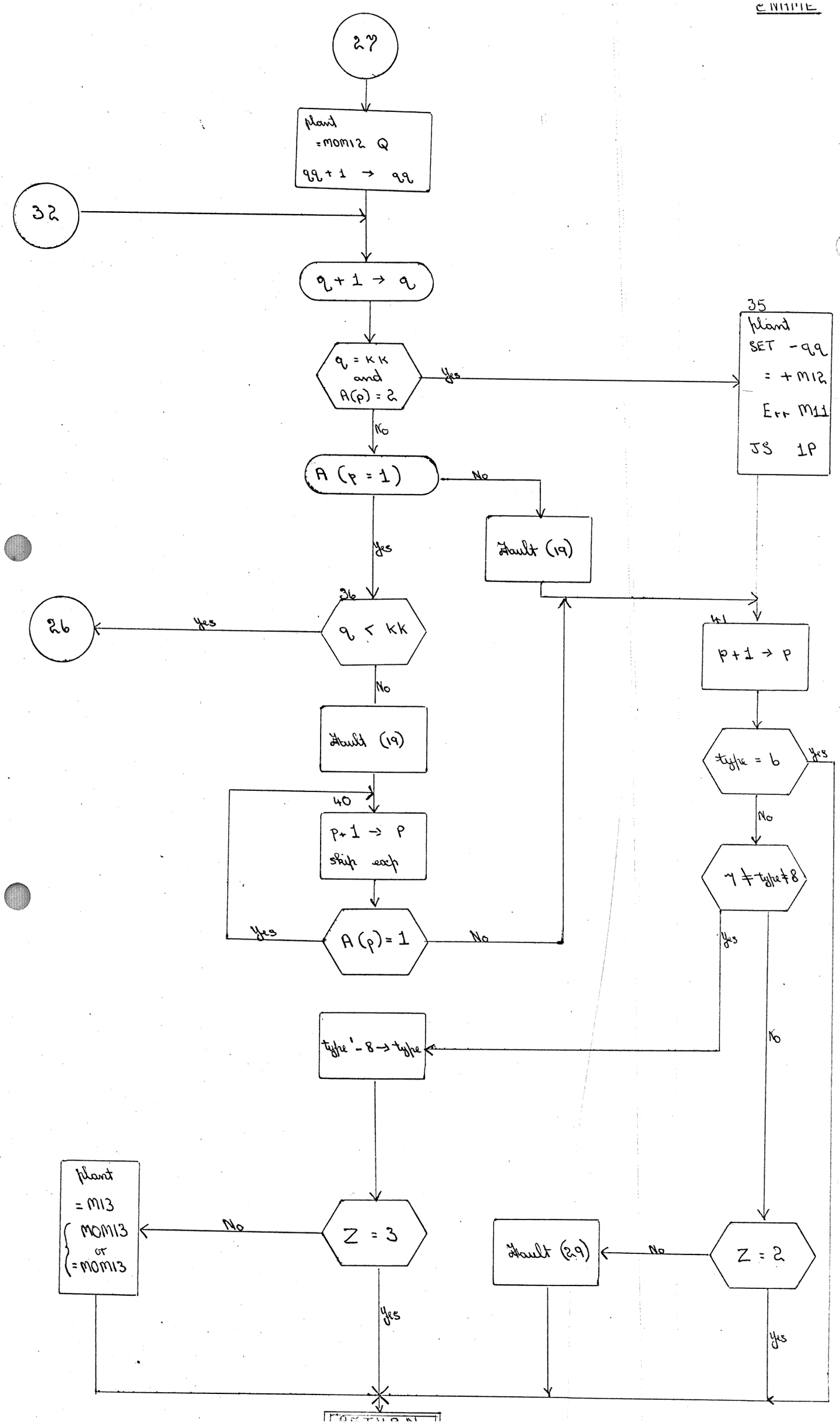

fpt(3)
fpt(4)
type → ii
p → ppp
A(p) = 3 and A(p+1) = 1 and A(p+3) = 2 and A(p+4) = 2
no
33
yes
A(p+2) → fname
copy tag (fname)
i = -1
yes
fault (16)
no
ii = type
fault (22)
plant:
Ek Mi
p+5 → p
27.

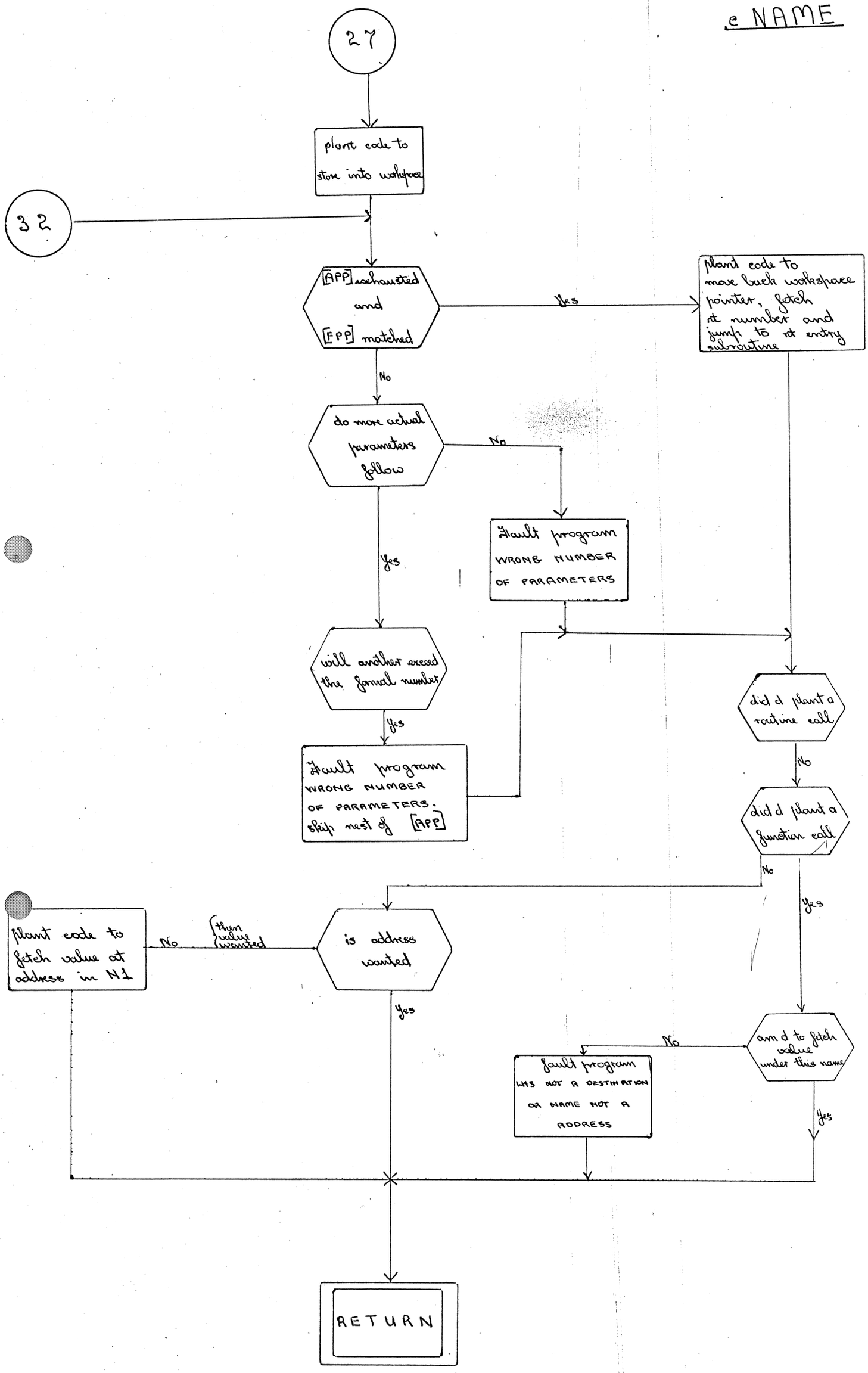

fpt(3)
real array name
fpt(4)
integer array name
is name combined or does it have APP
yes
33
no
fetch type, dimension level, and stack address of array name
name set
no
fault program NAME NOT SET
yes
does name type as declared match type as referenced
no
fault program ACTUAL PARAMETERFAULT
yes
plant code to fetch address
27.

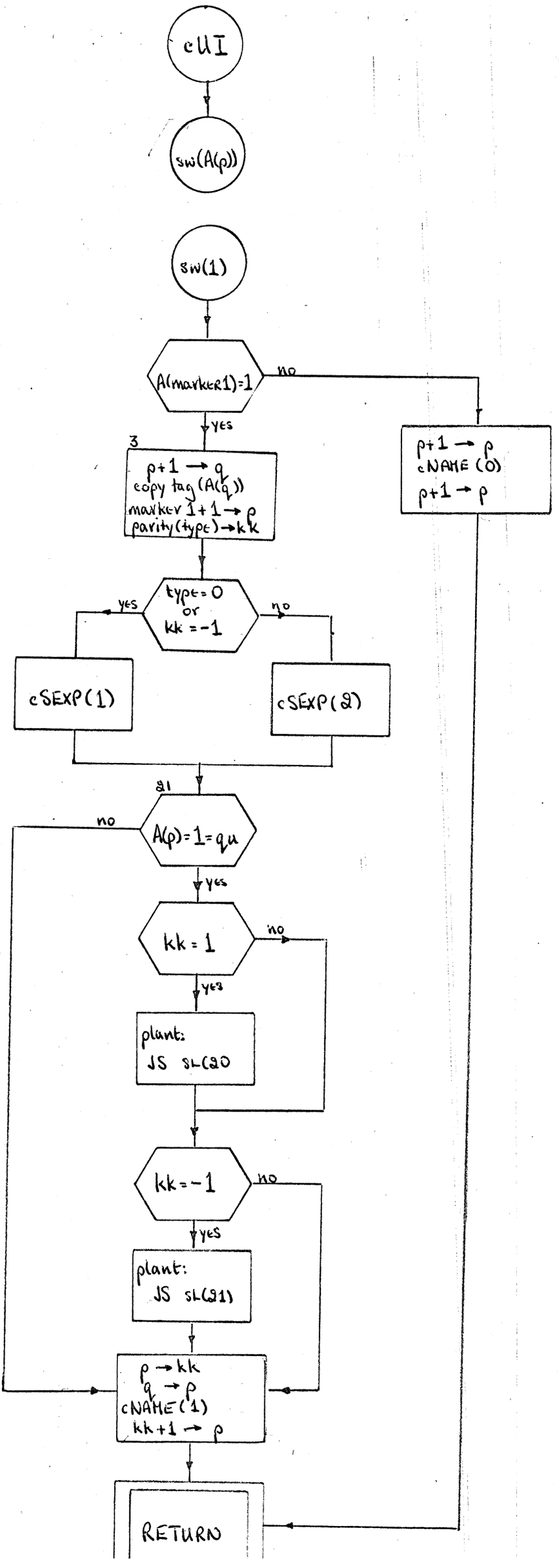

get (11, 12, 14)
type - 10 → ii
p → qqq
A (p) ≠ 3
or
A (p+1) ≠ 1
yes
33
No
p + 2 → p
eNAME (3)
A (p) = 1
yes
33
p + 1 → p
ii = type
or
ii = 4
yes
27
No
29
fault (22)
32

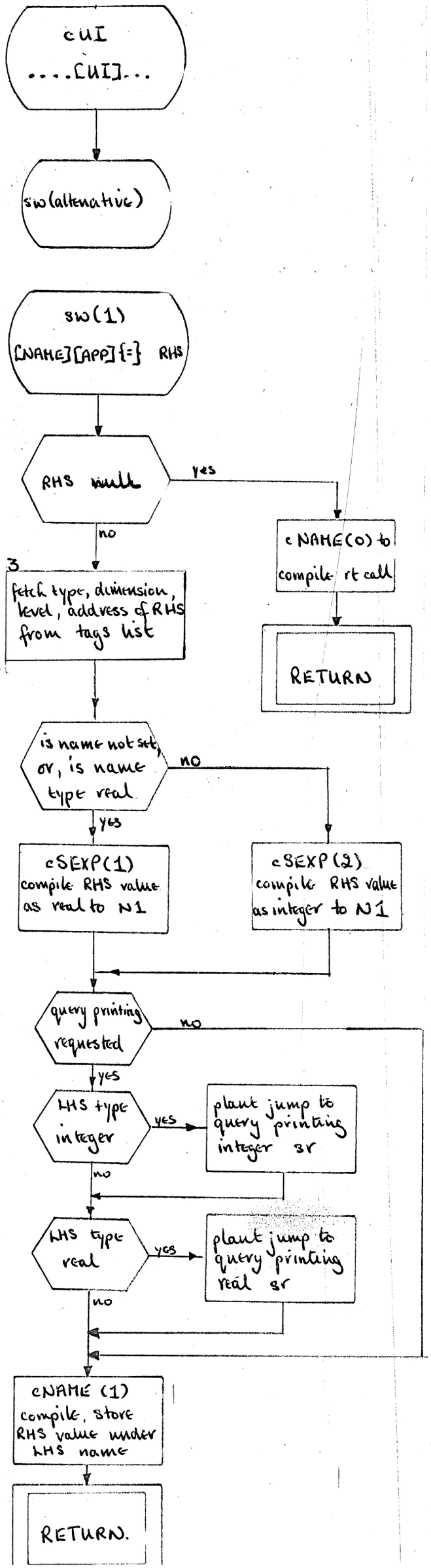
c NAME
ENGLIS




27
plant
=MOM12 Q
qq + 1 → qq
32
q + 1 → q
q = KK
and
A(p) = 2
Yes
35
plant
SET -qq
= +M12
Err M11
JS 1P
No
A (p = 1)
No
Yes
Fault (19)
36
26
Yes
q < KK
No
Fault (19)
41
p + 1 → p
40
p + 1 → p
skip each
type = 6
Yes
No
A (p) = 1
Yes
No
7 ≠ type ≠ 8
Yes
No
type' - 8 → type
plant
= M13
{ MOM13
or
= MOM13
Z = 3
No
Yes
Fault (29)
No
Z = 2
Yes
Yes

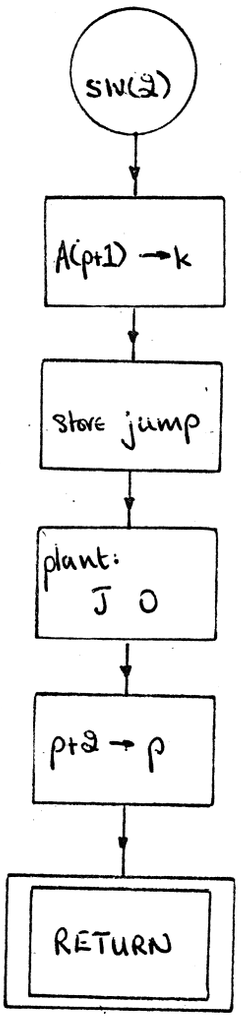

27
plant code to store into workspace
32
[APP] exhausted and [FPP] matched
Yes
plant code to move back workspace pointer, fetch it number and jump to it entry subroutine
No
do more actual parameters follow
No
Yes
fault program WRONG NUMBER OF PARAMETERS
will another exceed the formal number
Yes
fault program WRONG NUMBER OF PARAMETERS. skip rest of [APP]
did d plant a routine call
No
did d plant a function call
No
Yes
is address wanted
No
then value wanted
plant code to fetch value at address in N1
Yes
comd to fetch value under this name
No
fault program LHS NOT A DESTINATION OR NAME NOT A ADDRESS
Yes
RETURN


cUI
sw(A(p))
sw(1)
A(marker1)=1
no
yes
3
p+1 → q
copy tag(A(q))
marker 1+1 → p
parity(type) → kk
p+1 → p
cNAME (0)
p+1 → p
type=0 or kk=-1
yes
no
cSEXP(1)
cSEXP(2)
91
A(p)=1=qu
no
yes
kk = 1
no
yes
plant:
JS SL(20
kk=-1
no
yes
plant:
JS SL(21)
p → kk
q → p
cNAME(1)
kk+1 → p
RETURN

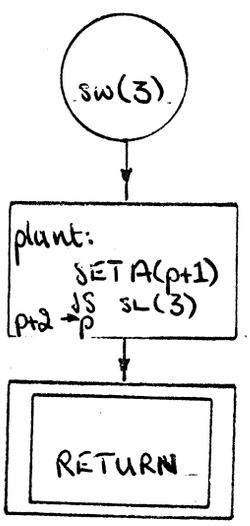

cUI
....[UI]...
sw(alternative)
sw(1)
[NAME][APP]{=} RHS
RHS null
yes
no
cNAME(0) to compile rt call
RETURN
3
fetch type, dimension, level, address of RHS from tags list
is name not set, or, is name type real
no
yes
cSEXP(1) compile RHS value as real to N1
cSEXP(2) compile RHS value as integer to N1
query printing requested
no
yes
LHS type integer
yes
plant jump to query printing integer sr
no
LHS type real
yes
plant jump to query printing real sr
no
cNAME(1) compile, store RHS value under LHS name
RETURN.

→N

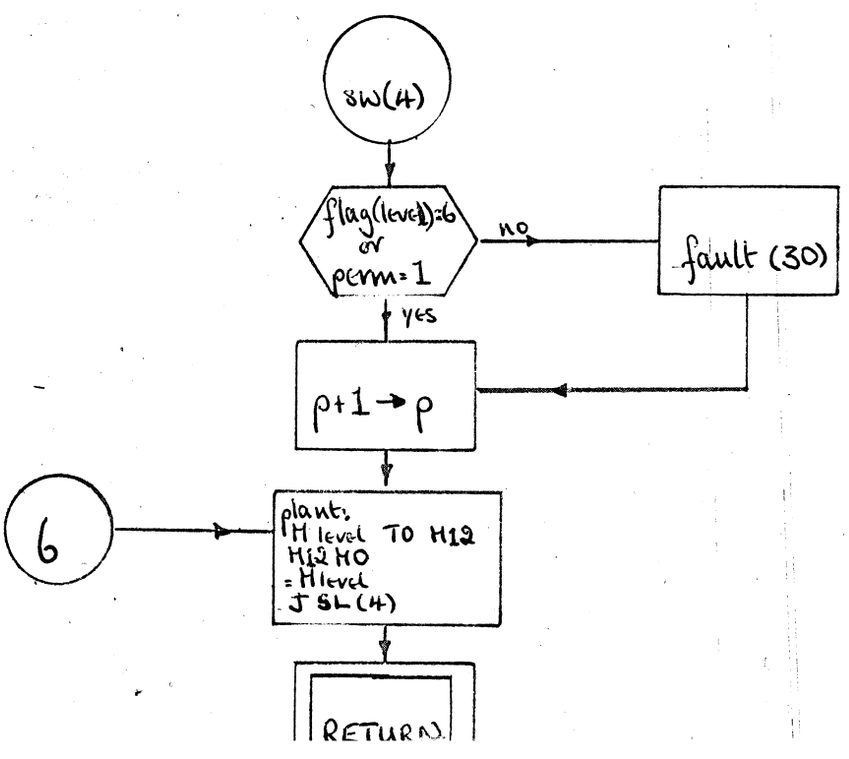
cUI.

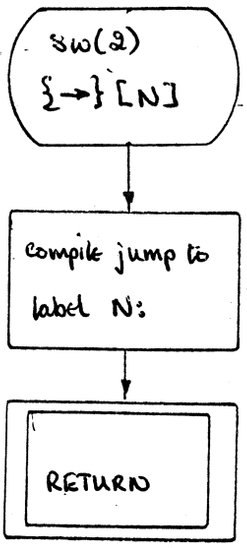


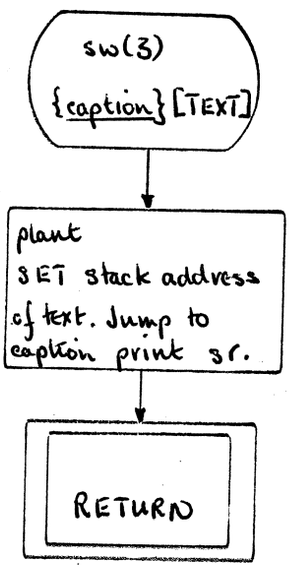
caption text

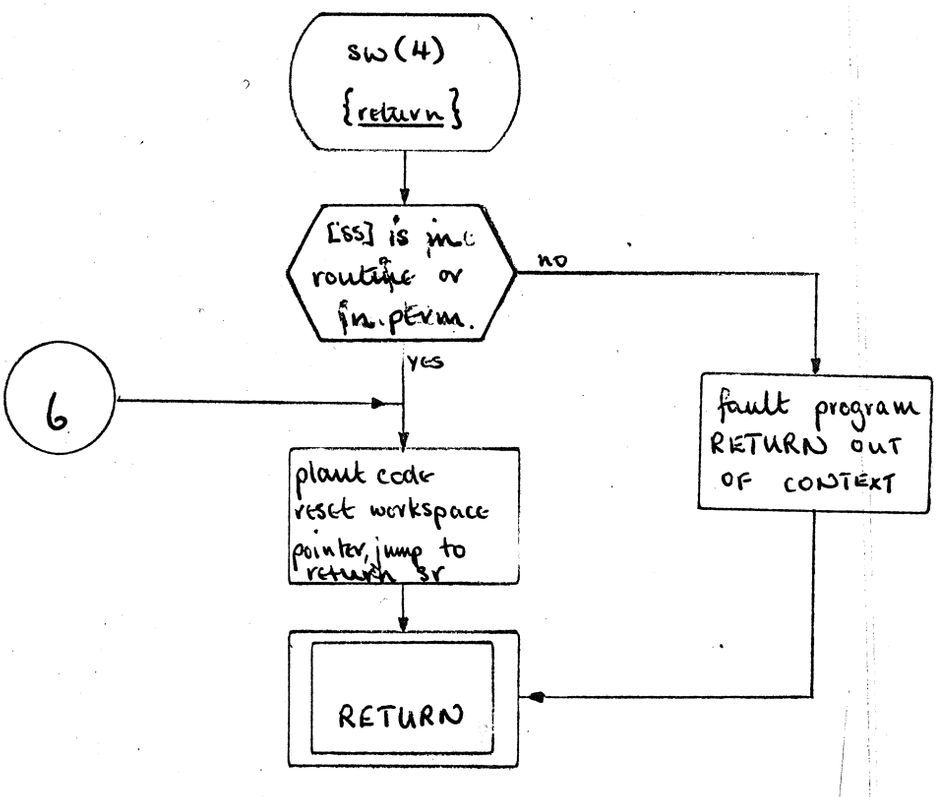


return

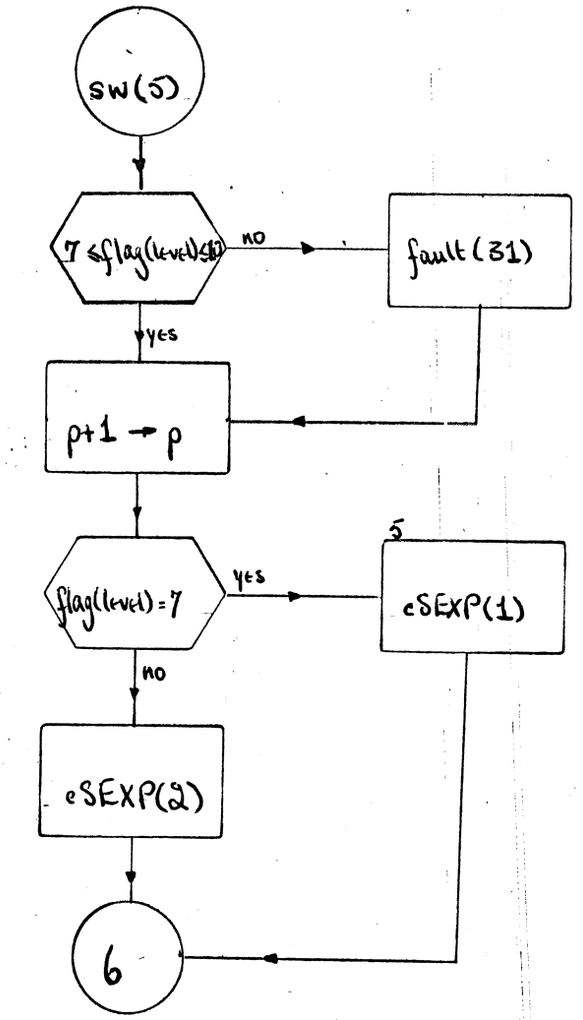



sw(2)
{→} [N]
compile jump to label N:
RETURN

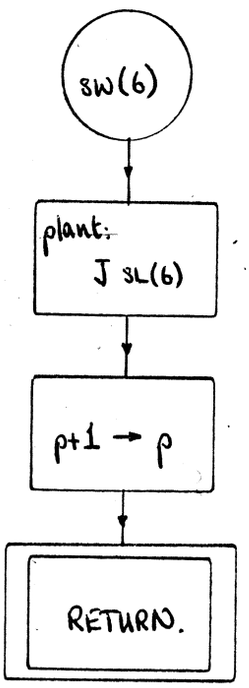

sw(3)
{caption} [TEXT]
plant SET stack address of text. Jump to caption print sr.
RETURN

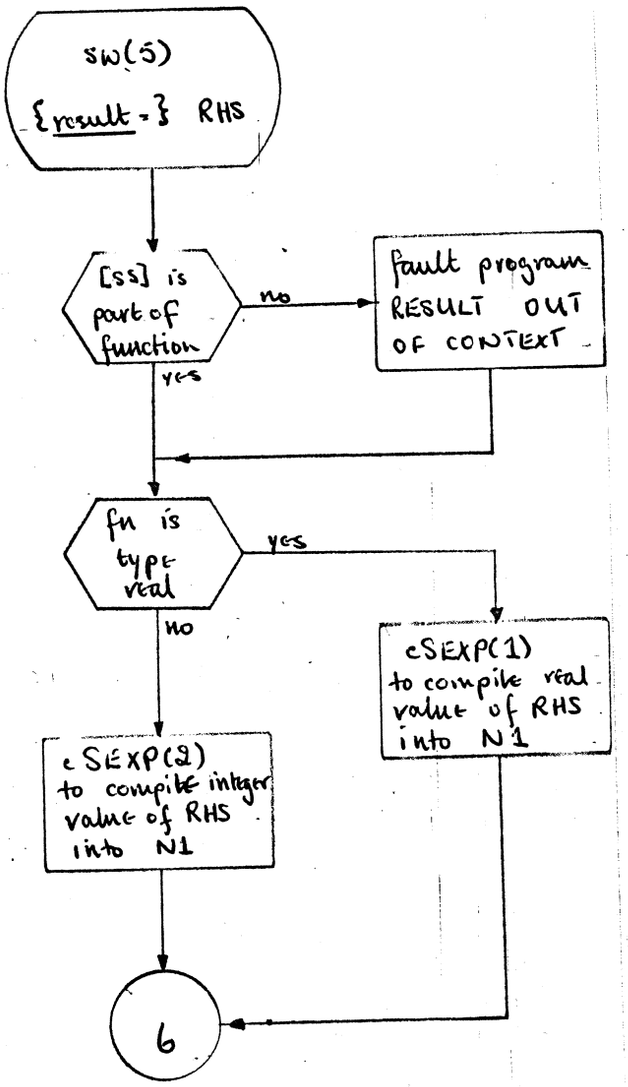

sw(4)
{return}
[SS] is in.c routine or in.perm.
no
yes
6
fault program RETURN OUT OF CONTEXT
plant code reset workspace pointer, jump to return sr
RETURN

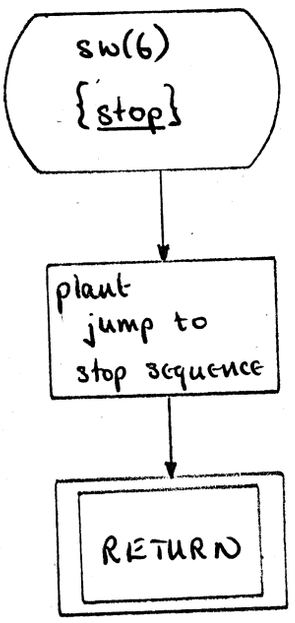
result =


SW(5)
7 ≤flag(level)≤12
NO
fault(31)
yes
p+1 → p
5
flag(level)=7
yes
cSEXP(1)
no
cSEXP(2)
6

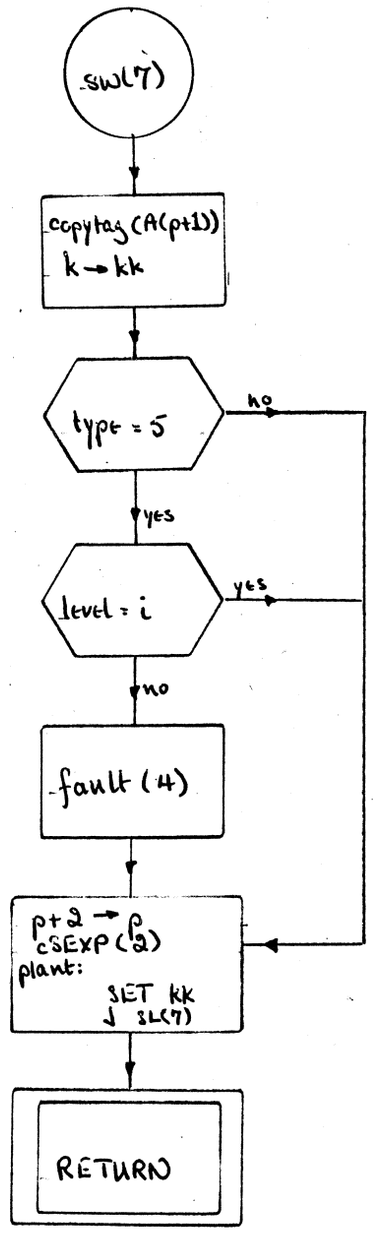
stop


SW(6)
plant:
J SL(6)
p+1 → p
RETURN.

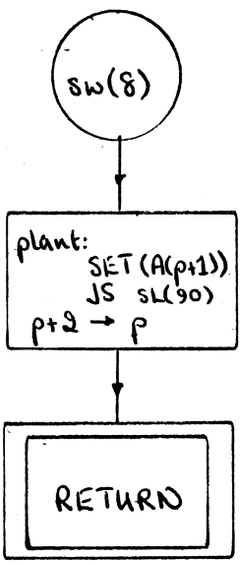

sw(5)
{result -} RHS
[ss] is part of function
no
fault program RESULT OUT OF CONTEXT
yes
fn is type real
yes
no
cSEXP(1) to compile real value of RHS into N1
cSEXP(2) to compile integer value of RHS into N1
6

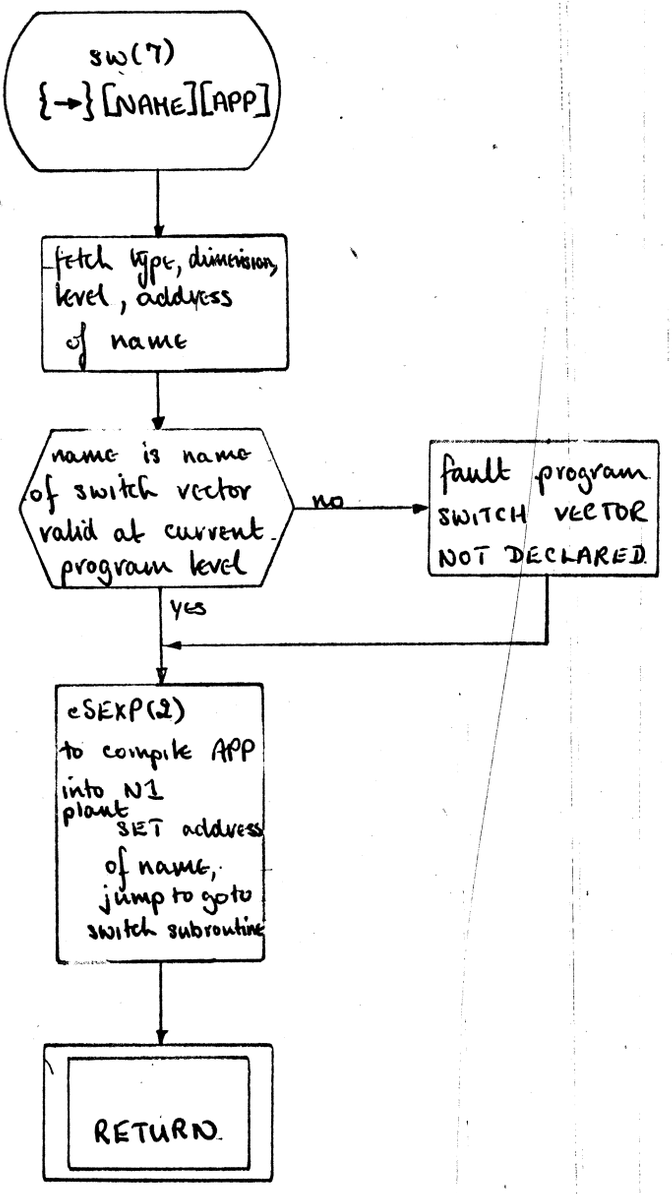

sw(6)
{stop}
plant jump to stop sequence
RETURN

switch



monitor N




sw(7)
{→}[NAME][APP]
fetch type, dimension, level, address of name
name is name of switch vector valid at current program level
no
fault program SWITCH VECTOR NOT DECLARED
yes
eSEXP(2) to compile APP into N1
plant SET address of name, jump to goto switch subroutine
RETURN
sw(8)
{monitor}[N]
plant SET value of [N] jump to monitor.sr
RETURN

Notes on cRSPEC

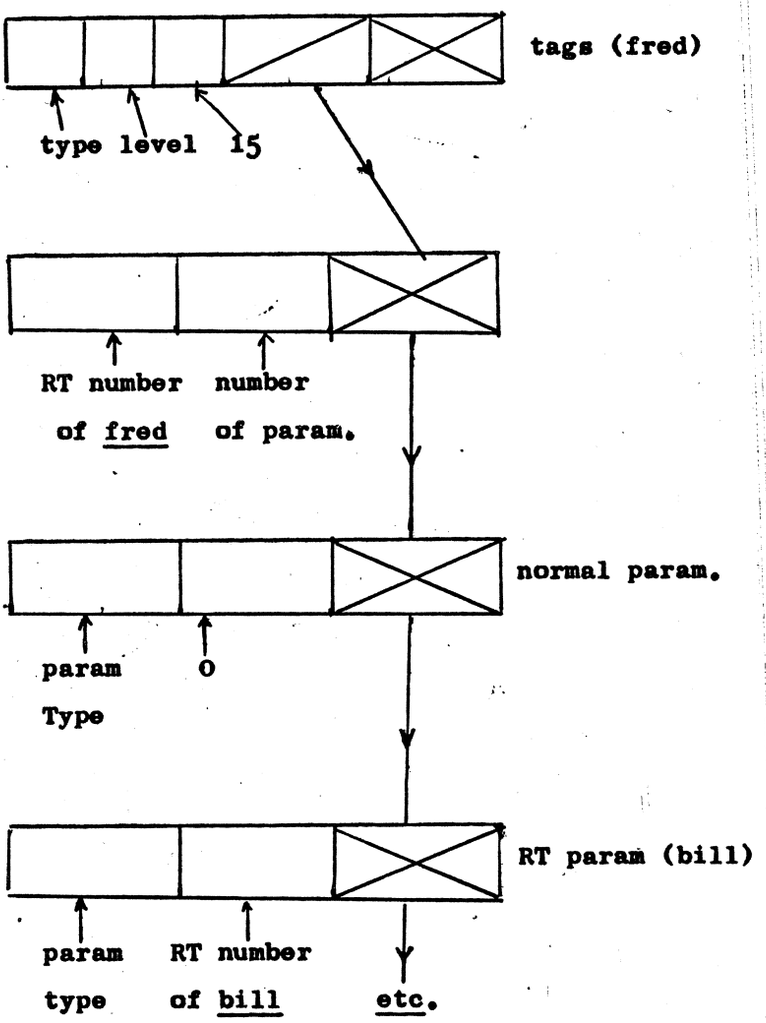
Name handling for routine names and formal parameters is described on the next two pages.

The source statement {spec} [NAME] [FPP] is used to specify the formal parameters of a routine type formal parameter. It is used inside routine descriptions.

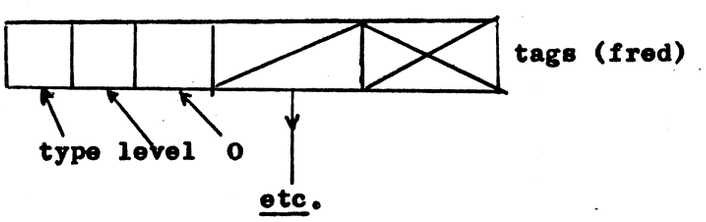
ALL COMPILERS..

ROUTINES

A routine that has been specified has:-



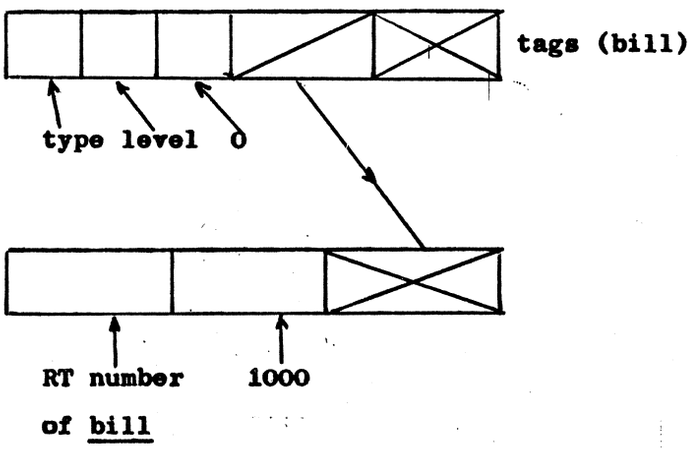
A routine that has been specified and described has :-



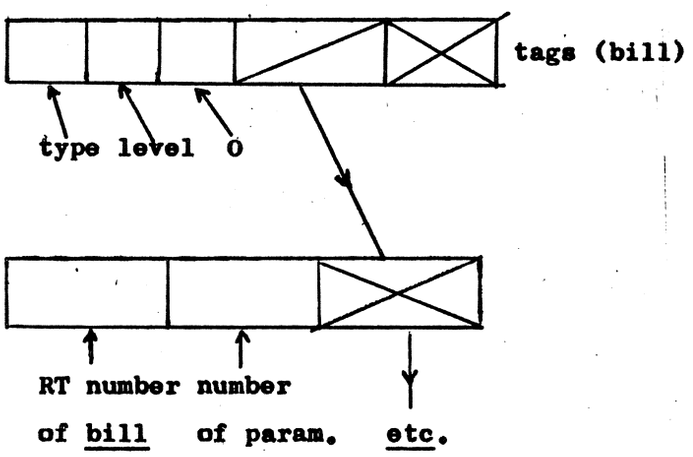
and the rest as before.

ALL COMPILERS.

However, for each of its RT parameters, there now exists:-



When one of these RT parameters is subsequently specified, it has:-



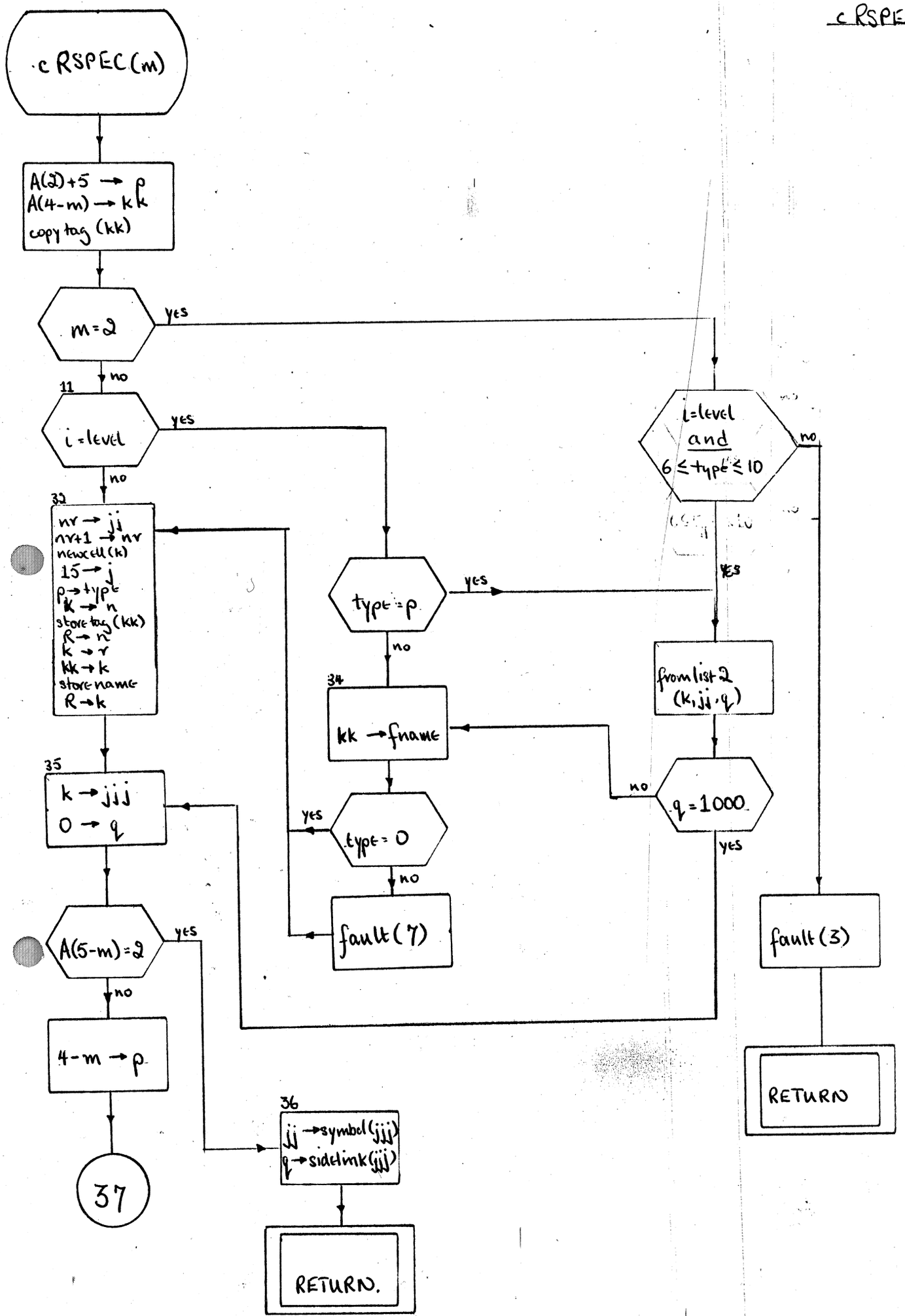
exactly like a routine which has been specified and described.

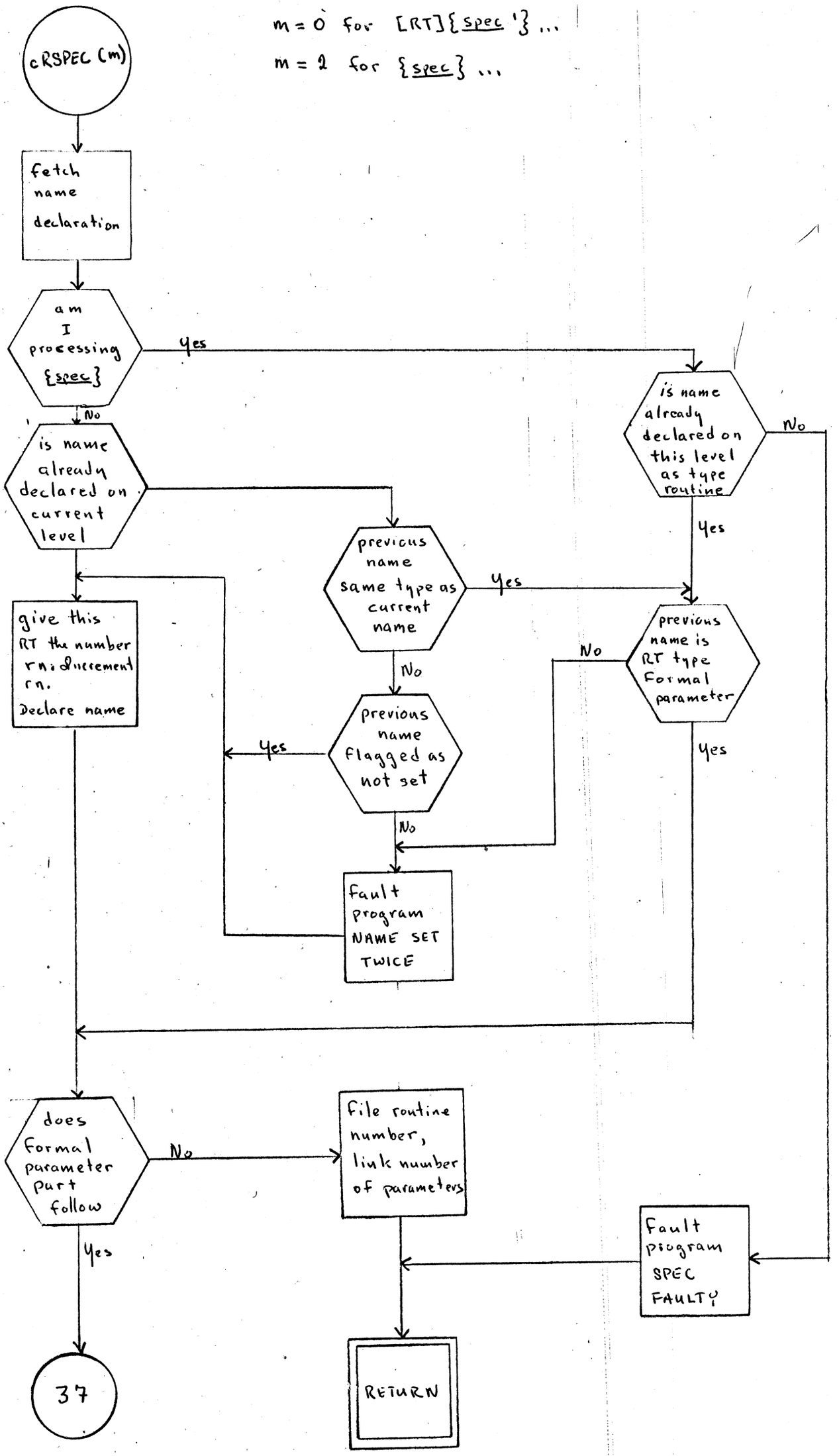
At run time, for a RT parameter, set RA(formal) = RA (actual)

ILLY RUE FOR READY UNITIALISED

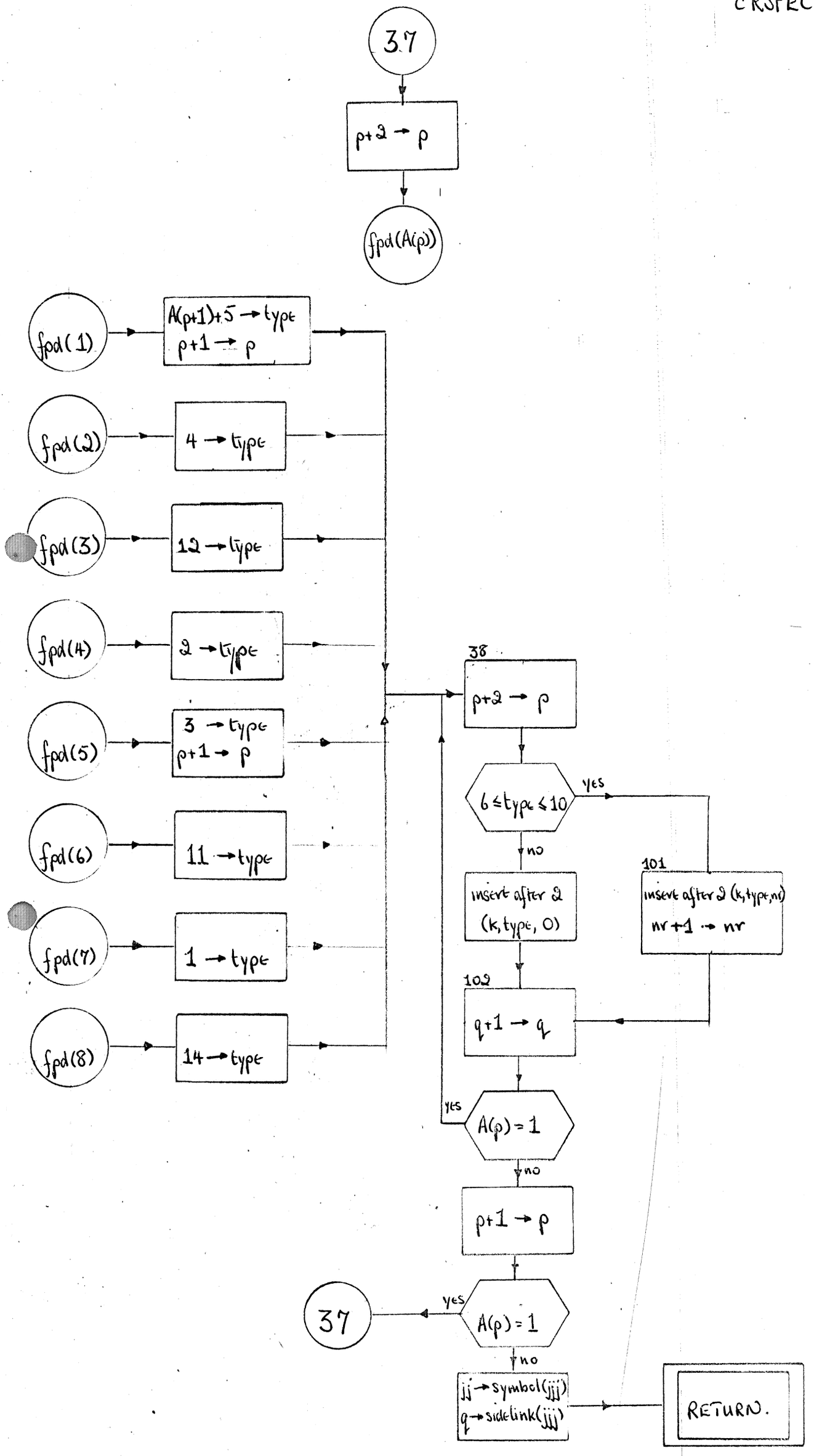
Phrase structure for simple name is :-

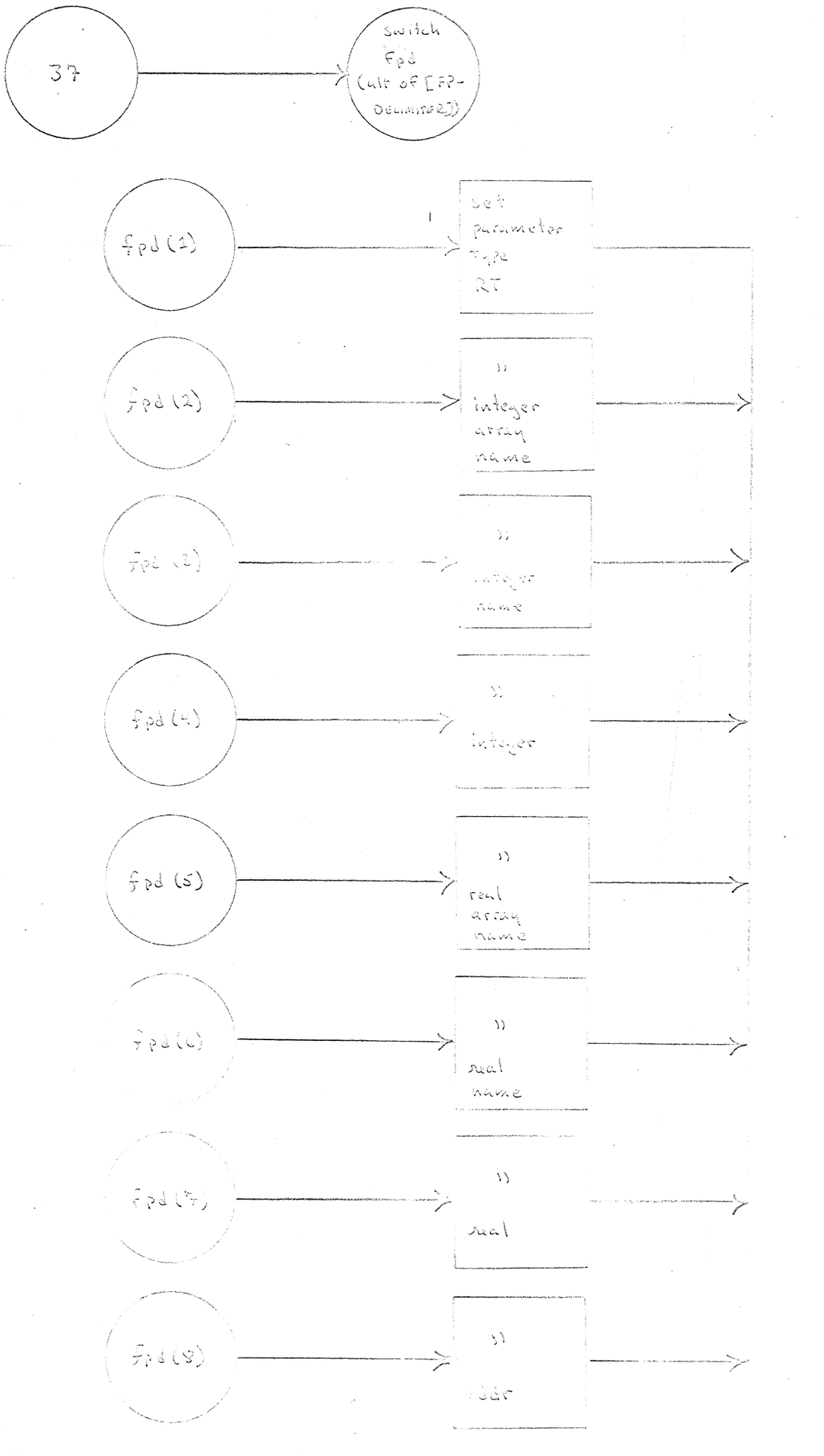
[±'] [ø] [EXPR] [OPERAND] [NAME] [APP] [ø] [,]

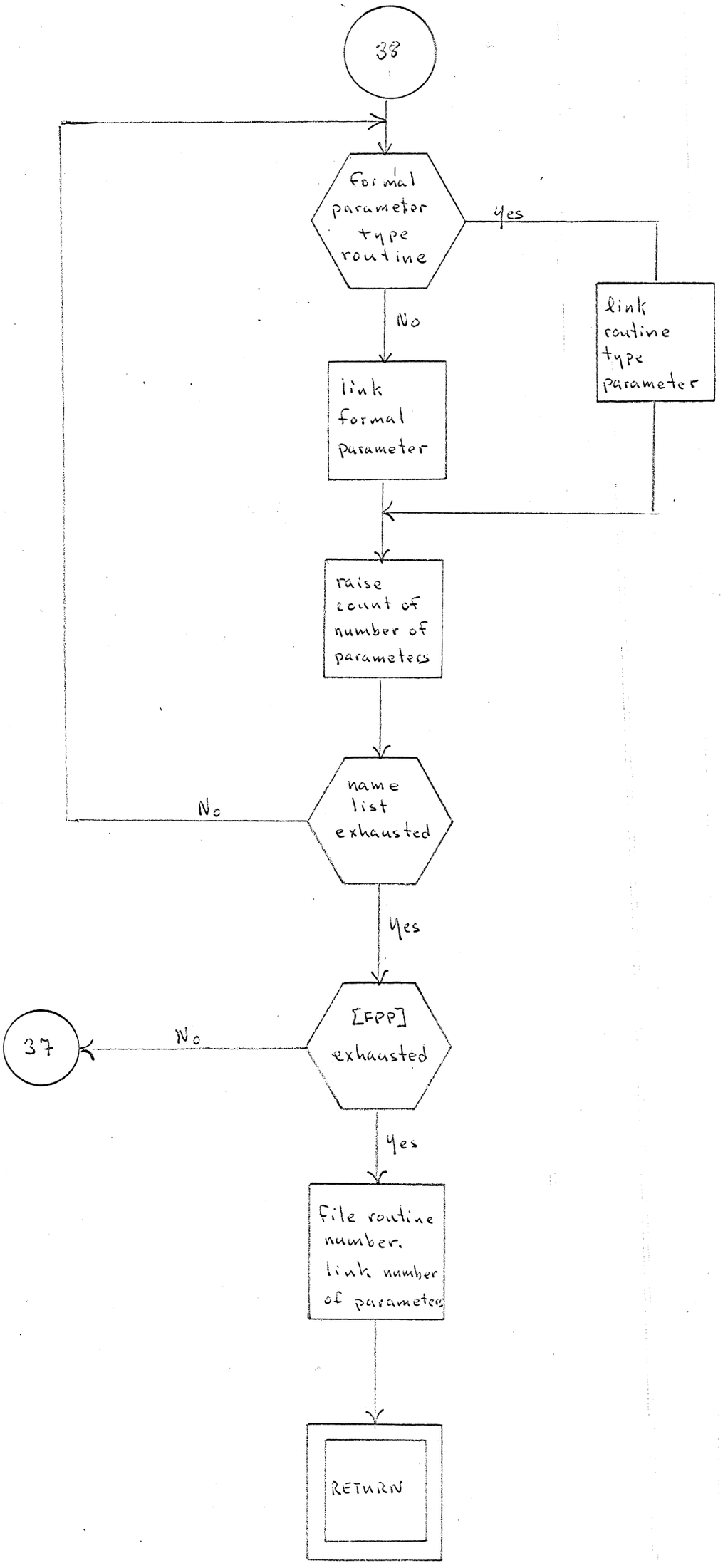

cRSPEC(m)
A(2)+5 → p
A(4-m) → kk
copy tag (kk)
m=2
yes
no
11
i=level
yes
no
32
nr → jj
nr+1 → nr
newcell(k)
15 → j
p → type
kk → n
store tag (kk)
R → n'
k → r
kk → k
store name
R → k
35
k → jjj
0 → q
A(5-m)=2
yes
no
4-m → p
37
type=p
yes
no
34
kk → fname
type=0
yes
no
fault(7)
i=level
and
6 ≤ type ≤ 10
no
yes
fromlist 2
(k, jj, q)
q=1000
no
yes
fault(3)
RETURN
36
jj → symbol(jjj)
p → sidelink(jjj)
RETURN.


m = 0 for [RT]{spec}...
m = 1 for {spec}...
cRSPEC (m)
fetch name declaration
am I processing {spec}
Yes
No
is name already declared on current level
is name already declared on this level as type routine
No
Yes
previous name same type as current name
Yes
No
previous name is RT type Formal parameter
No
Yes
previous name flagged as not set
Yes
No
give this RT the number rn; increment rn. Declare name
Fault program NAME SET TWICE
does Formal parameter part follow
No
Yes
File routine number, link number of parameters
Fault program SPEC FAULTY
37
RETURN

cRSPEC.


37
p+2 → p
fpd(A(p))
fpd(1)
A(p+1)+5 → type
p+1 → p
fpd(2)
4 → type
fpd(3)
12 → type
fpd(4)
2 → type
fpd(5)
3 → type
p+1 → p
fpd(6)
11 → type
fpd(7)
1 → type
fpd(8)
14 → type
38
p+2 → p
6 ≤ type ≤ 10
yes
no
insert after 2
(k, type, 0)
101
insert after 2 (k, type, nr)
nr+1 → nr
102
q+1 → q
yes
A(p) = 1
no
p+1 → p
37
yes
A(p) = 1
no
jj → symbol(jjj)
q → sidelink(jjj)
RETURN.


37
switch fpd (alt of [FP-DELIMITER])
fpd (1)
set parameter type RT
fpd (2)
" integer array name
fpd (3)
" integer name
fpd (4)
" integer
fpd (5)
" real array name
fpd (6)
" real name
fpd (7)
" real
fpd (8)
" char
38


38
formal parameter type routine
yes
No
link routine type parameter
link formal parameter
raise count of number of parameters
name list exhausted
No
Yes
[FPP] exhausted
37
No
Yes
File routine number. link number of parameters
RETURN

CODE DUMPING AND PROGRAM INITIALIZATION

Code dumping is accomplished by six routines

pJ	for three syllable jumps
pQ	for instructions using Q-stores (and for LINK,=LINK,JrCqNES)
pN	for one syllable nest instructions
pSH	for two syllable nest instructions
pMS	for directly addressed main store instructions.
pSET	for SET instructions

A list relating the routines and the various possible values of their parameters to the code planted may be found at the end of this section along with a list of the bit patterns for each machine code instruction.

The five routines arrange their parameters into a form suitable for dumping and conclude with a JS 50P: to where the actual dumping is done.

Code dumping consists in filling core buffers with code and writing then onto magnetic tape when they are full. There are three such buffers, each 100computer words in length. One is filled with program instructions, one in the SET constants, and one with jump addresses. (The reason for this separation is that SET constants and jump addresses are not always known when this instruction parts are planted. When their values are known, they are planted with the program code. Each buffer is filled and written out separately, and when the program is loaded, they are OR ed together.

Instructions are planted in the program buffer syllable by syllable, with binary zeros in the bits to be filled in with jump address parts and SET constants. SET constants and jump addresses are planted in the SET buffer along with the actual address of the instructions with which they are associated. When these latter are loaded, they are shifted into the correct syllable position(s) and OR ed with their absolute location(s). (For details see the annotated code of the entry sequence.)

The compiler subroutine 50P fills code into the buffer and dumps the buffer onto tape as they fill up. Buffers are dumped by

routine splash (integer warning, integer name from).

integer warning several s identifies the type of block to be written out

= 0 for program block

= 1 for label block

= 2 for set block

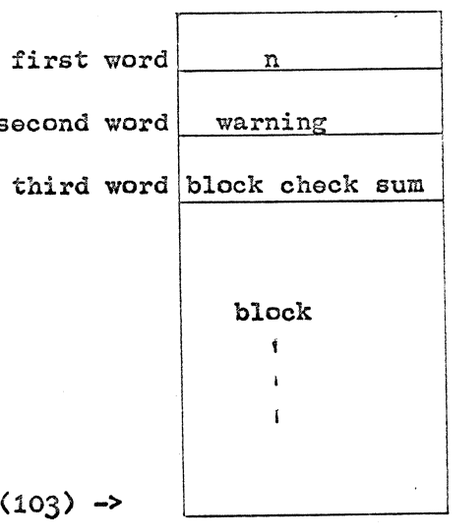
= 3 for stack block

= 4 for RA (routine address) block

= 7 for program check sum block

integer name from is the starting address of the block

Each block dumped is 103 words long*, with the following format-



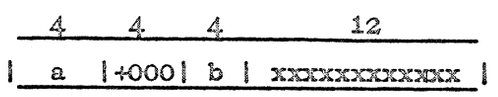
number of blocks previously written out +1

(see above)

*Not all 103 words are necessarily useful. c.f. ''dump stack and routines.''

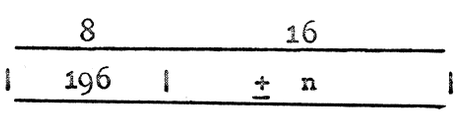
routine dump stack and routines is called when statement end of program has been compiled. It calls <splash> to dump any code remaining in the program, set, or jump label blocks, and then dumps the stack <ST>, routine addresses <RA> and program checksum block. Finally it initialises the entry sequence (at 61P in perm) and enters it. A flowchart and annotated code for the entry sequence is included here.

pJ (integer a,s,b)

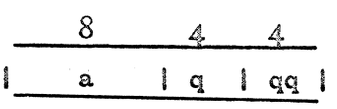


where s = 000 + xxxxxxxxxxx

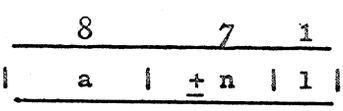
pSET (integer n)



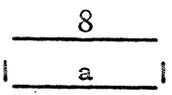
pQ (integer a,q,qq)



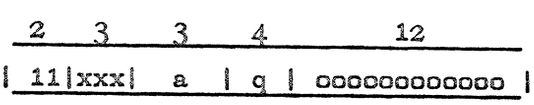
pSH (integer a,n)



pN (integer a)



pMS (integer a,m,q)



where m = xxxooooooooooo

pJ forms

Parameters are (c,a,b), and a holds the address r to be jumped to.

pJ(8,a,1)	Jr ≠
pJ(8,a,2)	Jr ≥ Z
pJ(8,a,4)	Jr ≤ Z
pJ(8,a,6)	Jr ≠ Z
pJ(8,a,8)	Jr NV
pJ(8,0,9)	OUT
pJ(8,a,10)	Jr NEN
pJ(8,a,11)	Jr
pJ(8,a,12)	Jr NEJ
pJ(8,a,13)	JSr
pJ(8,a,14)	Jr NTR
pJ(8,k,15)	EXIT n
	where k=8192 * parity (n) + intpt (n/2) +8192
pJ(8,k,15)	EXIT
	where k=16384

pJ(9,a,1)	Jr=
pJ(9,a,2)	Jr < Z
pJ(9,a,4)	Jr > Z
pJ(9,a,6)	Jr = Z
pJ(9,a,8)	Jr V
pJ(9,a,10)	Jr EN
pJ(9,a,12)	Jr EJ
pJ(9,a,14)	Jr TR
pJ(10,a,q)	JrCq Z
pJ(11,a,q)	JrCq NZ

P M S forms

pMS(0,n,q)	En Mq
pMS(1,n,q)	=En Mq
pMS(2,n,q)	En Mq Q
pMS(3,n,q)	=En Mq Q

p SET forms

pSET(n)	SET n	(n decimal)

pN(1)	VR
pN(2)	=TR
pN(3)	BITS
pN(4)	XF
pN(5)	xDF
pN(7)	x +F
pN(8)	NEGD
pN(9)	OR
pN(10)	PERM
pN(11)	TOB
pN(12)	ROUNDH
pN(13)	NEV
pN(14)	ROUND
pN(15)	DUMMY
pN(16)	ROUNDF
pN(17)	ROUNDF
pN(18)	-DF
pN(19)	+DF
pN(20)	FLOAT
pN(21)	FLOATD
pN(22)	ABS
pN(23)	NEG
pN(24)	ABSF
pN(25)	NEGF
pN(26)	MAX
pN(27)	NOT
pN(28)	XD
pN(29)	X
pN(30)	-
pN(31)	SIGN
pN(33)	ZERO
pN(34)	DUP
pN(35)	DUPD
pN(36)	+I

pN(37)	FIX
pN(39)	STR
pN(40)	CONT
pN(41)	REVD
pN(42)	ERASE
pN(43)	-D
pN(44)	AND
pN(45)	+
pN(47)	+D
pN(48)	÷
pN(49)	÷D
pN(50)	÷F
pN(51)	÷DF
pN(52)	÷R
pN(53)	REV
pN(54)	CAB
pN(55)	FRB
pN(56)	STAND
pN(57)	NEGDF
pN(58)	MAXF
pN(60)	+F
pN(61)	-F
pN(63)	SIGNF

pSH(113,n)	SHA ±n
pSH(114,n)	SHAD ±n
pSH(115,n)	X ÷ ±n
pSH(116,n)	SHL ±n
pSH(118,n)	SHLD ±n
pSH(119,n)	SHC ±n

pQ(115,0,0)	x+
pQ(116,q,0)	SHL Cq
pQ(118,q,0)	SHLD Cq
pQ(119,q,0)	SHC Cq
pQ(120,q,2)	=Mq
pQ(120,q,3)	=RMq
pQ(120,q,4)	=Iq
pQ(120,q,5)	=RIq
pQ(120,q,8)	=Cq
pQ(120,q,9)	=RCq
pQ(120,q,14)	=Qq
pQ(121,q,2)	Mq
pQ(121,q,4)	Iq
pQ(121,q,8)	Cq
pQ(121,q,14)	Qq
pQ(122,q,2)	=+Mq
pQ(122,q,4)	=+Iq
pQ(122,q,8)	=+Cq
pQ(122,q,14)	=+Qq
pQ(123,0,0)	LINK
pQ(122,0,0)	=LINK
pQ(127,q,0)	JrCq NZS (r is dummy)

pQ(64,q,k)	Mk Mq
pQ(65,q,k)	=Mk Mq
pQ(66,q,k)	Mk Mq Q
pQ(67,q,k)	=Mk Mq Q
pQ(68,q,k)	Mk Mq H
pQ(69,q,k)	=Mk Mq H
pQ(70,q,k)	Mk Mq QH
pQ(71,q,k)	=Mk Mq QH
pQ(72,q,k)	Mk Mq N
pQ(73,q,k)	=Mk Mq N
pQ(74,q,k)	Mk Mq QN
pQ(75,q,k)	=Mk Mq QN
pQ(76,q,k)	Mk Mq HN
pQ(77,q,k)	=Mk Mq HN
pQ(78,q,k)	Mk Mq QHN
pQ(79,q,k)	=Mk Mq QHN

pQ(96,q,0)	M+Iq
pQ(97,q,0)	M-Iq
pQ(98,q,0)	N Cq
pQ(99,q,0)	D Cq
pQ(100,q,0)	Iq + 1
pQ(101,q,0)	Iq + 2
pQ(102,q,0)	Iq - 1
pQ(103,q,0)	Iq - 2
pQ(105,k,q)	Mk TO Qq
pQ(106,k,q)	Ik TO Qq
pQ(107,k,q)	IMk TO Qq
pQ(108,k,q)	Ck TO Qq
pQ(109,k,q)	CMk TO Qq
pQ(110,k,q)	CIk TO Qq
pQ(111,k,q)	Qk TO Qq
pQ(113,q,0)	SHA Cq
pQ(114,q,0)	SHAD Cq
pQ(115,q,0)	x+ Cq

MACHINE ORDER PLANTING ROUTINE CALLS

ONE SYLLABLE TYPES

ROUNDHF	pN(17)
ROUNDF	pN(16)
ROUNDH	pN(12)
ROUND	pN(14)
FLOATD	pN(21)
FLOAT	pN(20)
ERASE	pN(42)
REVD	pN(41)
REV	pN(53)
ZERO	pN(33)
DUPD	pN(35)
DUP	pN(34)
NEGDF	pN(57)
NEGF	pN(25)
NEGD	pN(8)
NEG	pN(23)
ABSF	pN(24)
ABS	pN(22)
MAXF	pN(58)
MAX	pN(26)
SIGNF	pN(63)
SIGN	pN(31)
CAB	pN(54)
FRB	pN(55)
STAND	pN(56)
÷ DF	pN(51)
÷ D	pN(49)
÷ F	pN(50)
÷ R	pN(52)
÷ I	pN(36)
÷	pN(48)
+ DF	pN(19)
+ D	pN(47)
+ F	pN(60)
+	pN(46)
- DF	pN(18)
- D	pN(43)
- F	pN(61)
-	pN(30)

x DF	pN(5)
x D	pN(28)
x F	pN(4)
x +F	pN(7)
x	pN(29)
= TR	pN(2)
BITS	pN(3)
DUMMY	pN(15)
PERM	pN(10)
TOB	pN(11)
OR	pN(9)
VR	pN(1)
NEV	pN(13)
NOT	pN(27)
FIX	pN(37)
STR	pN(39)
CONT	pN(40)
AND	pN(44)

TWO SYLLABLE TYPES

MkMqQHN	pQ(78,q,k)	)
MkMqQH	pQ(70,q,k)	)
MkMqQN	pQ(74,q,k)	)
MkMqQ	pQ(66,q,k)	) for stores increase
MkMqHN	pQ(76,q,k)	) integer by 1
MkMqH	pQ(68,q,k)	)
MkMqN	pQ(72,q,k)	)
MkMq	pQ(64,q,k)	)
SHA ± n	pSH(113,n)	
SHAD ± n	pSH(114,n)	
SHL ± n	pSH(116,n)	
SHLD ± n	pSH(118,n)	
SHC ± n	pSH(119,n)	
SHA Cq	pQ(113,q,0)	
SHAD Cq	pQ(114,q,0)	
SHL Cq	pQ(116,q,0)	
SHLD Cq	pQ(118,q,0)	
SHC Cq	pQ(119,q,0)	
= LINK	pQ(124,0,0)	
= Qq	pQ(120,q,14)	
= Cq	pQ(120,q,8)	
= Iq	pQ(120,q,4)	
= Mq	pQ(120,q,2)	
= RCq	pQ(120,q,9)	
= RIq	pQ(120,q,5)	
= RMq	pQ(120,q,3)	
= +Qq	pQ(122,q,14)	
= +Cq	pQ(122,q,8)	
= +Iq	pQ(122,q,4)	
= +Mq	pQ(122,q,2)	
Qk TO Qq	pQ(111,k,q)	
Ck TO Qq	pQ(108,k,q)	
Ik TO Qq	pQ(106,k,q)	
Mk TO Qq	pQ(105,k,q)	
IMk TO Qq	pQ(107,k,q)	
CMk TO Qq	pQ(109,k,q)	
CIk TO Qq	pQ(110,k,q)	

			a
Iq = ± 1)	pQ(a,q,0)	+1	100
Iq = ± 2)		+2	102
		-1	101
		-2	103

Qq	pQ(121,q,14)
Cq	pQ(121,q,8)
Iq	pQ(121,q,4)
Mq	pQ(121,q,2)
NCq	pQ(98,q,0)
DCq	pQ(99,q,0)
LINK	pQ(123,0,0)
M + Iq	pQ(96,q,0)
M - Iq	pQ(97,q,0)
x + ± n	pSH(115,n)
x + Cq	pQ(115,q,0)
x +	pQ(115,0,0)

THREE SYLLABLE TYPES - JUMPS

JrCqNZ	pJ(11,a,q)	a = addr. ref. (r)
JrCqZ	pJ(10,a,q)	in standard form.
Jr = Z	pJ(9,a,6)	
Jr =	pJ(9,a,1)	
Jr ≠ Z	pJ(8,a,6)	
Jr ≠	pJ(8,a,1)	
Jr > Z	pJ(9,a,4)	
Jr ≥ Z	pJ(8,a,2)	
Jr < Z	pJ(9,a,2)	
Jr ≤ Z	pJ(8,a,4)	
JrV	pJ(9,a,8)	
JrNV	pJ(8,a,8)	
JrEN	pJ(9,a,10)	
JrNEN	pJ(8,a,10)	
JrEJ	pJ(9,a,12)	
JrNEJ	pJ(8,a,12)	
JrTR	pJ(9,a,14)	
JrNTR	pJ(8,a,14)	
Jr	pJ(8,a,11)	
JSr	pJ(8,a,13)	
EXIT n	pJ(8,b,15)	where b = 8192 parity (n) + intpt(n/2) + 8192
OUT	pJ(8,0,9)	
EXIT	pJ(8,b,15)	where b = 16384

DIRECTLY ADDRESSED MAIN STORE TYPES

THREE SYLLABLE

EnMq	pMS(0,n,q)
EnMqQ	pMS(2,n,q)
= EnMq	pMS(1,n,q)
= EnMqQ	pMS(3,n,q)

MISCELLANEOUS

SET ± n	pSET(n)	)
SET Bt	pSET(decimal form of octal no. t)	) 3 - SYLLABLE
JrCqNZS	pQ(127,q,0) [r is a dummy]	) 2 - SYLLABLE
** NAME (APP)	fetch NAME (APP) -> NS	)
** = NAME (APP)	store NAME (APP)	) MULTI - SYLLABLE
** α NAME (APP)	fetch addr. of NAME (APP)	)

---

* ; by itself means start new word,
fill up spaces with DUMMY. ... letters.

1. JUMP INSTRUCTIONS

	0 3	4 7	8 11	12 15	16 23
JSr	1 0 0 0	a s s s	1 1 0 1	a a a a	a a a a a a a a
JrCqZ	1 0 1 0		q q q q		
JrCqNZ	1				
Jr	1 0 0 0		1 0 1 1		
JrCqNZS	0 1 1 1	1 1 1 1	q q q q	b b b b	2-SYLLABLE
Jr=	1 0 0 1	a s s s	0 0 0 1	a a a a	a a a a a a a a
Jr≠	0				
Jr < Z	1 0 0 1		0 0 1 0		
Jr ≥ Z	0				
Jr > Z	1 0 0 1		0 1 0 0		
Jr ≤ Z	0				
Jr = Z	1 0 0 1		0 1 1 0		
Jr ≠ Z	0				
JrV	1 0 0 1		1 0 0 0		
JrNV	0				
JrEN	1 0 0 1		1 0 1 0		
JrNEN	0				
JrEJ	1 0 0 1		1 1 0 0		
JrNEJ	0				
JrTR	1 0 0 1		1 1 1 0		
JrNTR	0				
EXIT	1 0 0 0	a o h o	1 1 1 1		
OUT	1 0 0 0	b b b b	1 0 0 1	b b b b	b b b b b b b b

2. Q-STORE OPERATIONS

	0	7	8	15
LINK	0 1 1 1 1	0 1 1	0 0 0 0	b b b b
=LINK	0 1 1 1 1	1 0 0	0 0 0 0	
M+Iq	0 1 1 0 0	0 0 0	q q q q	
M-Iq		0 0 1		
NCq		0 1 0		
DCq		0 1 1		
Iq=1		1 0 0		
Iq=-1		1 0 1		
Iq=2		1 1 0		
Iq=-2		1 1 1		
QqTOQq'	0 1 1 0 1	1 1 1	q q q q	q'q'q'q'
CqTOQq'		1 0 0		
IqTOQq'		0 1 0		
MqTOQq'		0 0 1		
IMqTOQq'		0 1 1		
CMqTOQq'		1 0 1		
CIqTOQq'		1 1 0		
Qq	0 1 1 1 1	0 0 1	q q q q	1 1 1 b
Cq				1 0 0 b
Iq				0 1 0 b
Mq				0 0 1 b
SET	1 1 b b b	1 b 0	<----------16 bit parameter---------->	
=Qq	0 1 1 1 1	0 0 0	q q q q	1 1 1 0
=Cq				1 0 0 0
=Iq				0 1 0 0
=Mq				0 0 1 0
=RCq				1 0 0 1
=RIq				0 1 0 1
=RMq				0 0 1 1
=+Qq		0 1 0		1 1 1 b
=+Cq				1 0 0 b
=+Iq				0 1 0 b
=+Mq				0 0 1 b
SHA±n	0 1 1 1 0	0 0 1	± n n n	n n n 1
SHAD±n		0 1 0		

	0				7	8							15
SHL±n					1 0 0								
SHLD±n					1 1 0								
SHC±n					1 1 1								
x+±n					0 1 1								
SHACq	0	1	1	1	0 0 0 1	q	q	q	q	b	b	b	0
SHADCq					0 1 0								
SHLCq					1 0 0								
SHLDCq					1 1 0								
SHCCq					1 1 1								
x+Cq					0 1 1								

3. MAIN STORE TRANSFERS

	0		3	4		7				12		15							23
YoMq	1	1	a a	a	0	0	0	q	q	q	q	a	a	a	a	a	a	a	a	a	a
YoMqQ					0	1	0														
Mq'Mq	0	1	0	0	0	0	0	0	q	q	q	q	q'	q'	q'	q'					
Mq'MqQ					0	0	1	0													
Mq'MqN					1	0	0	0													
Mq'MqQN					1	0	1	0													
Mq'MqH					0	1	0	0													
Mq'MqQH					0	1	1	0													
Mq'MqHN					1	1	0	0													
Mq'MqQHN					1	1	1	0													

2-SYLLABLE

5. SINGLE SYLLABLE INSTRUCTIONS

with d2=0	0	2				7	with d2=1
	0	0	0	0	0	0	0	
VR						1	ZERO
=TR					1		DUP
BITS					1	1	DUPD
xF				1	0	0	+I
xDF				1	0	1	FIX
x+F				1	1	1	STR
NEGD			1	0	0	0	CONT
OR			1	0	0	1	REVD
PERM			1	0	1	0	ERASE
TOB			1	0	1	1	-D
ROUND H			1	1	0	0	AND
NEV			1	1	0	1	
ROUND			1	1	1	0	+
DUMMY			1	1	1	1	+D
ROUND F		1	0	0	0	0	+
ROUND HF		1	0	0	0	1	+D
-DF		1	0	0	1	0	+F
+DF		1	0	0	1	1	+DF
FLOAT		1	0	1	0	0	+R
FLOAT D		1	0	1	0	1	REV
ABS		1	0	1	1	0	CAB
NEG		1	0	1	1	1	FRB
ABSF		1	1	0	0	0	STAND
NEGF		1	1	0	0	1	NEGDF
MAX		1	1	0	1	0	MAXF
NOT		1	1	0	1	1	
xD		1	1	1	0	0	+F
x		1	1	1	0	1	-F
-		1	1	1	1	0	
SIGN		1	1	1	1	1	SIGNF

4. PERIPHERAL INSTRUCTIONS

	0	7	8	15	
FRQq	0 1 0 1 0	1 0 0	q q q q	0 0 0 u	PIA Qq
FREQq		0 1			PIB Qq
BRQq		1 0			PIE Qq
BREQq		1 1			PIF Qq
RCQq		0 0		1 0 0 u	PIC Qq
RCEQq		0 1			PID Qq
TLOQq		0 0		0 1 0 u	
WQq	0 1 0 1 1	0 0 0		0 0 b u	PO A Qq
WEQq		0 1			PO B Qq
LWQq		0 0		1 0 b u	PO C Qq
LWEQq		0 1		1 0 0 0	POD Qq
GAPQq		0 0		1 1 b u	POE Qq
FSKQq	0 1 0 1 1	1 0 b		0 b 0 u	PΠA Qq
BSKQq		1 b			PME Qq
RWDQq				1 b 0 u	PΠD Qq
INTQq		0 b		0 b 1 u	
METQq	0 1 0 1 0	0 1 0		0 0 0 u	PM F Qq
PARQq		0 1			
BTQq		0 0		1 0 0 u	PΠB Qq
LBQq		0 0		0 1 0 u	PΠC Qq
BUSYQq				0 0 1 u	
MANUAL Qq	0 1 0 1 0	0 0 0	q q q q	0 0 0 1	
WIPE Qq	0 1 0 1 1	0 0 0	q q q q	0 1 0 0	PO F Qq
DIRECTOR ONLY CTQq	0 1 0 1 0	0 0 0	q q q q	0 0 0 0	DIRECTOR
DIRECTOR ONLY CLOQq	0 1 0 1 0	1 0 0	q q q q	0 0 1 0	ΠODE ONLY

Jump Instructions:-

Notes:-

(i) Positions d4, d12 - d23 FOrm the (13 bit) relative core address of the word containing the first syllable of the object instruction. d4 is the m. s. bit, d12 next, down to d23.

(ii) JSr stores its own address in the most accessible cell of the (subroutine) jump nesting store.

(iii) EXIT acts as [illegible] instruction. Any number of half words (up to [illegible],303) may be added to the jump address stored in the JNS. In this instruction d4, 12 - 23 form the integral number of words to be added, a half word is also added if d6 = 0. There is no check to ensure that the final sum is less than 8,192.

Q-store Operations (continued)

Note:-

The instruction SETn is a three syllable instruction; the 16-bit parameter is d8 - d23 with d8 acting as the sign digit.

Main store Transfers

Direct Stores	S38	Same as Direct fetches except d7 = 1.
Indirect Stores	S39	Same as Indirect fetches except d7 = 1.

Notes:- d2, d3, d4, d12 - 23 form the (15-bit) relative core address.

d2 is the most significant bit.

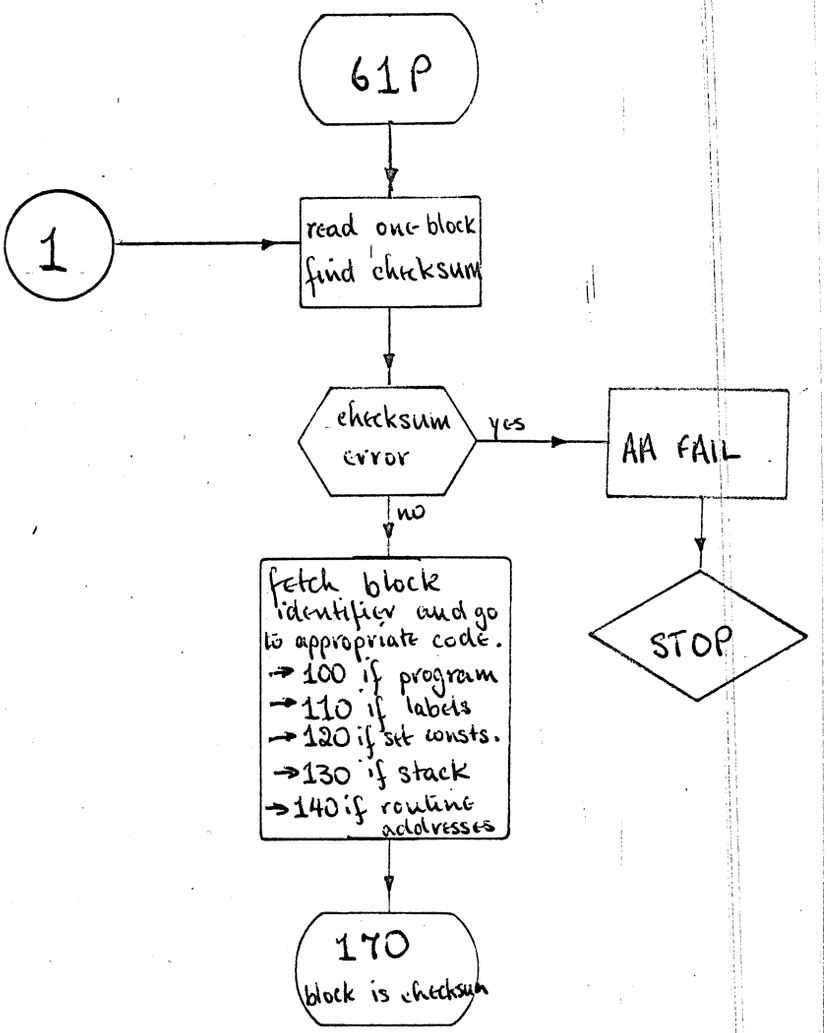
Peripheral Instructions

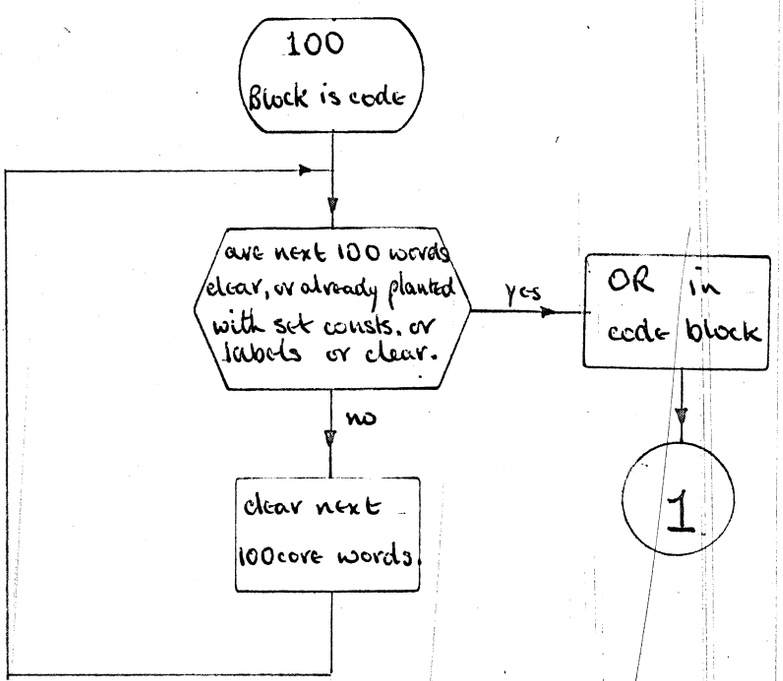
Notes:-

(i) In all the above instructions d15 should be zero. If d15 = 1 then the selected peripheral device will be set unready when the specified transfer has been completed.

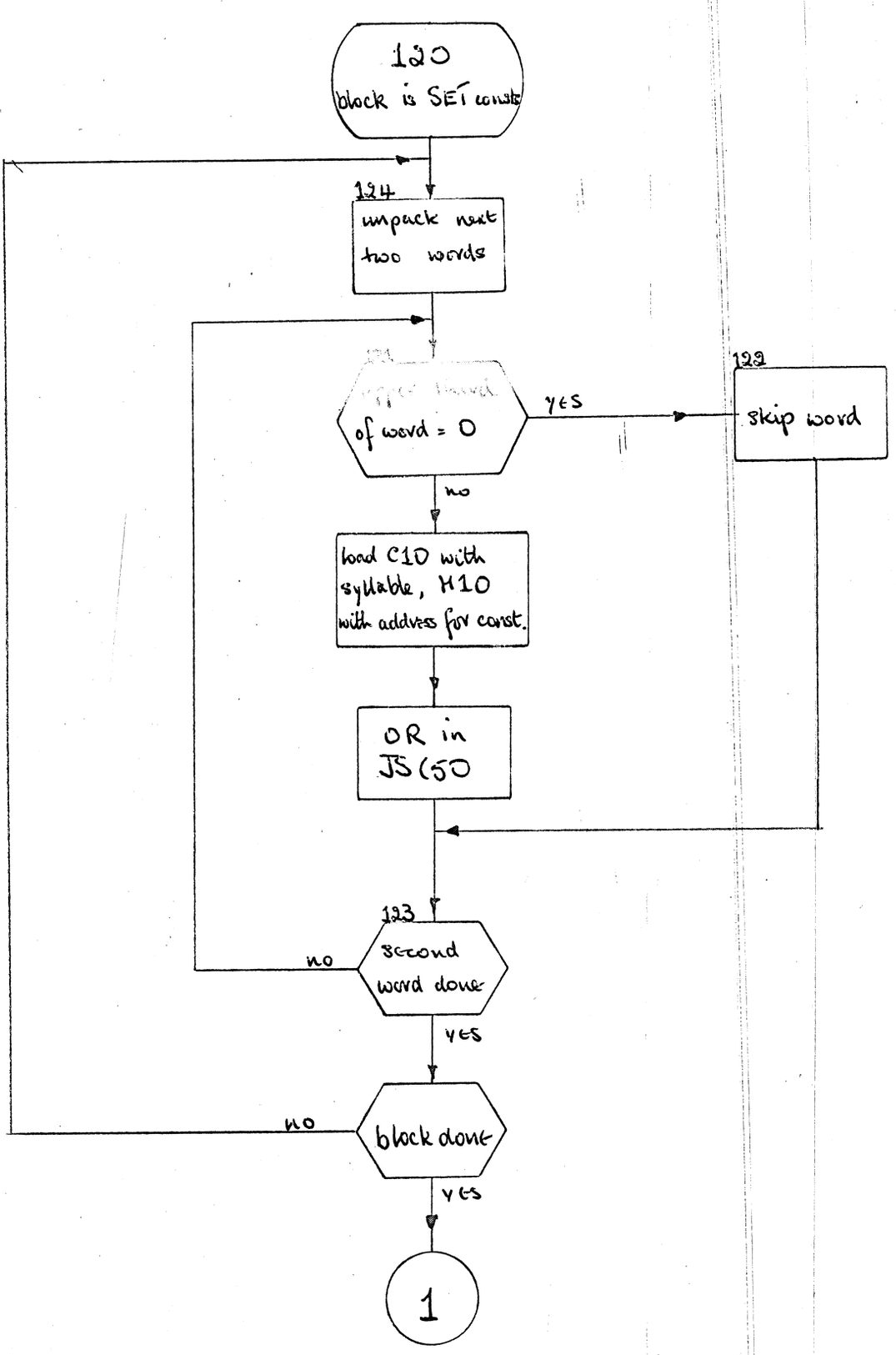
(ii) Not all the above instructions are applicable to all peripheral devices. Thus PLWQq is not allowed. For a list of allowable instructions see the programming manual.

Entry Sequen

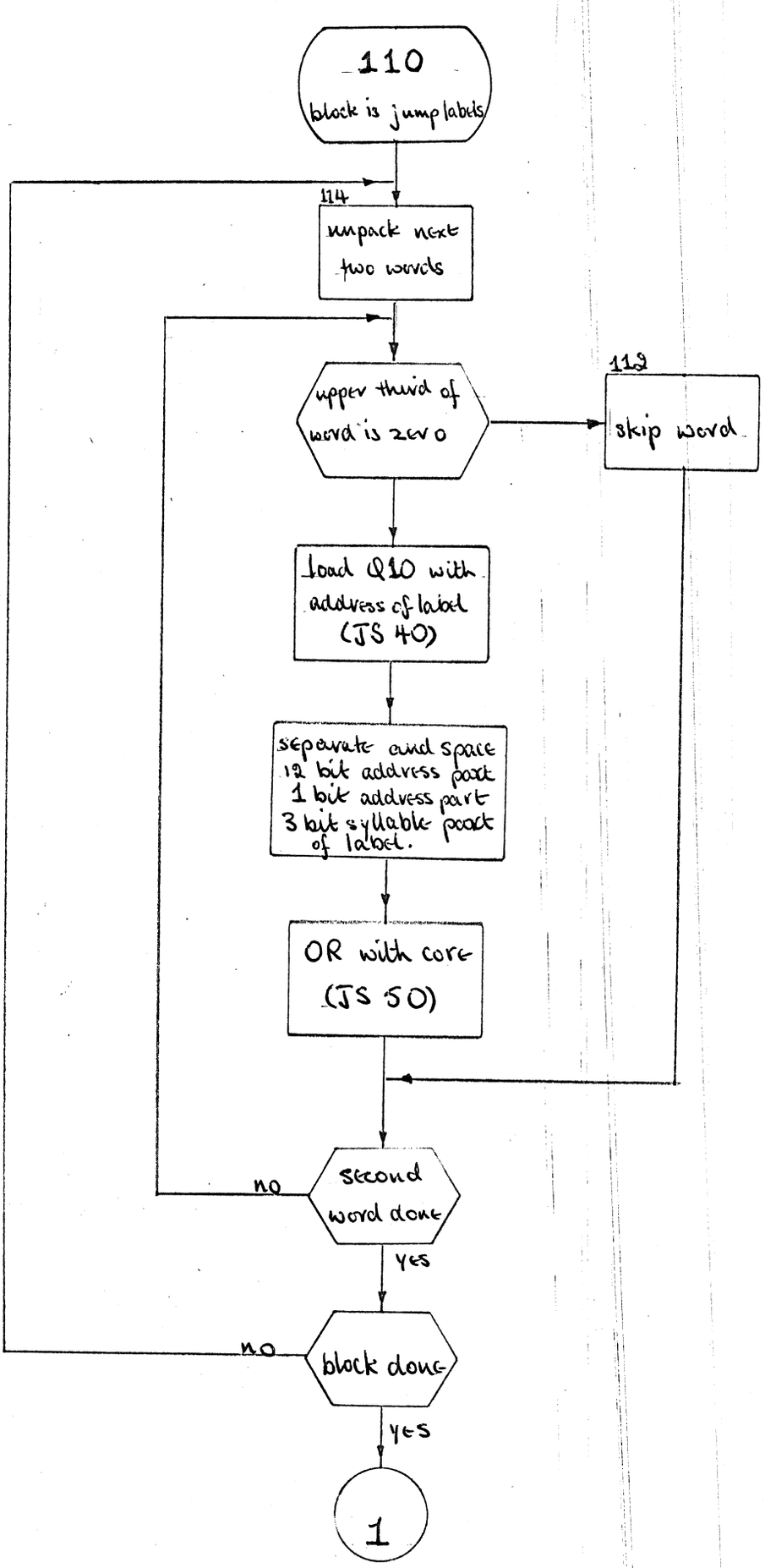




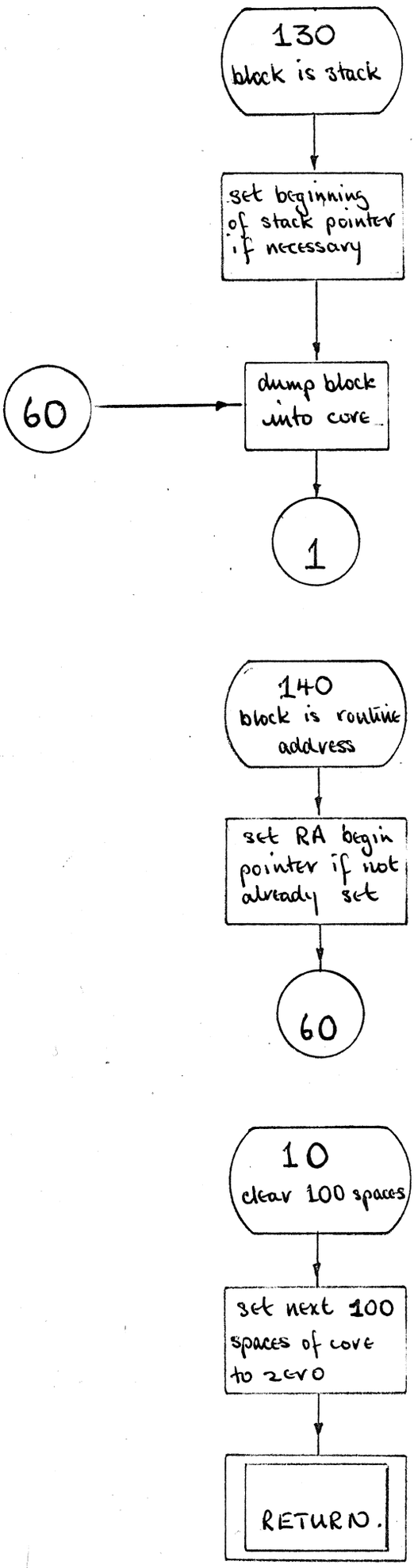
Entry Sequ

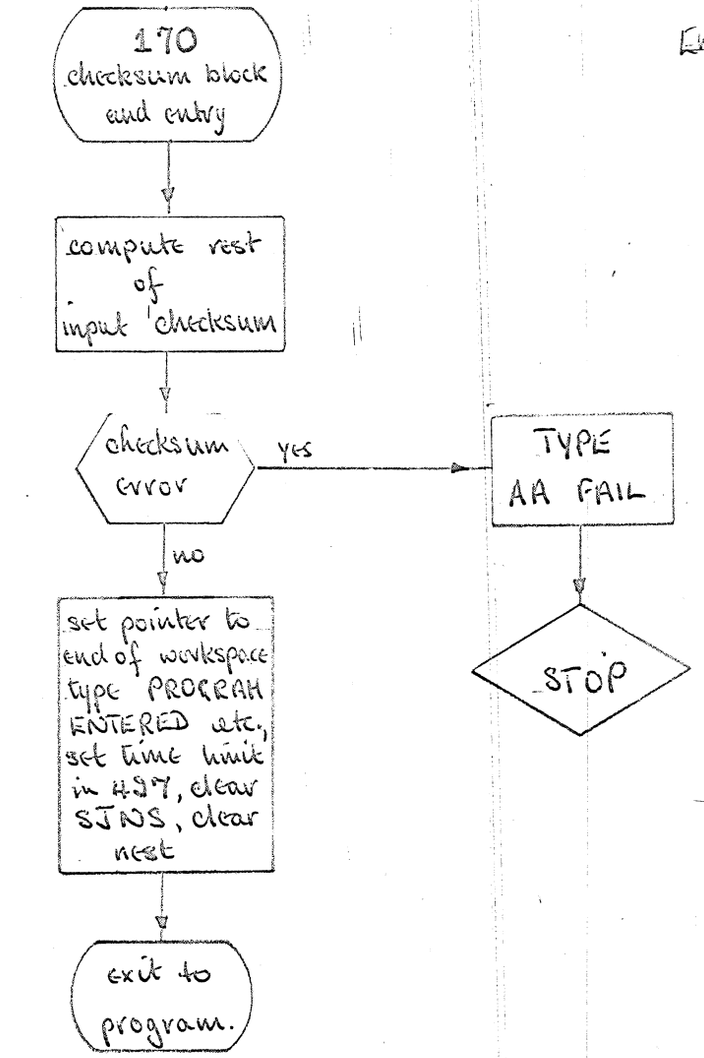

120
block is SET counts
124
unpack next two words
of word = 0
yes
123
skip word
no
load C10 with syllable, H10 with address for const.
OR in JS(50
123
second word done
no
yes
block done
no
yes
1

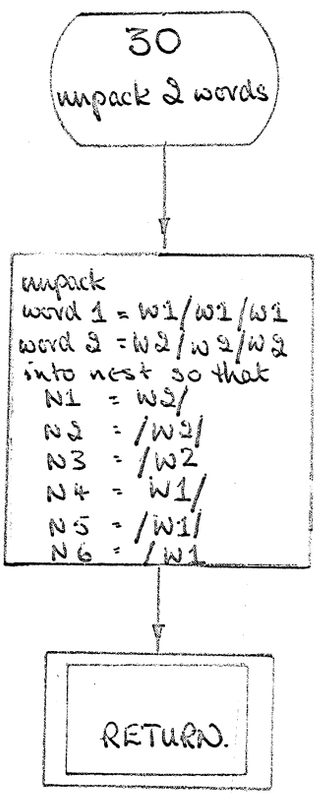
Entry Sequence



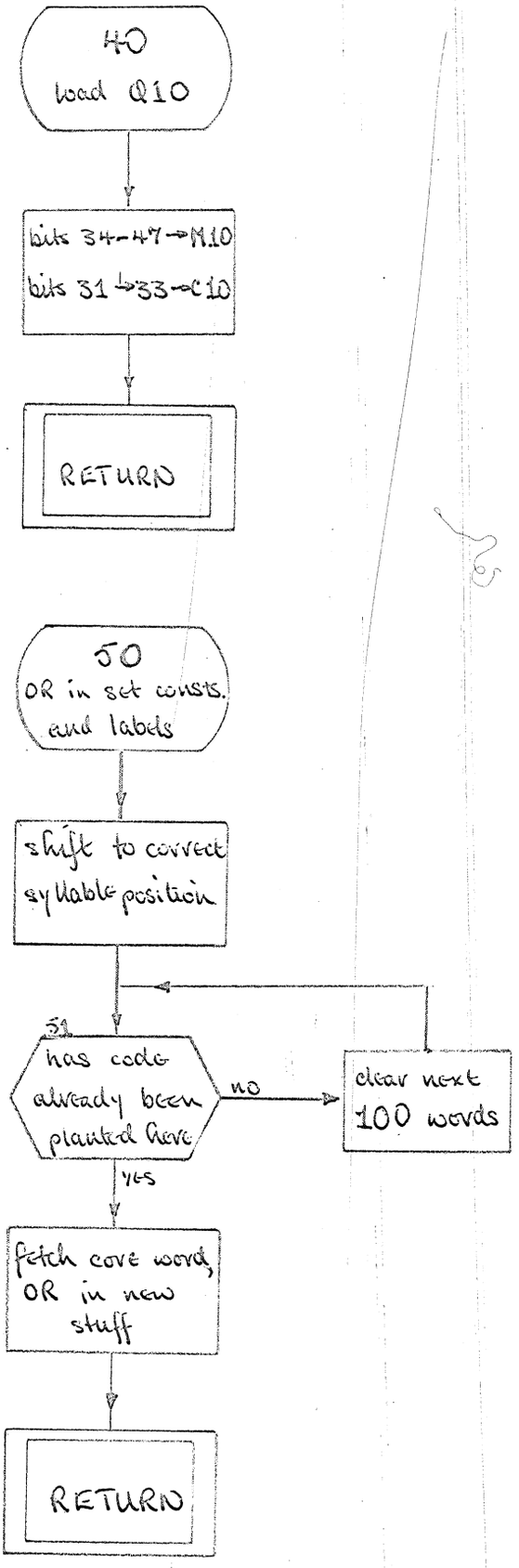
Entry Sequence




170
checksum block
and entry
compute rest
of
input checksum
checksum
error
yes
TYPE
AA FAIL
STOP
no
set pointer to
end of workspace
type PROGRAM
ENTERED etc,
set time limit
in 497, clear
SINS, clear
nest
exit to
program.


30
unpack 2 words
unpack
word 1 = W1/W1/W1
word 2 = W2/W2/W2
into nest so that
N1 = W2/
N2 = /W2/
N3 = /W2
N4 = W1/
N5 = /W1/
N6 = /W1
RETURN.

Entry Sequence



Note Q stores filled by <dump stack and routines>

Q1 - tape device no /10/112
Q5 = Q6 = 0/1/a58P
Q7 = 0/1/11
Q8 = 0/1/13

```
          begin
61P:      comment entry sequence
1:        * PIAQ 1  ; comment read one block
          * SET 100
          * = RC9
          * ZERO
          * VR
3:        * M8M9Q
          * +
          * J2NV
          * NOT
          * NEG
2:        * J3C9NZ
          * MOM7N
          * J199 ≠
          * +
          * J4NV
          * NOT
          * NEG
4:        * SET 100
          * = RC9
          * MOM7
          * NEG
          * NOT
          * DUP
          * J100<Z
          * DUP
          * J110 = Z
          * SET 2
          * -
          * DUP
          * J120<Z
          * DUP
          * J130=Z
          * NEG
          * NOT
          * J140=Z
          -> 170
```

now M8M9 refrences block elements

compute block checksum on input

compare with block checksum on output
-> 199 if checksum error

switch on block identifier

(warning character for

```
100:    *  ERASE ; comment program      Comes here to process program block
101:    *  SET 100
        *  M5
        *  +
        *  M6
        *  -
        *  J104<Z
        *  JS10          ..... zero next 100 words of core
        -> 101
        *
65P:    *  0/0/160000
        *  0/0/177777
64P:    *  0/0/0
        *  17133/132227/24544
        *  0/0/0
        *  17030/40232/15154
#104:   *  M8M9Q
        *  MOM5
        *  OR
        *  = MOM5Q
        *  J104 C9 NZS
        -> 1

120:    *  ERASE  ; comment set      Process Set block
124:    *  JS30          .... unpack next two words
121:    *  DC3
        *  DUP
        *  J122=Z        --> 122 if nil entry
        *  JS40          .... load Q10 with syllable posn/.../address of SET const
        *  MOM13N        ... fetch mask at 65P+1
        *  AND              mask constant
        *  JS50          ... plant constant
123:    *  J121 C3 NZ    --> 121 for second word
        *  J124 C9 NZ    --> 124 for rest of block
        -> 1
122:    *  ERASE
        *  ERASE
        -> 123
```

→ 104 if set or label entries have been loaded into the core area where code is to go.

masks

code for message ENTERED

code for message AA FAIL

OR block into core

```
110:    * ERASE ; comment labels      Process jump addresses
114:    * JS30          ... unpack next two words
111:    * DC3
        * DUP
        * J112=Z        -> 112 if null entry
        * JS40          ... fetch address of jump into Q10
        * DUP
        * DUP
        * SET B 7777    separate 12 bit address part
        * AND
        * CAB
        * SET B 10000   }
                        } separate 1 bit address part
        * AND           }
        * SHL 7         }
        * CAB
        * MOM13         }
        * AND           } separate syllable count
        * SHL 3         }
        * OR
        * OR            recombine
        * JS50          ... plant
113:    * J111 C3 NZ    loop for second word
        * J114 C9 NZ    loop for rest of block
        -> 1
112:    * ERASE
        * ERASE
        -> 113


130:    * ERASE ; comment stack
        * I11           }
        * J60≠ Z        } set pointer to beginning of stack
        * M5            } unless already set.
        * = I11         }
        -> 60                -> 60
```

```
140:    *  M11  ;  comment routine addresses
        *  J60≠Z
        *  I11
        *  = M5
        *  MOM5
        *  = + M5
        *  M5 TO Q11
        -> 60
```

set pointer to beginning of routine address list, unless already set

-> 60

```
170:    *PMDQ1  ; commentrewind work tape
        *  MOM8  ;  comment checksum
        *  VR
        *  MOM7N
        *  +
        *  J171 NV
        *  NOT
        *  NEG
171:    *  -
        *  J199 ≠ Z
        *  M11 TO Q12 ; comment ENTRY
        *  I11
        *  = M5
        *  MOM5N
        *  = +M12
        *  I12 = +1
        *  ZERO
        *  =MOM14
        *  M14
        *  M+I14
        *  =I14
        *  Q14
        *  SET 8
        *  OUT          ; comment ENTERED
        *  DC11
```

Process checksum block

total up input checksum

subtract output checksum

-> 199 if checksum error

set up and type message ENTERED

```
10:     *  SET 100  ; comment clear 100 spaces
        *  = C6
11:     *  ZERO                    'clear to zero next 100 words
        *  = MOM6Q                  of core for blocks
        *  J11 C6 NZ
        *  EXIT 1

30:     *  M8M9Q  ;  comment unpack 2 words
        *  = Q2
        *  M8M9Q
        *  = Q3
        *  M3
        *  I3
        *  C3
        *  M2
        *  I2
        *  C2
        *  SET 3
        *  = C3
        *  EXIT 1

40:     *  DUP  ;  comment load Q10
        *  SET B 17777
        *  AND
        *  = M10                   13 bit address → M10
        *  SHL -13
        *  SET 7
        *  AND
        *  = C10                   3 bit syllable count → C10
        *  EXIT 1
```

```
        comment time limit is in NS
        *SET 427
        *=M14
        *=MOM14
201:    *J200EJ
        *LINK
        *ERASE
        ->201
200:    *J202 EN
        *ERASE
        ->200
202:    *VR
        *J203 TR
203:    *ZERO
        *=LINK
        *EXIT 300  ; comment into compiled program


199:    *M+I14
        *M+I14
        *ZERO
        *=MOM14
        *M14
        *M+I14
        *=I14
        *Q14
        *SET 8
        *OUT            ; comment AA FAIL
        *ZERO
        *OUT
```

store time limit in E427

clear SLNS, nest, and test register

print AA FAIL

exit to director

```
50:     *  SHL 24  ; comment load pattern as Q10
        *  ZERO
        *  REV
        *  J51 C10 Z
52:     *  SHLD-8
        *  DC10
        *  J52 C10 NZ
51:     *  M6
        *  M10
        *  NOT
        *  +
        *  J53 > Z
        *  JS10
        -> 51
53:     *  MOM10
        *  OR
        *  = MOM10
        *  MOM10N
        *  OR
        *  = MOM10N
        *  EXIT 1
        *

60:     *  M8M9Q  ; comment dump stack /RA
        *  = MOM5Q
        *
        *  J60 C9 NZS
        ->1
end

*

1000: 58P:  end of perm
```

Set constant or jump address planting.

shift word designated number of syllables

clear next 100 words of core unless code already planted there

OR in address or constant

copy block into core

user program and compiler begin here.

Compiling a New Compiler

This section outlines the steps involved in changing part of the AA system or bootstrapping a new compiler by compiling it and its perm with an old compiler. The notations perm 1, compiler 1, and PS1 refer to the old versions of perm, compiler, and compiler phrase structure.

Since Atlas Autocode 'systems' are (a) phrase structure oriented, and (b) divided into two sections (perm and compiler,) copies of the system may be in any of these states:

(1) unstructured - In this state, the compiler has been successfully compiled, but has not yet processed a phrase structure.

(2) structured - In this state, the unstructured compiler has just processed its phrase structure. It is now capable of compiling programs, but exists only as a program-in-code.

(3) defined - In this state, the structured of compiler has been copied onto an input / output stream in its binary form along with a copy of its perm (which it has compiled) and the call section is operational.

Because of (a) above, it is obvious why an operational compiler will need to process a phrase structure. It is further required to compile its own perm (the perm which both it and the programs it compiles will use) in order to establish its 'property lists' which contain the addresses of routines and sub routines in perm which may be used by a running program. Thus it is necessary to keep an unstructured version of the AA system as an I/O stream to facilitate the process of changing and bootstrapping new AA systems.

To alter perm, compiler on both involves two or three of the three 'steps' outline below. Two user code programs are used in making alterations: LNAACOMP which loads an unstructured compiler stream into core, and PTFC which plants a system call block onto an I/O stream which will eventually become a defined compiler stream. A special AA source statement, define compiler, has the effect (at compile time) of converting state (2) into state (3) by copying the compiler-perm core image onto an I/O stream.

step A

1. Use LNAACOMP to load unstructured compiler 1 stream.
2. Feed PS1 to structure compiler 1 for source statements in AA1.
3. Compile perm 2 (written in AA1). This resets the property lists of the new structured compiler 1 for perm 2.
4. Compile compiler 2 (written in AA1). The resulting unstructured compiler 2 program will now run with the just compiled perm 2.
5. Output is intermediate unstructured compiler 2 stream.

step B

1. Use LNAACOMP to load intermediate unstructured compiler 2 stream. (Output of step A).
2. Feed PS2 to structure compiler 2 for source statements in AA2.
3. Compile perm 2 (written in AA2). This sets the property lists of compiler 2 for perm 2. Compiler 2 is now structured.
4. Compile compiler 2 (written in AA2). The resulting unstructured compiler 2 is now in its final and most efficient form. In effect compiler 2 has now compiled itself.
5. Output is final unstructured compiler 2 stream.

step C

1. Use PTFC to write call block onto defined compiler 2 stream.
2. Use LNAACOMP to load unstructured compiler 2 stream.
3. Feed PS2 to structure compiler 2 for source statements in AA2.
4. Compile perm 2 and discard the output. Compiler 2 is now structured.
5. Compile source statement define compiler. The result is that a copy of perm 2 and structured compiler 2 are written onto the defined compiler 2 stream.

The output of step C, defined compiler 2 stream, is an operational AA system.

(Note: In most case the language AA1 = AA2).

A complete bootstrapping process involves executing

step A

step B

step C

in order. The more common situation involves changes to perm 2 or compiler 1 in which no real advantage is gained by allowing compiler 2 to compile itself. In these cases the change involves executing

step A

step C

in order, using the output A.5 as the input to C.2.

BACKGROUND RTs.

NOTES

ON

NAMES, LABELS, & JUMPS

Name Handling

Names for variables are treated as follows.

During the recognition phase, names are recorded as character strings in the array <lett>, and pointers to the section of <lett> containing a particular string are stored in an array <word>. Pointers to <word> are planted in the analysis record to represent names.

There is a limit of 256 different character strings which can be used as names. (The same character string may be declared on different levels, of course.) Each string is associated with a number (actually the analysis record pointer to <word>) which is used by the compilation phase.

Suppose the following program were being compiled

```
    begin
1:  integer K,L
        begin
2:      integer K
        end
    end of program
```

During the recognition of (1:) K and L would be filed in <lett>, K would be assigned the number 1,L the number 2. <word(1)> would point to K in <lett>, and <word(2)> to L in<lett>. The analysis record for (1:) and (2:) would be processed as declarations.

The actual text of the name is not available to cSS, and names are processed by their number only. Declaring K and L consists in making entries in two pushdown lists, <tags> and <name>, and allocating storage for them in the program stack. The heads of these lists are array elements. <tags> is declared tags(0:255) and <name> is declared name (0:15)

routine find label

This routine searches the list whose head is <label(level)> for a match with <k> (global variable in compiler). If a match is found, <j> is set equal to the address associated with the lable. If no match is found, <j> is set equal to -1.

routine fill label (integer at, equals)

<at> is the address of a jump instruction which references this label. ''equals'' is the address labeled.
<fill label> puts this information into the nest and jumps to 51P (in compiler) where <equals> and <at> are loaded into the jump labels buffer.

routine fill set (integer equals, at)

<at> is the address of a SET instruction using this constant.
<equals> is the constant's value.
<fill set> puts this information into the nest and jumps to 52P (in compiler) where <equals> and <at> are loaded into the set buffer.

Label and Jump Handling

Labels ([N] [:])

When a label is processed, it (i.e.,[N]) is pushed down into the list <label(level)> along with the current value of <ca>. This enables references to that label to be completed by finding the list element for the label and extracting the compiled address of the label. (Note: For switch labels, see cSS.)

Jumps (->[N])

When a jump-to-private-label is processed, we may distinguish two cases:

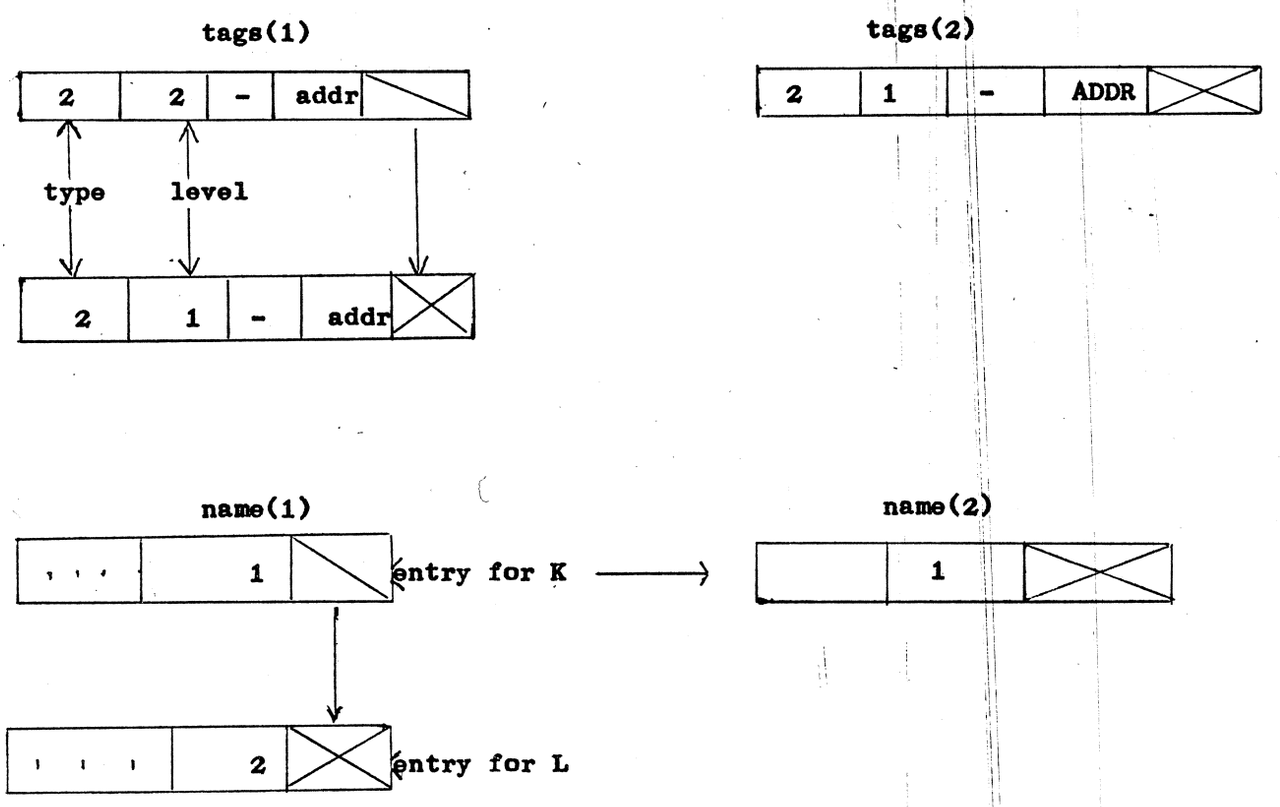
(1) The label referenced has already been processed. In this case the compiled address of the label is fetched from the list <label(level)> and the jump is planted to that address.

(2) The label referenced has not already been processed. In this case the compiled address of the jump compiled is pushed down the list <jump(level)> along with [N] of the label referenced. A jump to 0 (zero) is planted. When end for the block in which the jump was compiled is processed, <jump(level)> and <label(level)> are searched for matching [N], and the compiled addresses of the label and jump are planted in the labels-block so that the label address may be OR ed into the jump instruction when the program is loaded.

If a name has the number n, then the head of the tags list in which it is recorded is <tags(n)>.

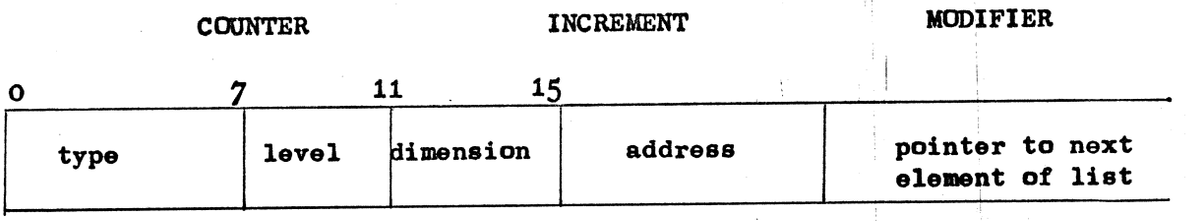
An entry in <name> for a given name consists of the number associated with that name. The head of the names list for a particular name is <name(level)>, where <level> is the textual level of the program when the name is declared.

Thus for our example, after 2 is compiled, the names K would be under <tags(1)> and <name(1)>, <name(2)>; the name L would be under <tags(2)> and <name(1)>:



<name> is used to check whether a name is set twice on one level,(See BACKGROUND ROUTINES <test nst>, and <store name>.)

An entry in <tags> consists of bytes representing the type, level, dimension, and address of the variable, packed as follows.



The name type codes are

0	not set
1	real
2	integer
3	real array
4	integer array
5	switch vector
6	routine
7	real function
8	integer function
9	real map
10	integer map
11	real name
12	integer name
14	address

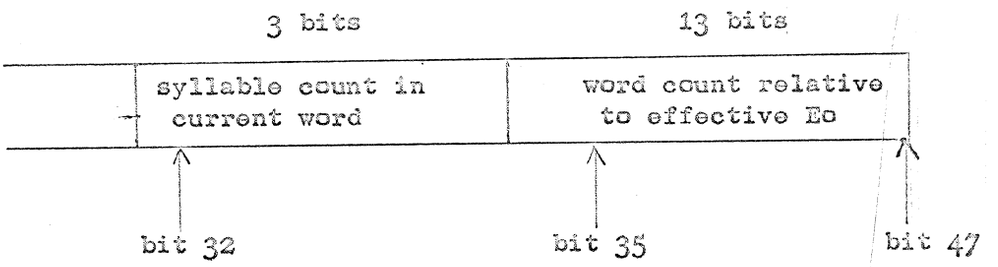
Dimension is coded to represent the dimensionality of arrays, and for routines, functions, and maps has a special use, being set to 0 if the routine has been specified but not described, set to 15 if the routine has been described.

Level codes are by number, from 1 to 15.

Background Routines

integer fn ca

The value of <ca> is the current compiling address in the form



<ca> is computed from <ppcurr> which is stored in Q-store format.

syllable count in currentword / 150 / word count relative to E0+150,

150 words being the length of the call block.

routine store jump

This routine executes <pushdown2(jump(level),ca,k)>

<k> is the identifier of the label referenced by the jump.

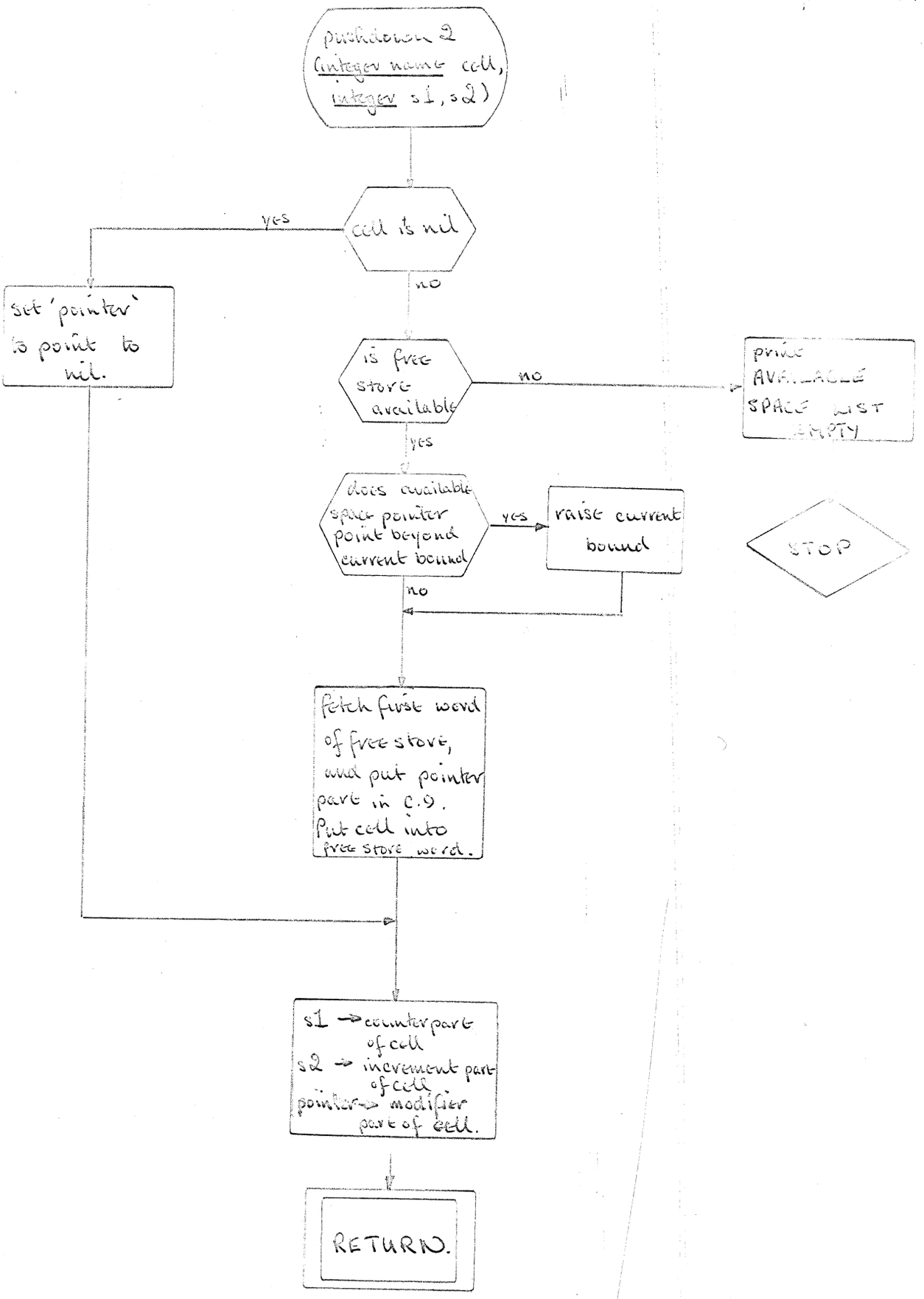
routine store name

This routine executes <pushdown2(name(level),0,k)>

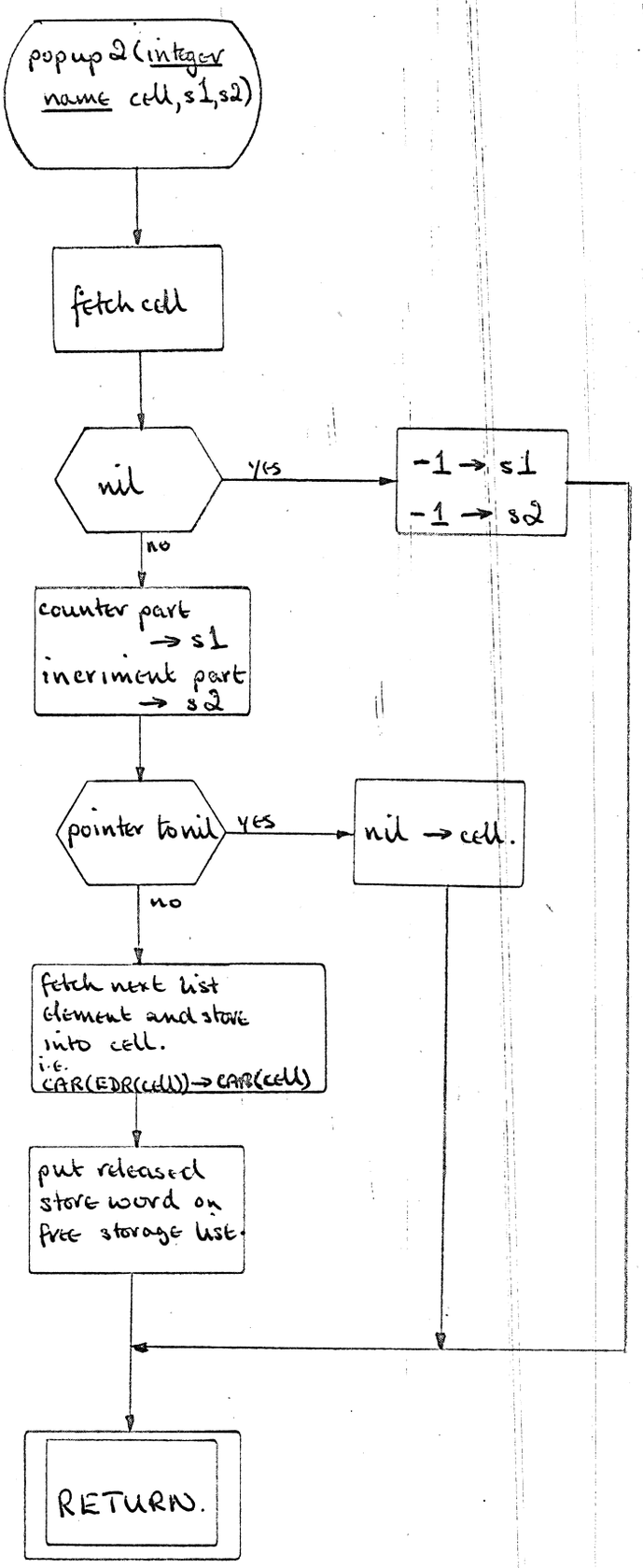
<k>:s the number associated with a particular name. The name(.) lists are kept as a flag telling whether or not a name is set on a particular level.

routine test nst (test whether name set twice)

This routine searches the list whose head is <name(level)> for an element whose value is <k>. If the search is successful, the program is faulted: NAME SET TWICE, and <test nst> returns. If the search fails, the name is not set, and <test nst> returns.

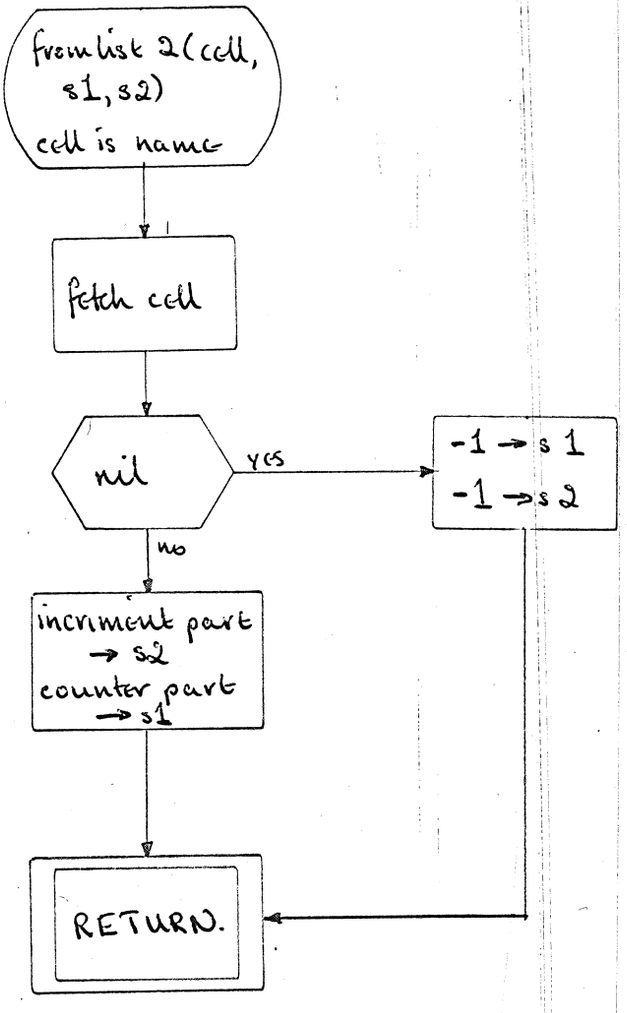


NOTE C9 points to 1st word of free store list.

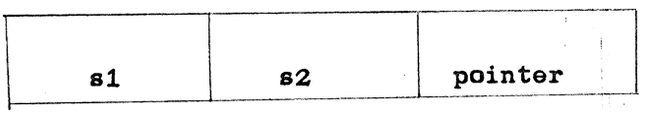


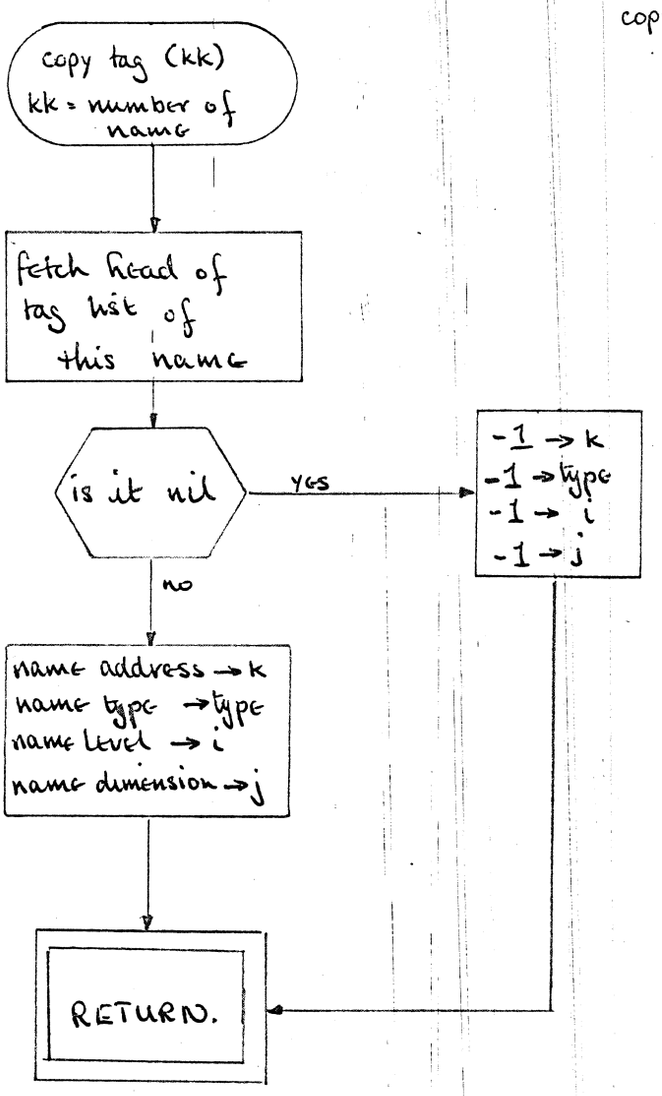
The 'head' of the free storage list is C9, which points to the first free word. The modifier part of each free word points to the next free word, etc.

from list 2.



Q store format for list 2 elements

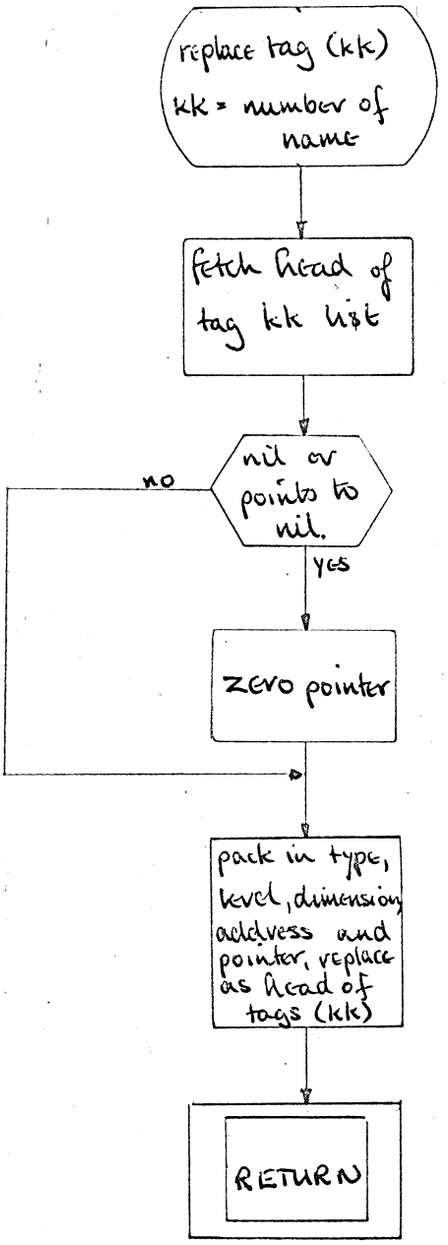




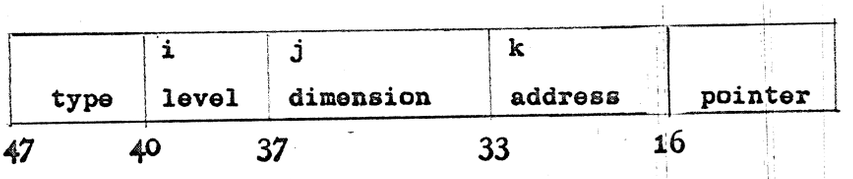
format of tags list element

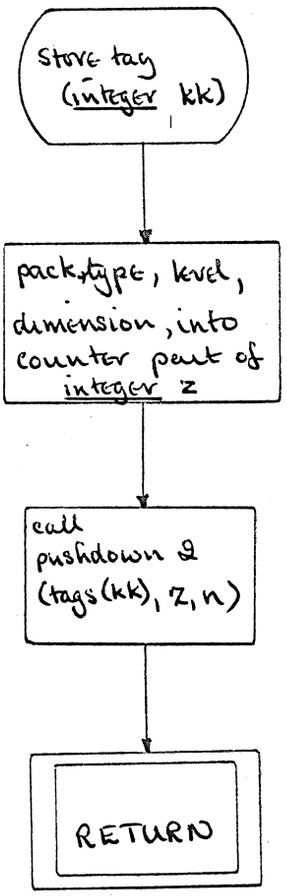
(Q store)

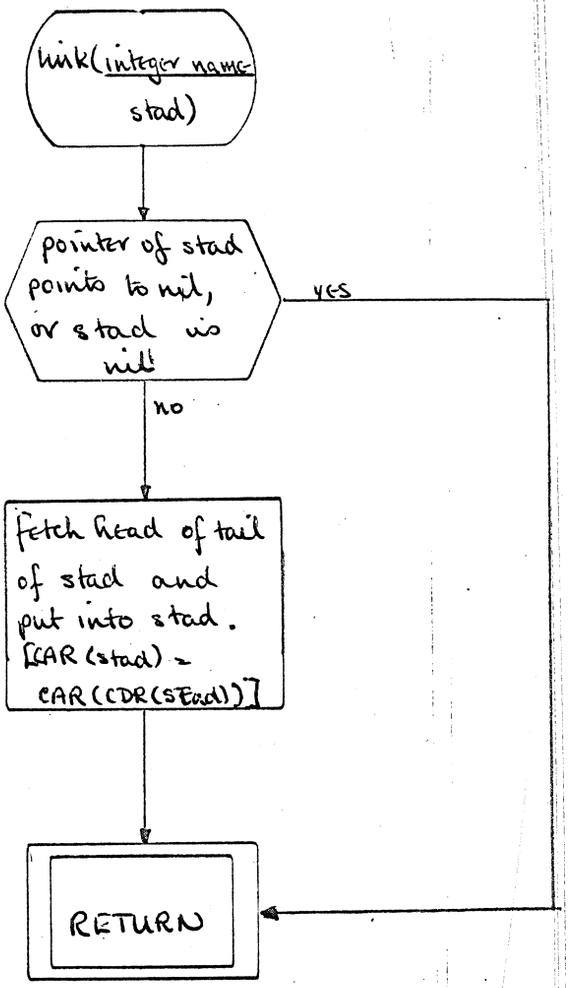
The address part is the stack address of the value for names of simple integers or reals. It is the address of the base address of the array for array names.



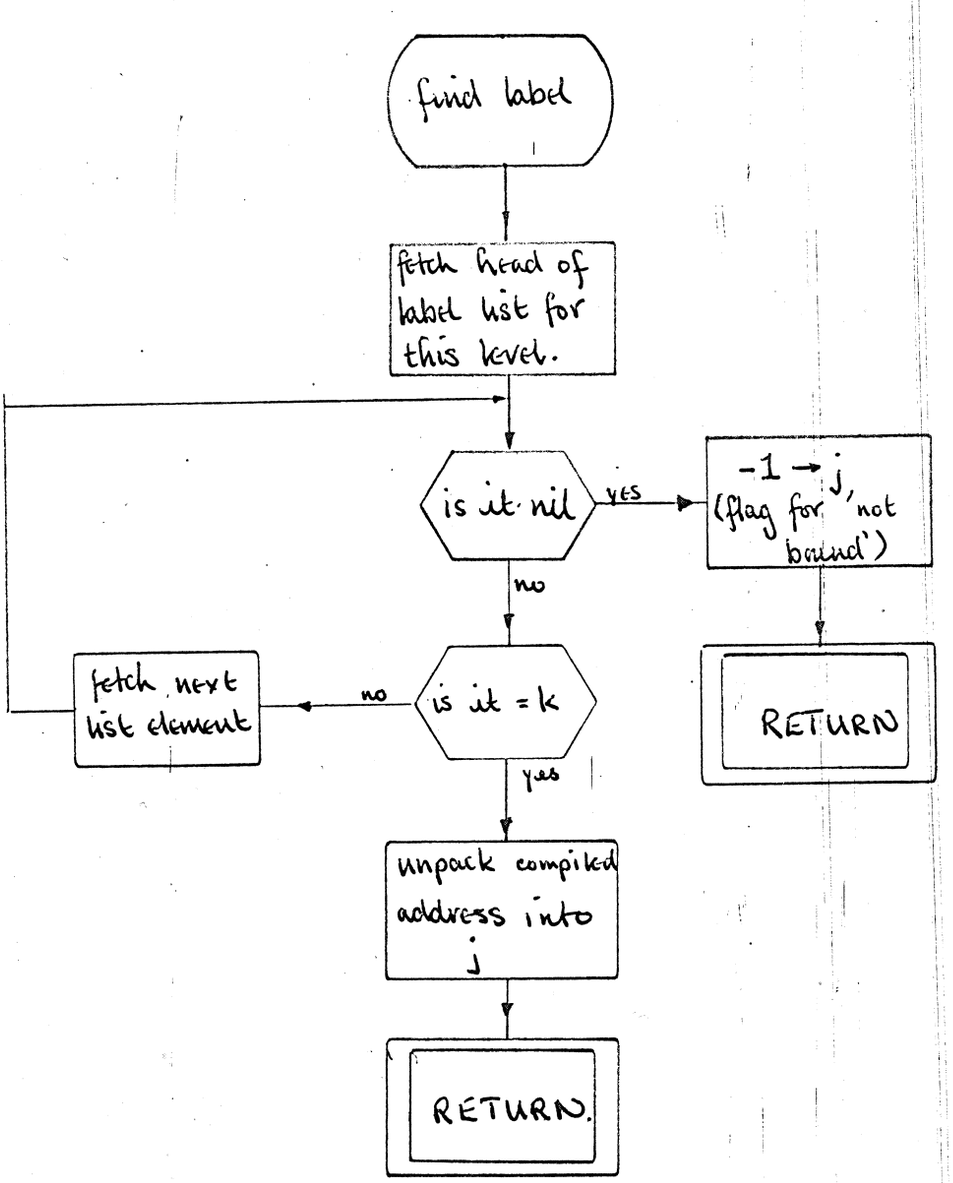
format of tags element




store tag
(integer kk)
pack,type, level,
dimension, into
counter part of
integer z
call
pushdown 2
(tags(kk), z, n)
RETURN

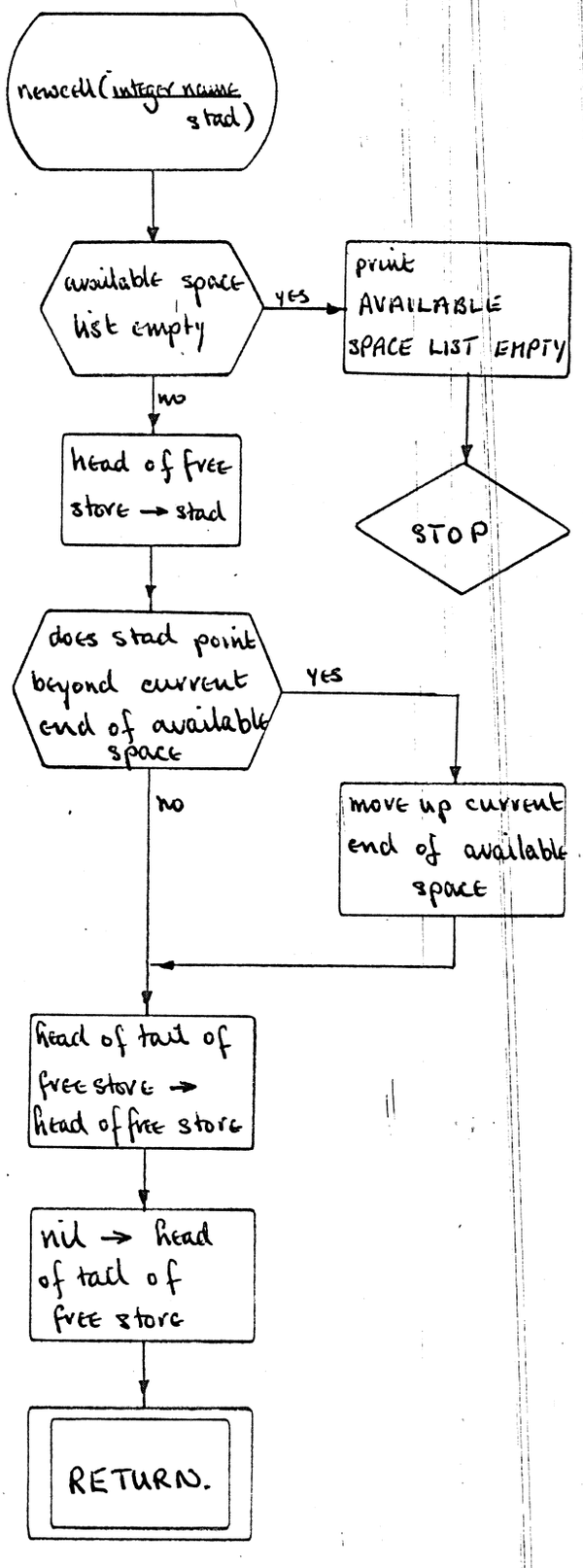


link effectively sets stad = CAR(CDR(stad))



Label list: head = label(level)

Q-store format compiled address/label number/pointer


newcell(integer name stad)
available space list empty
yes
print AVAILABLE SPACE LIST EMPTY
STOP
no
head of free store → stad
does stad point beyond current end of available space
yes
move up current end of available space
no
head of tail of free store → head of free store
nil → head of tail of free store
RETURN.

Arrays

ST	stack being compiled for program. Also used by compiler.
A	analysis record
word	contains pointers to name text in <lett>
lett	contains name-text
tags*	property lists for user declared names
SL	subroutine list. Addresses of P-labels e.g.,SL(5) contains the address of 5P:
symbol	contains phrase structure tree
sidelink & mlink	list links
RA	contains routine addresses. RA(n)=address of routine whose number is n.
pp buf	program buffer
ps buf	set buffer
pl buf	label buffer
T	contains the priority of arithmetic operations

*Names starred are list heads.

cycle*
name*
label*
jump*

list heads for cycles,name,labels,and jumps.
(See cSS and Background Routines)

SET	holds value of SET constant for saving workspace pointer when entering new textual level (cf.,cSS)
RAL(i)	holds number of variables allocated on level i at entry to level i+1.
flag	check for matching block delimiters.

RUN-TIME Q STORE LAYOUT

Q	C	I	M	
1		47	β1	data reference points
2		line no.*	β2	
⋮				
9			β9	
10			mod	used in [EXPR]
11	≠0	αST(0)	αRA(0)	for this program
12		1	ws	
13			mod	
14				reserved for routines and functions
15				evaluating parm routines.

COMPILE TIME as above but in addition

Q9 asl ptn αST(0) ST is array declared in compiler.

Q5-Q8 used by Recognition Routine also. [Version G]

C11 = 0 during compilation.

* Version G, H, I, J

C4 = { 0 : normal delimiters
-1 : upper case delimiters

C3 = { 0 : one character consts
-1 : multi-character consts (i.e. strings)