IMPLEMENTATION OF NARP

Roger House

Document No. M-16
Issued January 25, 1968
Office of Secretary of Defense
Advanced Research Projects Agency
Washington, D. C. 20325

Related Documents

1. For a description of the input to the NARP assembler (i.e., the source language) see Project GENIE Document R-32, Reference Manual for NARP, An Assembler for the SDS 940, issued
2. For a description of the output produced by the NARP assembler (i.e., the input to the loader, DDT) see Project GENIE Document R-25, Format of Binary Program Input to DDT, issued January 26, 1967.
3. For a description of the SDS 940 and its instructions set see one of the following:
   a. SDS publication: SDS 940 Computer Reference Manual, No. 90.06.40A, August 1966.
   b. Project GENIE Document R-27: SDS 930 Instructions, issued October 11, 1966.
4. For a description of ARPAS, the predecessor of NARP, see Project GENIE Document R-26, ARPAS, Reference Manual for Time-Sharing Assembler for the SDS 930, issued February 24, 1967.

TABLE OF CONTENTS

TABLE OF CONTENTS FOR FIGURES

1.0 Introduction

This document is intended for use as a companion to the listing of the NARP program, and thus only contains information not given in the comments to the program. Therefore, precise table formats will be found in this report, but no specification of input or output for subroutines, and only sketches of most algorithms.

This report will probably be of most use to those who wish to correct bugs or to make changes or additions to NARP. It is not written as a primer on assembly writing, but some motivation for the implementation is given, so that a relatively experienced programmer should be able to use it with little trouble.

Following is a short description of the over-all workings of NARP: When NARP is called by a user, the entire program is read off the drum and into core, where it stays until the assembly is over. The assembly is done in one pass: NARP reads an input file containing a source program and writes an output file containing relocatable binary program ready to be loaded by DDT; no other input/output operations are performed except for messages on the teletype, and perhaps a listing. As the program is translated, several tables are built up in core, the principal ones being the main table (containing literals and definitions of symbols and opcodes) and the string storage (containing macro definitions).

The basic scheme for processing an instruction in the source language is quite simple: The opcode is looked up in the main table to get its value and this is merged with the value of the operand which is obtained by evaluating the expression in the operand field. The instruction so constructed is then output. The evaluation of the expression may involve the lookup of symbols in the main table to get their values. If a symbol is undefined, a chain of references to it is created in the object program, and the loader replaces the links of this chain with the actual value of the symbol at load time. Undefined expressions consisting of more than one undefined symbol are saved in a

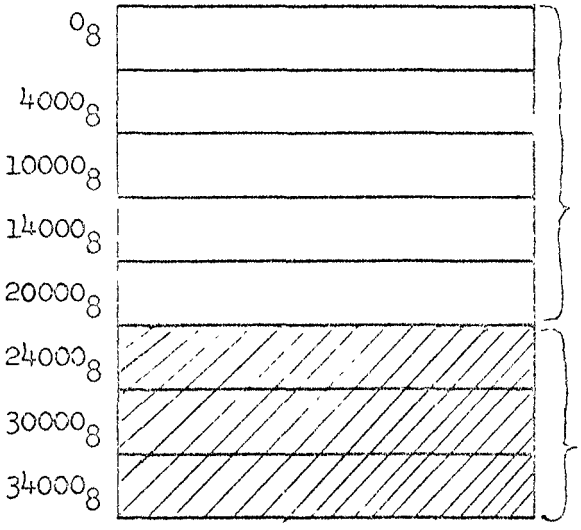
table in core until all the symbols in them become defined, at which time the expressions are evaluated.

Of course, the general scheme above doesn't account for the processing of directives or macros, which makes up an appreciable part of the code for NARP. Refer to the listing and the following report for more details.

2.0 Organization of NARP in Core

2.1 Core Maps

When NARP is running, the core is used as follows:



See Figure 2-1 for the details of the first five blocks of core. The areas in these blocks are discussed below:

pop communication cell: This is simply cell ∅, used by pops to transmit parameters and return address.

cells 1 through 77B: These cells are not used by NARP or the system.

pop links: This is cell 100B through 177B; it contains the links for the pops used by NARP. This area is initialized when NARP is started and then never altered again.

programmed interrupt links: These are cells 201B through 224B; if programmed interrupts are used, these cells contain pointers to the routines to handle the interrupts; they are not used or altered by NARP at present.

	[illegible] subroutines	3.1
	DTAB and MACT	6.4
	normal temps	2.1
~1010	recursive temps	2.1 and 6.5
	recursive subroutine links	2.2
	normal subroutine links	2.2
	freeze area	7.0
	table bounds	2.1
	initial reference tables	4.1
5000	main table	4.1
2250	string storage	6.0
1000	expression table	4.4
40	input pointer stack	3.1
90	character stack	4.2
99	operand stack	4.3
25	operator stack	4.3
~ 276	pile	6.5
300	input buffer	2.1 and 3.0
150	output buffer	2.1 and 3.0
10240		

altered subroutines: These are subroutines that are not pure procedures but whose instructions are altered periodically (in order to make the subroutines as fast as possible). These subroutines pack and unpack characters.

DTAB and MACT: These are two tables used in processing macro definitions. They are placed in read-write memory because they are altered.

normal temps: These are the temporary working cells used by NARP; each one is described in the NARP listing. They are called "normal" to distinguish them from "recursive" temps.

recursive temps: These are temporary working cells used by routines that are called recursively (e.g., the expression evaluator, EXPR).

recursive subroutine links: These are the cells where return addresses for subroutine calls are stored. The recursive subroutines are distinguished because their return addresses must be saved when the subroutines are called recursively.

normal subroutine links: Same as recursive subroutine links but for non-recursive subroutines.

freeze area: When FREEZE is encountered, information about the various tables in NARP is stored in this area.

table bounds: Pointers and upper bounds for the variable tables used by NARP (main table through pile) are stored here. See NARP listing for format.

initial reference tables: These are tables used to make main table lookups fast.

main table: Symbols, opcodes, and literals are stored here.

string storage: Macro bodies, macro arguments, and repeat bodies are stored here.

expression table: Undefined expressions are stored here until they become defined.

input pointer stack: The input pointer is stacked here when input is switched from one source to another (e.g., from an input file to a macro).

character stack: Characters are stacked here during token recognition.

operand stack: Operands are stacked here during expression evaluation.

operator stack: Operators are stacked here during expression evaluation.

pile: Recursive temps and recursive subroutine links are stored here when a recursive subroutine call occurs. This table has no pre-set length but simply takes up all the core in the first five blocks that is left over after the other tables and areas are defined.

input buffer: Blocks from the input file are read into this area.

output buffer: Blocks to be written on the output file are stored here.

2.2 NARP As A Pure Procedure

NARP is coded as a pure procedure (i.e., it is re-entrant) so that several users may use the same copy of NARP at the same time (of course, each user has his own working storage). Since some temps used by NARP have initial values, it is necessary to have some means of getting these values initialized for each user. In NARP this is done simply by including an extra block on the drum which contains all the initial values for the initialized temps, as well as the pop links (see Section 7 for a description of what happens when NARP is started).

Subroutine linkage is handled as in the following example:

```
SUB     ZRO   Ø       RETURN ADDRESS SAVED HERE      } in read-write memory
        BRU   ESUB
ESUB    < body of subroutine >                       } in read-only memory
        BRR   SUB
```

This scheme uses no more memory cells than that using SBRM and SBRR and has the advantage of being faster. The only drawback is that one must remember to precede a subroutine name with an 'E' when writing the subroutine. A call of the above subroutine looks like

```
        BRM     SUB
```

Assume that the subroutine SUB is to be added to the MAC package of NARP; it should be done as follows:

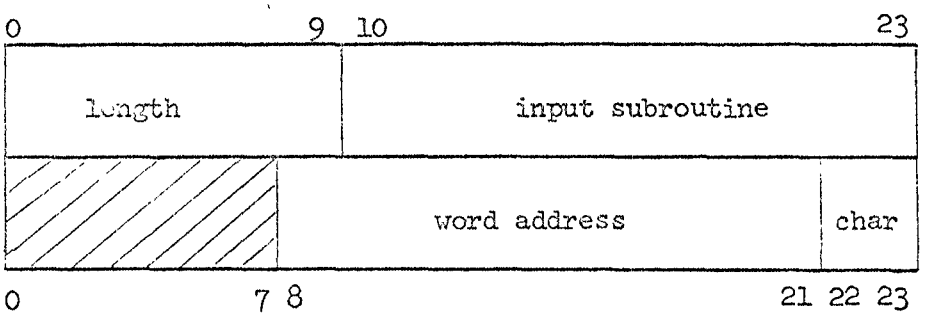
1. Insert the name SUB in either the macro RSTRG or the macro SSTRG (depending on whether SUB will be called recursively). These two macros are the very first macros defined in the DEF package.
2. Insert the code for the subroutine in the MAC package. Don't forget that the first instruction of the subroutine is labelled ESUB, not SUB.
3. At the end of the MAC package insert ESUB in the list of external symbols.

3.0 Input/Output

3.1 Input

The input source program is read blockwise by GETBLK in MISC, and NARP itself does the unpacking of characters. This unpacking is done by so-called "altered routines" which are in the read-write portion of NARP because some instructions in these routines are altered to make the unpacking as fast as possible. The four unpacking routines are FILE, UNPCK, DUMI, and DIGI (see NARP listing for when which routine is used), but they are all called indirectly by BRM* INPUT, so most of the time the main part of NARP does not know where characters are coming from.

Since macro calls may occur within macro calls to an arbitrary depth, information about where the input is coming from must be stacked and unstacked. This is done in the input pointer stack (by STKI and USTKI in MISC), which has the following format:



length: This is only applicable when the input subroutine is DUMI; it is the length of the part of the argument that is yet to be processed.

input subroutine: This is the address of the input subroutine, i.e., one of FILE, UNPCK, DUMI, or DIGI.

word address: This is the address of the word from which the next character will be taken when this input source is unstacked.

char: This is the position of the next character within the word specified by word address; it is not applicable for DIGI.

Although BRM* INPUT delivers the next character, this is seldom what is wanted. Thus, BRM GNLC is used instead because GNLC (in CENTRL) does some preprocessing (like weeding out most characters > 77B) that is usually necessary no matter which part of NARP is asking for the next character. (Currently the only places where BRM* INPUT is used are in GNLC and in LCHAR (in MISC). Due to later changes to GNLC (to make it handle 135B characters in two different ways depending on a flag) it is now possible to replace BRM* INPUT by BRM GNLC in the routine LCHAR (of course, other changes should be made to LCHAR at the same time). From a logical point of view it would be wise to make these changes.)

3.2 Output

Binary program produced by NARP is output in blocks (by OUTB in MISC). The house-keeping needed to organize the binary program into blocks preceded by 3-bit codes is done by the routines BPF, MWC, SWC, FF, and OVAL in MISC.

4.0 Basic Processes

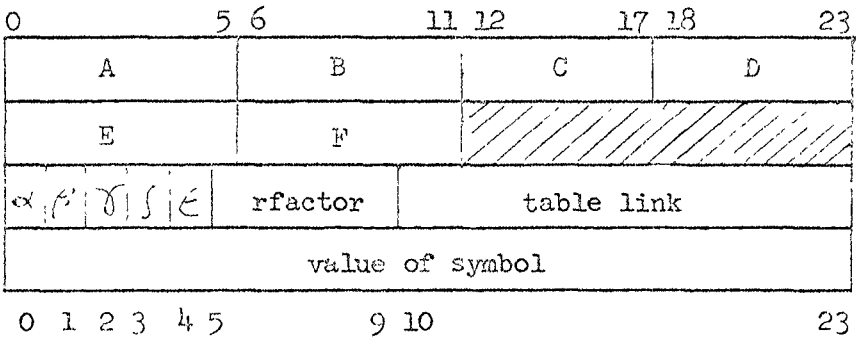
4.1 Lookup of Symbols, Opcodes, and Literals (including format of main table)

An entry in the main table consists of four words, representing a symbol, an opcode, or a literal. These entries are arranged so that the first two words and the address part of the third word are the same (as far as the lookup routine is concerned) for the three different types of entries (see Figures 4-1, 4-2, and 4-3).

As new symbols, opcodes, and literals are encountered they are added at the end of the main table, so if scanned linearly this table contains symbols, opcodes, and literals all mixed together in an unpredictable order. However, this table is not scanned linearly; its various components are linked together via a "table link," the address part of the third word of each entry. Thus, for example, it would be possible to link all symbols together in one chain, all opcodes together in another chain, and all literals together in yet a third chain. Then when a symbol was to be looked up, the lookup routine would only need the symbol in question and the address of the first symbol in the main table. By following the table link from this first symbol, all symbols in the main table could be examined. This, in fact, is precisely how literals are handled, i.e., all literals are linked together in one chain, and when a literal is to be looked up this chain is followed from the first literal in the table until either a match is found or the chain ends (in which case the literal being looked up is not in the table).

Although opcodes and symbols could be looked up in the same way, it is much faster to organize things slightly differently.

Figure 4-1: Format of a Symbol in the Main Table



ABCDEF: This is the character string forming the symbol; if it is not six characters long, then there is blank fill on the right. The shaded area is all zeroes.

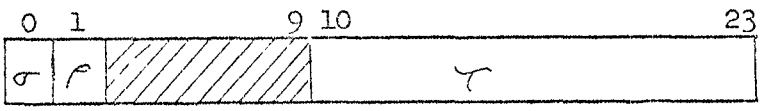
α : symbol is defined
β : symbol is equated
γ : symbol is to be forgotten
$\int$: symbol is generated
ϵ : symbol is external

} These are all booleans; 1 means true, ϕ means false.

rfactor: This is the relocation factor for the symbol; it is stored as a signed, two's complement integer with the sign in the first bit (thus, if the third word of the above entry is in A, LSH 5; RSH 19 will leave the rfactor in A).

table link: This is the address of the second word of another symbol in the main table, a symbol with the same hash code as ABCDEF.

value: If the symbol is defined, then this is its value. If the symbol is undefined, then the value word is interpreted as follows:

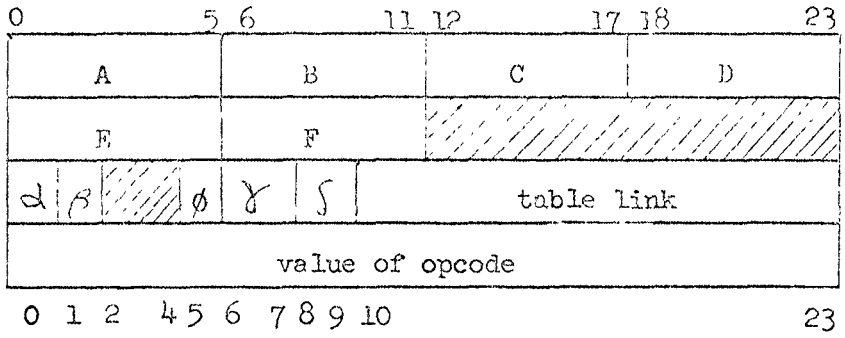


if $\sigma = \phi$ then $\rho = \phi \rightarrow \tau$ is meaningless (ϕ)

$\rho = 1 \rightarrow \tau$ is the address of the last output instruction referring to ABCDEF.

if $\sigma = 1$ then ρ is meaningless (ϕ) and τ is the address of the first occurence of the symbol ABCDEF in the expression table.

Figure 4-2: Format of an Opcode in the Main Table



ABCDEF: This is the character string forming the opcode; see Figure 4-1.

$\alpha = \emptyset$: non-shift opcode
$\alpha = 1$: shift opcode
} see NARP Reference Manual, Section 3.0

$\beta = \emptyset$: do not output opcode definition (the case for macros and pre-defined opcodes)
$\beta = 1$: output opcode definition

$\emptyset$: This bit is always zero so that it, γ, and $\int$ can be accessed together in analogy with the rfactor of a symbol.

γ: Class of opcode:
- $\emptyset$ - operand optional
- 1 - no operand
- 2 - operand required

$\int$: Type of opcode:
- $\emptyset$ - instruction
- 1 - directive
- 2 - macro

Table link: This is the address of the second word of another opcode in the main table, an opcode with the same hash code as ABCDEF.

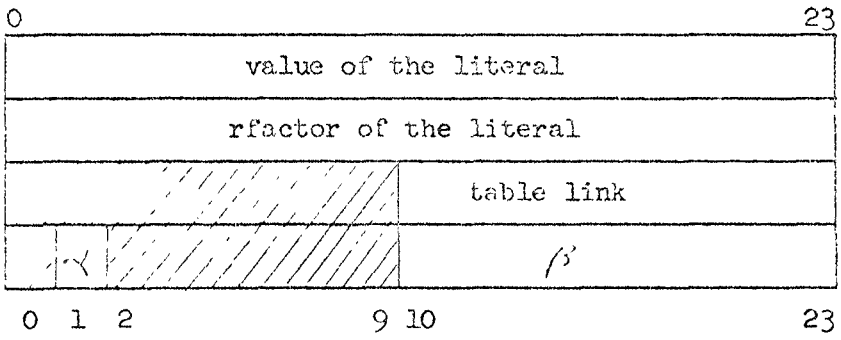
value: the interpretation of this word depends on the type of the opcode:

instruction: value of the instruction.

directive: address of the routine in NARP for processing the directive.

macro: address of the first word of the macro head in string storage (see Section 6.5)

Figure 4-3: Format of a Literal In The Main Table



value: This is the value of the literal.

rfactor: This is the relocation factor of the literal.

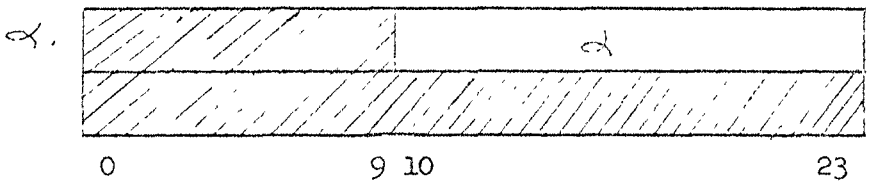
table link: This is the address of the second word of another literal in the main table.

$\alpha = \emptyset$: β is meaningless ($\emptyset$).

$\alpha = 1$: β is the address of the last output instruction referring to the literal.

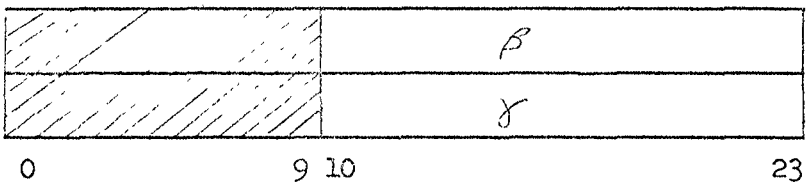
Figure 4-4: Format of an Initial Reference Table Entry

Initial contents of an entry:



α : This is simply the address of the word itself. Thus, the first word of an initial reference table entry simply contains its own address. The shaded areas are zero.

Contents of an active entry:



β : Address of the second word of the last main table entry in a chain.

γ : Address of the second word of the first main table entry in a chain.

Namely, when a symbol is to be looked up, a function is applied to the character string forming the symbol to yield an integer called the hash code for the symbol. This integer is even and is in the range [0,62], and is used to access an initial reference table (see Figure 4-4). An initial reference table is essentially a linear array of pointers to the beginnings and endings of chains in the main table. Thus, instead of having all symbols on one chain, they are distributed over 32 chains, and all the symbols on a given chain have the same hash code. Thus, the number of symbols that must be looked at before discovering if a given symbol is in the table is reduced from n to (on the average) $\frac{n}{32}$, where n is the number of symbols in the table (this, of course, assumes that the function distributes symbols evenly over the 32 possible hash codes). The lookup works as follows: (Input is a symbol and the address of the base of an initial reference table.)

1. Compute the hash code for the symbol and add it to the base of the initial reference table given as input. Thus, two words are accessed which delimit the chain on which the symbol must be, if it is in the table at all.
2. Alter the table link of the last symbol in the chain so that it points to the symbol being looked up. (Note: The symbol to be looked up is always placed at the end of the main table before calling the lookup routine). This essentially adds one more element to the end of the chain, namely the symbol to be looked up.
3. Now start at the first symbol in the chain and follow the chain looking for a symbol equal to the symbol being looked up. It is guaranteed that such a symbol will be found because it is always the last symbol on the chain.
4. When the symbol being searched for is found, check if it is the last symbol on the chain. If not, the symbol being looked up is in the table and has been found. On the other hand, if it is the last symbol, then the symbol being looked up is not in the table.

5. In case the looked up symbol is not found, it is usually added to the table. This is done simply by changing the initial reference table entry which points to the previous last symbol on the chain so that it now points to the symbol at the end of the main table. In case the looked up symbol is <u>not</u> to be added to the main table then no action need be taken (this means, in other words, that the table link of the last symbol on a chain may point anywhere).

The lookup of an opcode works in precisely the same way except that a different initial reference table is used. Literals are also looked up in this way, but the hash code is always zero and the initial reference table only contains one entry. (It would be possible to speed up the literal lookup by computing some sort of hash code on the basis of the magnitude of the literal.)

4.2 Token Recognition

The input to NARP is essentially a stream of characters that must be analyzed to find which sequences of them form symbols, which form numbers, etc. For the purposes of this description, a token is a symbol, a number, or a single non-alphanumeric character. The basic routine for recognizing tokens is GNE (in CENTRL).

The logic used in GNE is displayed by means of a state table in Figure 4-5. With this mechanism, a string of characters can be analyzed by using an integer called the mode (or state) to describe the kind of string processed so far. A new character in the string selects a column in the table, while the current mode selects a row; the table entry common to the column and row contains the new mode. This process is repeated until a non-alphanumeric character is encountered, at which time the label indicated by the rightmost column of the table and the current mode is jumped to. Here is an example: (Note: Blanks have no meaning in this example)

```
input string:  1  4  6  B  3  A  *
      mode: 2  1  1  1  0  -2 -3  symb
```

Figure 4-5: State Table for Token Recognition

character / mode	:,A,C,E,F, ..., Z	B	D	0,1,...,9	<all other characters>
2 first character	-3	-3	-3	1	ud
1 in a digit string	-3	0	-1	1	ns
0 after <digits>B	-3	-3	-3	-2	nsb
-1 after <digits>D	-3	-3	-3	-3	nsd
-2 after <digit> B <digit>	-3	-3	-3	-3	nsbn
-3 in a symbol	-3	-3	-3	-3	symb

comment the mode is initially 2 (first character);

symb: pack symbol; compute hash code; goto symbol exit;

nsbn: gnel:=8; comment gnel contains the radix for the number; scale:=last digit stored; chp:=chp-2; comment chp points to characters; goto E2;

nsd: gnel:=10; goto D1;

nsb: gnel:=8; D1: chp:=chp-1; goto E1;

ns: gnel:=radix; E1:scale:=ϕ; E2:concatenate number using gnel as radix and checking that all digits are less than the radix; goto number exit;

ud: goto other exit;

Note that in this example it is not clear that 146B3A is a symbol until the very last character, 'A', is read.

Although the state table could be implemented by having a two-dimensional array of words containing the new modes, it is less space-consuming and probably just as fast to do it in a more ad hoc fashion as is done in NARP. For example, when an input character selects the leftmost column of the table it is not necessary to use the current mode to select a row because the new mode is always -3 no matter what the current mode is. The selection of a column is done by using the input character to access a linear array of 64 words (CTAB) which contains the addresses of five pieces of code corresponding to the five columns in the table. Thus, in the above example, when the first character of 146B3A is read, it is used to access CTAB, where the label DIGIT is found. A jump to DIGIT (in GNE) is made, and there the current mode is looked at to determine what the new mode should be.

The format of CTAB is:

α : used by statement processor (see CLOOP, Section 5)

β : used by EXPRI (see Section 4.3)

γ : address of code in GNE (NOTBD, BLET, DLET, DIGIT, or OTH)

At the bottom of Figure 4-5 is a sketch of the processing done by GNE after a token is recognized. It should be noted that as characters are read by GNE they are stored in the character stack (pointed to by CHP). This stack works recursively because GNE is a recursive subroutine.

4.3 Expressions

A NARP expression is essentially a string of symbols and constants connected by operators. The main routine for evaluating expressions is EXPR (in CENTRL) which also checks their syntax and the compatibility of the rfactors of their terms.

An auxiliary routine, EXPRI, is called by EXPR to get the next syntactical element for EXPR. The flowchart in Figure 4.6 gives an outline of the workings of EXPRI. Note that EXPRI uses a part of CTAB (see Section 4.2 for format) to access another table, namely OPWRD (see Figure 4-7). Although entries in OPWRD describe operators and are used by EXPR when the operations are to be performed, one field in OPWRD (EXPRI type) is used by EXPRI, both to branch to actions within EXPRI and to be delivered in the A register to tell EXPR what kind of syntactical element was delimited. EXPRI also puts information in the operand stack (sometimes called anstk) and in the operator stack (sometimes called atstk).

The syntax of an expression is checked by a state table (see Section 4.2). In this case, the state table is implemented by having a two-dimensional array of words, indexed row-wise by the current mode and column-wise by the type of the syntactic element (delivered to EXPR by EXPRI). The state table is a bit more complex than for token recognition since each entry contains the address of a piece of code as well as new mode; after the mode is changed the piece of code is executed. Below is a pseudo-ALGOL description of the actions of the code pieces; refer to the NARP listing for the state table and for further details.

next syntactical element	next: EXPRI; snmode:=new mode; goto action determined by old mode and EXPRI type;
plus or minus	pm: make newop a unary operator instead of a binary one;
negate or binary op.	nt:bo: STACK; goto next;
symbol or constant	sc: increment anp so the value delivered by EXPRI is now in the operand stack; goto next;

Figure 4-6: Flowchart for EXPRT (in CONTRL)

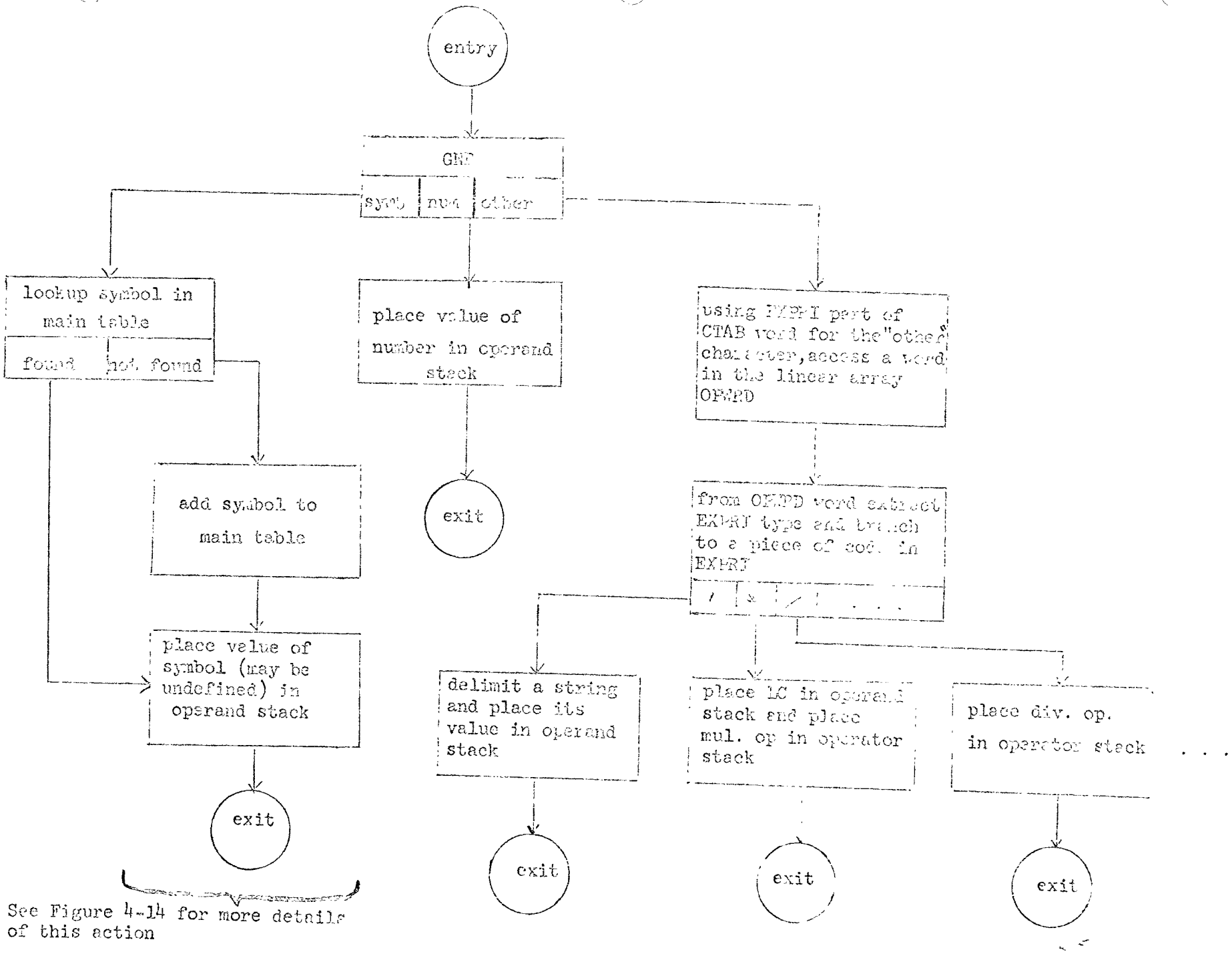
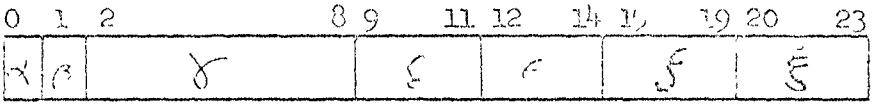


Figure 4-7: Format of OPWRD Table



α : rfactor of result (Ø-absolute, 1-found by applying operator to rfactors)

β : degree of operator (Ø-binary, 1-unary)

γ : EXPRI type (used to branch to a piece of code in EXPRI; see NARP listing)

ζ : rfactor of operands (Ø - all absolute, 1-arbitrary, 2-arbitrary but equal, 3-at least one is absolute)

ϵ : subopcode: used to distinguish the various relational operators

ξ : opcode: used (by EXPR) to access OPTAB when operator is applied to its operands

ε : hierarchy (also called precedence) of the operator

Figure 4-8: Format of An Entry in the Operand Stack

Entry is a place holder:

∅
∅
∅

Entry is an undefined symbol:

	address of fourth word of the main table entry for the symbol
∅	
1	

Entry is a defined value:

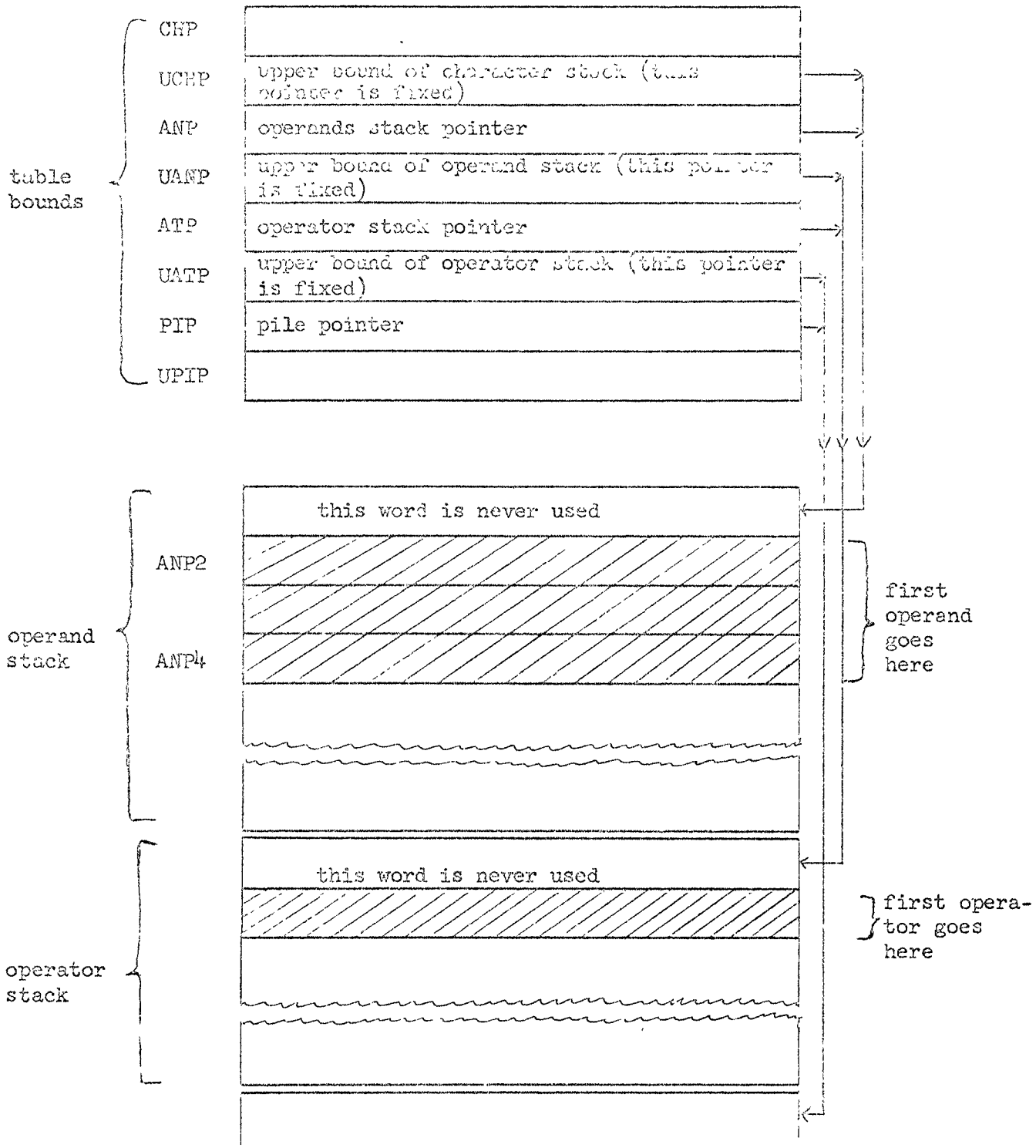
value as a signed integer
rfactor as a signed integer
2

Entry is a defined literal:

	address of fourth word of the main table entry for the literal
∅	
3	

Note that the third word determines what kind of entry it is. The second word is only used for defined values.

Figure 4-9: Organization of the Operand and Operator Stacks



In this figure the pointers ANP and ATP indicate that the stacks are in the neutral state, i.e., nothing is going on that uses the stacks. These two pointers always point to the last active word in their respective stacks.

equal sign	eq: make newop a literal operator instead of a relational operator;
left parenthesis ([)	lp: increment atp and put newop in the operator stack; goto next;
right parenthesis (])	rp: STACK; decrement atp twice to get rid of [] in the operator stack; goto next;
exit	ex: STACK; exit from EXPR;

Of course, the "exit from EXPR" covers a lot of details, particularly about undefined expressions; this is discussed in the NARP listing and in Section 4.4. (See also Figures 4-15, 4-16, and 4-17).

The subroutine STACK does the actual evaluating, applying the operators in the operator stack to the operands in the operand stack. The compatibility of xfactors is also checked by this routine. If an operator is to be applied to an undefined value, special actions must be taken as is discussed in Section 4.4. See the NARP listing for more comments on STACK. Figures 4-8 and 4-9 give details of the two stacks involved; an entry in the operator stack is precisely the same as that in OPWRD (Figure 4-7) except that the EXPRI type is always zero.

The best way to get an idea of how EXPR works is to try a few examples by hand. Below is an example showing the changes in the stacks as "A+B*[C+D]-3;" is evaluated (assume A=1, B=2, C=3, D=4).

Operand Stack	Operator Stack	Comments
A		
A	+	
A B	+ *	'*' has greater hierarchy than does '+' so it is stacked
A B	+ * [	'[' is always simply stacked without comparing hierarchies
A B C	+ * [	

Operand Stack	Operator Stack	Comments
A B C	+ * [+	
A B C D	+ * [+	
A B 7	+ *	']' pushes all operators out of the stack through '['; 7=C+D
A 14	+	first '-' pushes out '*'; 14=B*[C+D]
15	-	then '-' pushes out '+'; 15=A+B*[C+D]
15 3	-	
12		';' pushes out '-'; 12=A+B*[C+D]-3

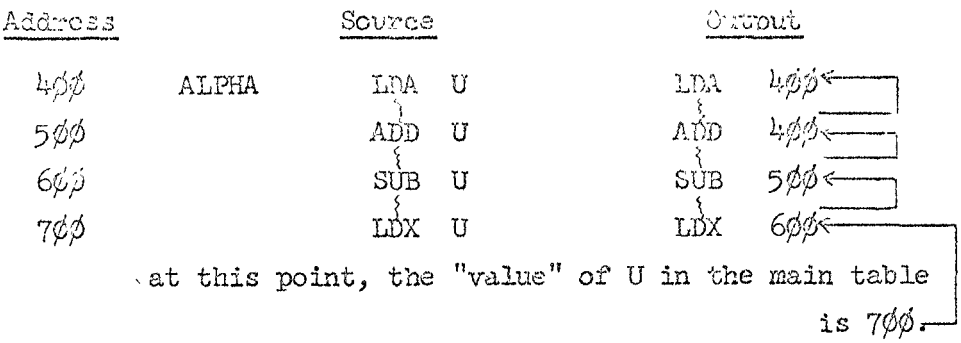
4.4 Handling Undefined Quantities (including literals)

There are essentially three different kinds of undefined quantities treated by MARP:

1. An expression consisting on one symbol which is undefined. This is usually referred to as the undefined symbol case.
2. An expression consisting of more than one symbol (i.e., a symbol and an operator, several symbols and operators, etc.) which contains at least one undefined symbol. This is called the undefined expression case, although strictly speaking, 1. is also an undefined expression.
3. A literal.

4.4.1 Undefined Symbols

The symbol is placed in the main table (if it is not already there), marked as undefined. The "value" of the symbol is the address of the instruction for which the symbol is an operand (see Figure 4-1). The second time the symbol is referred to, the "value" is output with the current instruction, and then the address of the current instruction is placed in the "value." Thus, a chain is created with its head stored in the main table. For example, if U is undefined, one might have:



When the undefined symbol is finally defined, a special piece of output is generated (labfix head) which tells DDT to follow the chain beginning at "head," replacing each link by the current value of the location counter. The end of the chain occurs when

the link in an instruction is equal to the address of the instruction. (Some assemblers indicate the end of a chain by a zero link, but this means, however, that in certain cases an undefined symbol in the first instruction of a program will not be treated correctly.) To continue with the above example, if U DATA -1 appears at location 8ØØ, then

labfix 7ØØ

is output. DDT traces down the chain, replacing the links by the current location counter, 8ØØ. The program then looks like

4ØØ	LDA	8ØØ
5ØØ	ADD	8ØØ
6ØØ	SUB	8ØØ
7ØØ	LDX	8ØØ
8ØØ		-1

It is clear from the fore-going that NARP does not handle forward references completely by itself, but in a sense utilizes the loader as a sort of second pass. This scheme works because the address part of the instruction with the undefined symbol can be used to hold information for the loader, namely a link.

4.4.2 Undefined Expressions

However, if the operand for an instruction is undefined and consists of more than a single symbol (e.g., -U, U+1) then there is not enough room in the instruction to hold all the information, especially since the undefined expression may be very long and may contain many undefined symbols. To solve this problem, undefined expressions are stored in a table in core (called the expression table or etab, pointed to by ETP). When all the symbols in a given stored expression become defined, the expression is evaluated and output to DDT. One other complication enters, however, namely that there is not enough room in the main table entry for both the head of an undefined chain and a pointer to the expression table. Thus, a bit in the "value" word is set

to indicate that this word points to the expression table (see Figure 4-1). The word pointed to in the expression table represents the occurrence of the undefined symbol in an expression; it contains either the head of an undefined chain or else points to the next occurrence of the undefined symbol in a saved expression (see example below).

It should be clear that the case of U as a single undefined symbol is kept completely separate from the case of U as an undefined expression. The occurrence of U in an undefined expression does not at all effect the undefined chain that is output to DDT. The only difference is internal to NARP, namely that the head of the chain may not be in the main table but may be at the end of the chain running through the expression table; it will simply take a bit longer to retrieve it than if it were in the main table.

The expressions are stored by EXPR in Polish post-fix form (see Figure 4-10) along with the address of the instruction where the expression occurred. When an undefined symbol that appears in the expression table is defined, the following action is taken (by the routine FOLEC, see Figure 4-13):

1. The chain running through the expression table is followed and each link is replaced by a pointer to the symbol in the main table (it is not possible to put the actual value of the symbol in the expression table because defined values need three words in this table and there is only one available).
2. In each expression where the symbol appears, the undefined symbol count is decremented, and if it becomes zero (indicating that all the undefined symbols in the expression are now defined) the expression is evaluated and its value is output followed by either

   fix 14 saved lc

   or

   fix 24 saved lc

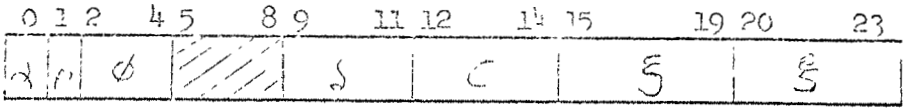
   These cause DDT to fix up the instruction at "saved lc" with the last output value.

Figure 4-10: Format of Saved Expressions

Bits b2-b4 indicate the type of the word stored in the etab.

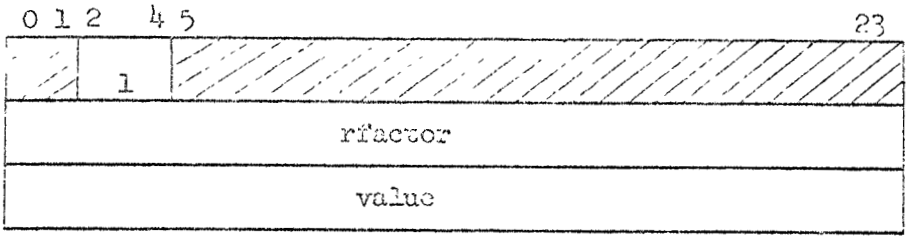
The shaded areas are zeroes.

Operator:

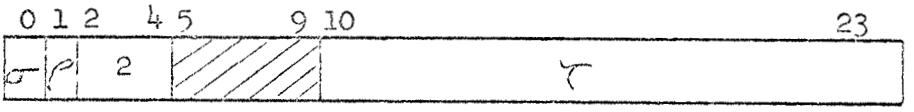


$\alpha, \rho, \zeta, \xi, \xi$: See OPWRD table format, Figure 4-7.

Defined value:

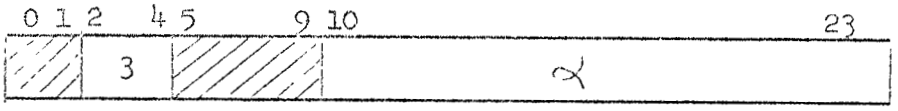


Undefined symbol:



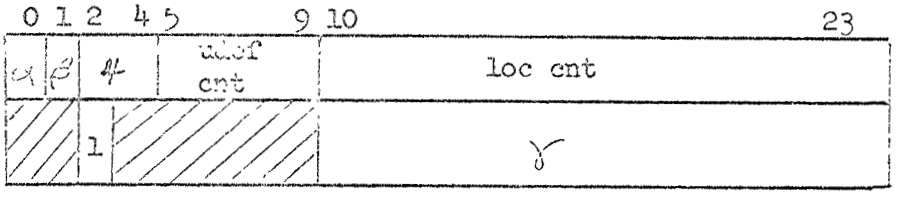
σ, ρ, τ: Same as σ, ρ, τ in Figure 4-1 except that symbol is not necessarily ABCDEF, and τ does not point to the first occurrence of the symbol in the etab.

Pointer to defined value:



α: Address of third word of a defined symbol in the main table.

End-expression:



α: ϕ means that the value of the expression should be computed mod 2^{24}, 1 means mod 2^{24}.

β: ϕ means that loc cnt is meaningless (ϕ), 1 means it has meaning.

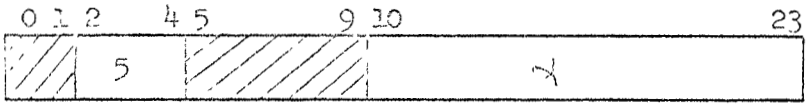
udef cnt: number of undefined symbols in the expression.

loc cnt: address of the instruction in which the expression appeared.

γ: If X register contains (the address of this word) then EAX= ϕ,2 makes X point to the first word of the expression in the etab (thus the word ζ contains 1+ the negative of the number of words in the etab entry, with b0-b2 and b3-b9 zeroed out).

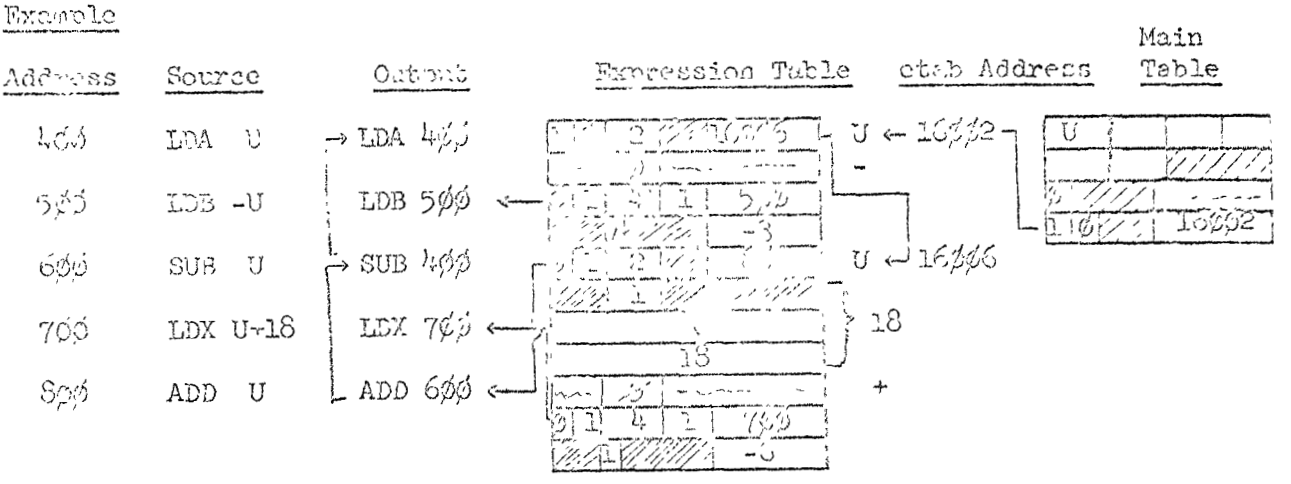
Figure 4-10 (continued): Format of Saved Expressions

Vacant area:



α : number of words in the etab entry; after an expression in the etab becomes defined and is thus evaluated and output, the first word of the entry is replaced by the above word to aid in garbage collecting (see Section 8); the above word is in cell β, then $\beta + \alpha$ is the address of the first word of the next expression in the etab.

3. When the end of the expression table chain is reached, its contents (the head of an undefined chain) are used to output "labfix head" as described above in Section 4.4.1.



Now if U is defined, say by U DATA -3 at location 9ØØ then the following is output:

setrel	-1	(sets special relocation factor)
	-9ØØ	(marked as special relocation; value of -U)
fix 14	5ØØ	(points to location where -U appeared)
	918	(value of U+18)
fix 14	7ØØ	(points to location where U+18 appeared)
labfix	8ØØ	(points to start of undefined chain)
	-3	(contents of word labelled U)

After DDT finished processing the above, the result in core is

4ØØ	LDA	9ØØ
5ØØ	LDB	-9ØØ
6ØØ	SUB	9ØØ
7ØØ	LDX	918
8ØØ	ADD	9ØØ
9ØØ		-3

4.4.3 Literals

The literal operator '=' takes a value as its operand and produces the address of this value. Since literals are output at the end of the program being assembled, the address of a literal is unknown until the end of the program, so that references to literals are logically the same as references to undefined symbols. There are two cases:

1. The expressions following the literal operator is defined:
   In this case, the action of the literal operator (in EXPP) is to look up the literal in the main table (adding it to this table if it is not already there) and return the fourth word of the main table entry as the value. This word is almost the same as the fourth word of an entry for an undefined symbol, i.e., it is the head of a chain of references to the literal. The only exception is that bØ is always Ø because this word never points to the expression table.
2. The expression following the literal operator is undefined:
   In this case the literal is placed in the expression table just as any other undefined expression. When all the symbols in the expression become defined, it is evaluated and processed as described above in 1. and in Section 4.4.2.

When the END directive is encountered, the literals in the main table are output one by one, each preceded by a labfix so that DDT will fix up all the references to these literals. After this the expression table is scanned. All expressions in this table except those of the form =symbol cause error messages because they are expressions involving external symbols. (If DDT is every changed so that it can handle expressions in Polish form, then these expressions will all be output to DDT.) Expressions of the form =symbol are external literals; they are handled by outputting

labfix address of instruction referring to the literal

followed by the equivalent of

DATA symbol

The only drawback to this scheme is that every occurrence in the program of =symbol (where symbol is undefined) is placed in the expression table. If EXPR were changed to recognized =symbol as a special case, a less space-consuming scheme for processing external literals might be devised.

4.4.4 Flowcharts for Routines That Handle Undefined Quantities

The observant reader will have noted that many of the pointers connected with handling undefined quantities are similar. The following four words have essentially the same format as far as bits b1 and b10-b23 are concerned:

1. The fourth word of a main table entry for an undefined symbol (Figure 4-1).
2. An undefined symbol word in the expression table (Figure 4-10).
3. An end-expression word in the expression table (Figure 4-10).
4. The fourth word of a main table entry for a literal (Figure 4-3).

For the first two above, bØ has a common meaning; if it is 1, then b10-b23 point to an undefined symbol word in the expression table, while if it is Ø, then b10-b23 is the head of an undefined chain. In case 3. and 4. b10-b23 is also the head of an undefined chain (although in case 3. this chain always has only one link). In all four cases, if b1 is Ø, then b10-b23 is meaningless; it is to be ignored. The reason this case may arise is that an undefined quantity may appear in a wrong context, but this is not discovered until entries have been made in the main table and the expression table. Thus, it is easier to set a bit than to undo what has been done.

Another format that is used in several places is that of an operator:

1. In the OPWRD table (Figure 4-7).
2. In the operator stack (Section 4.3).
3. In the expression table (Figure 4-10).

The following routines are flowcharted in considerable detail. See the NARP listing for a specification of the input to the routines.

1. DEFLB: This routine defines labels.
2. ASGN: This routine defines equated symbols.
3. FOLEC: This is an auxiliary routine used by both DEFLB and ASGN to process the expression table when an undefined symbol is defined.

Figure 4-11: Flowchart of DEFLB (in CREFRI)

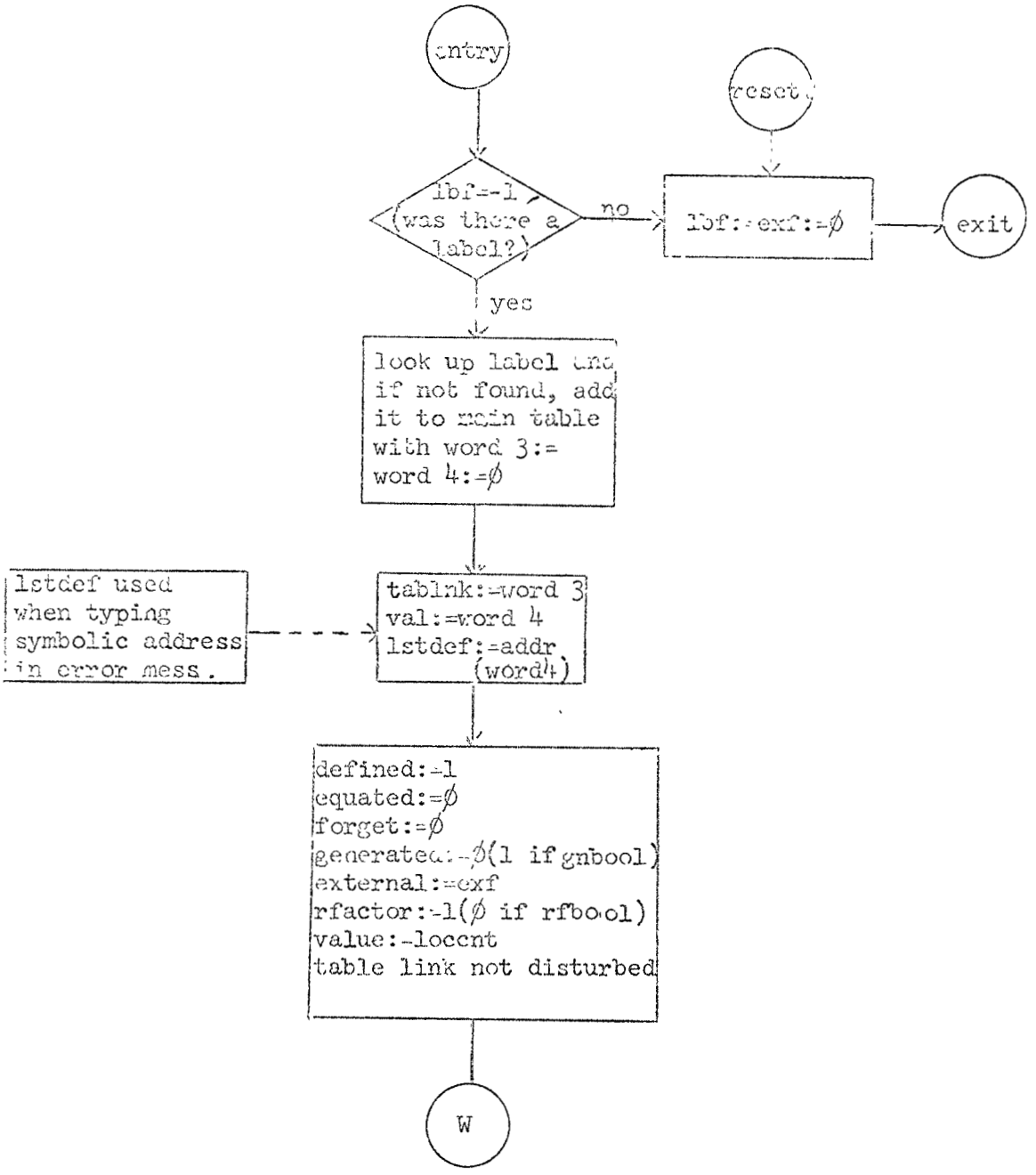


Figure 4-11: Flowchart of DEFLB (in CENTRL)

(cont'd)

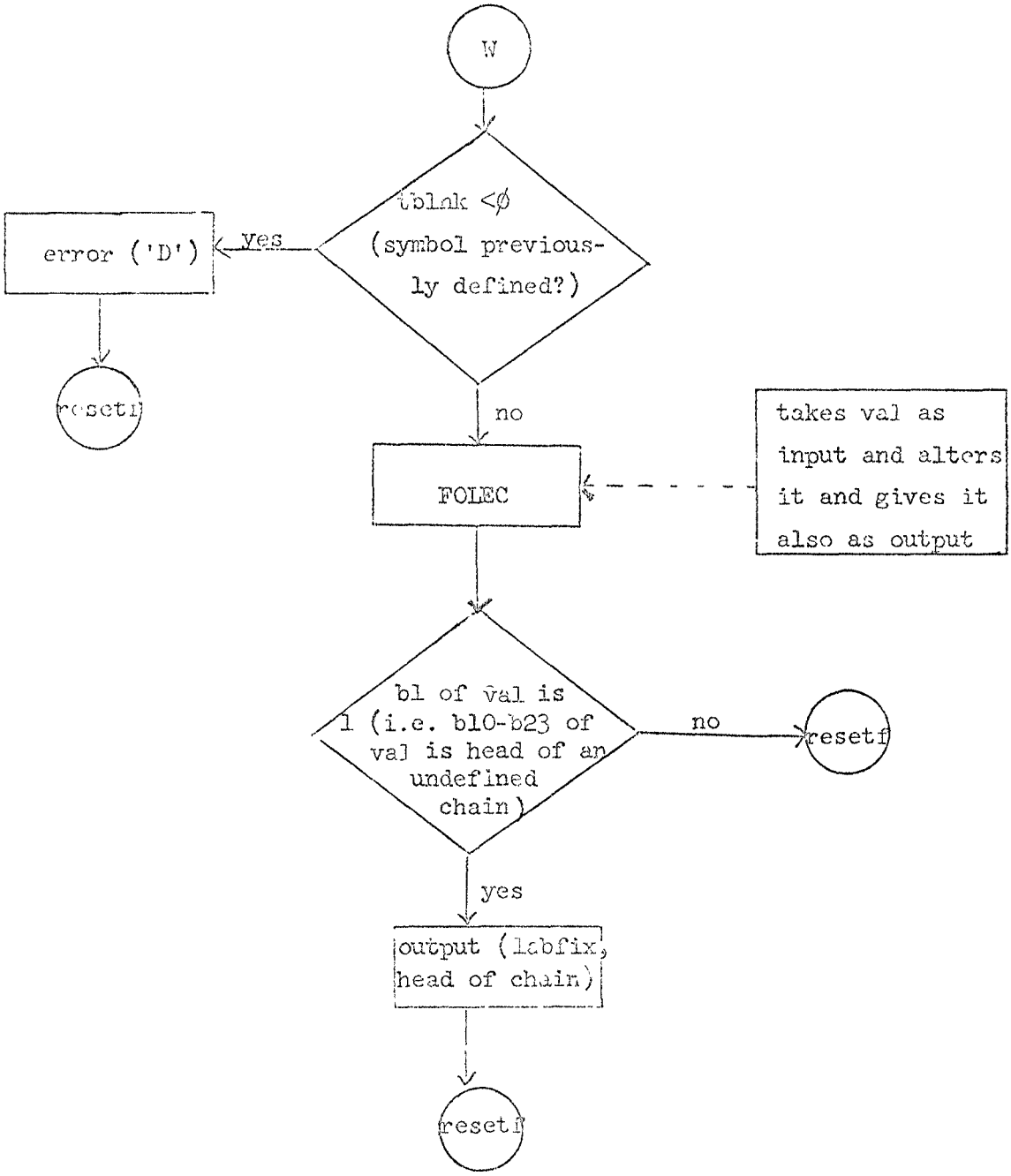


Figure 4-12: Flowchart of ASGN (in DIRECT)

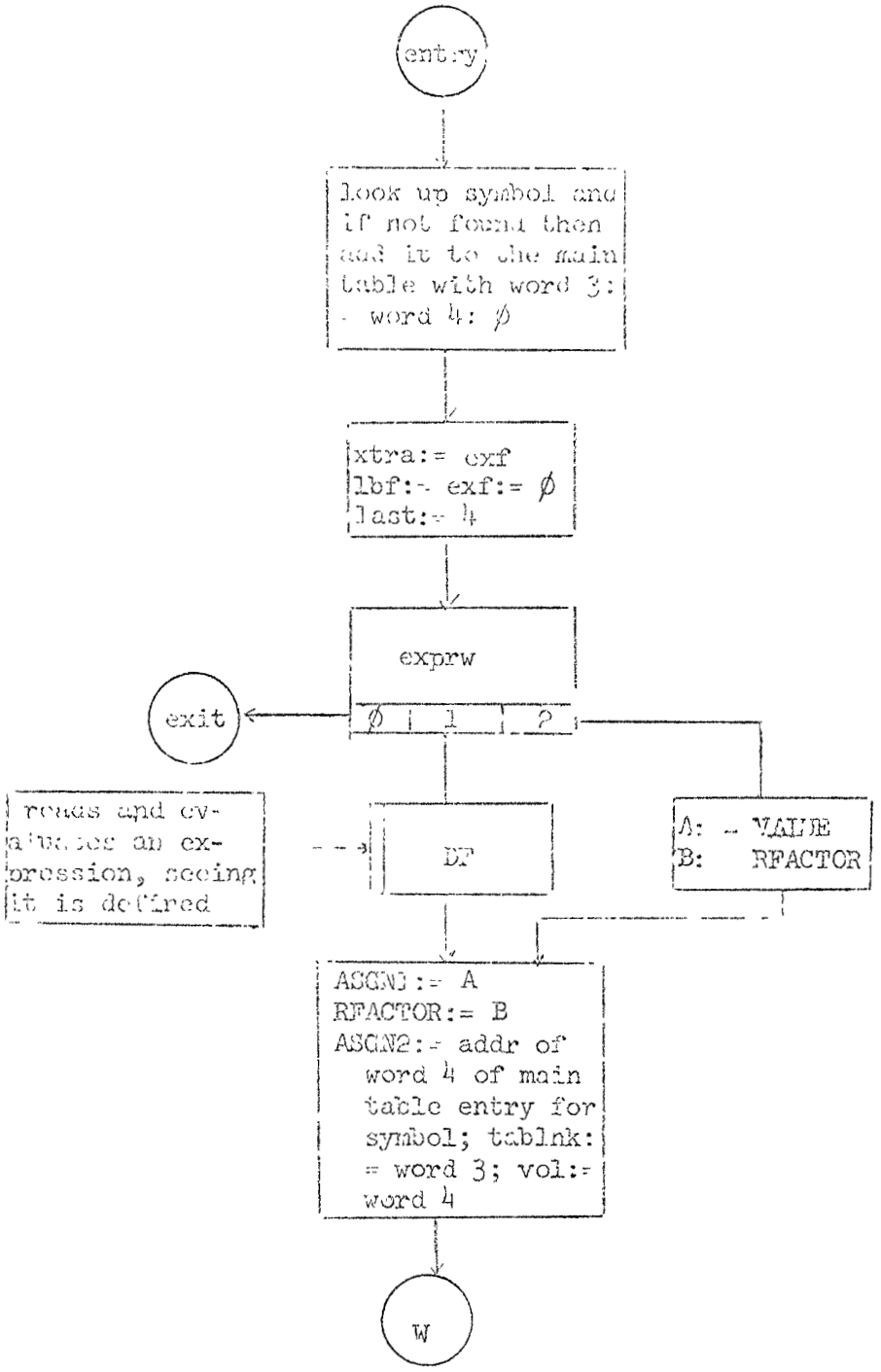


Figure 4-12: Flowchart of ASGN (in DIRECT)

(cont'd)

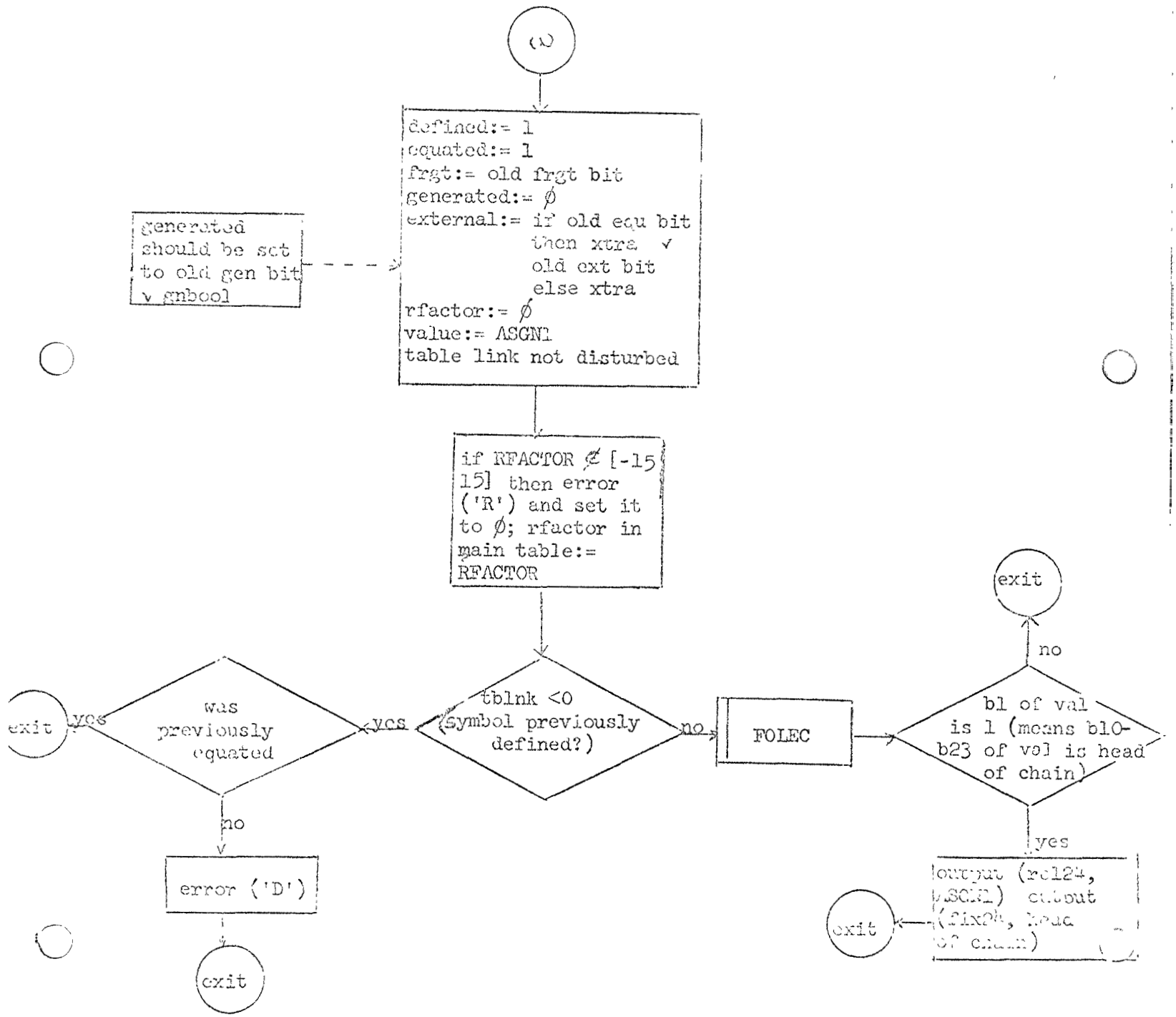


Figure 4-13: Flowchart of FOIFC (in CENTRL)

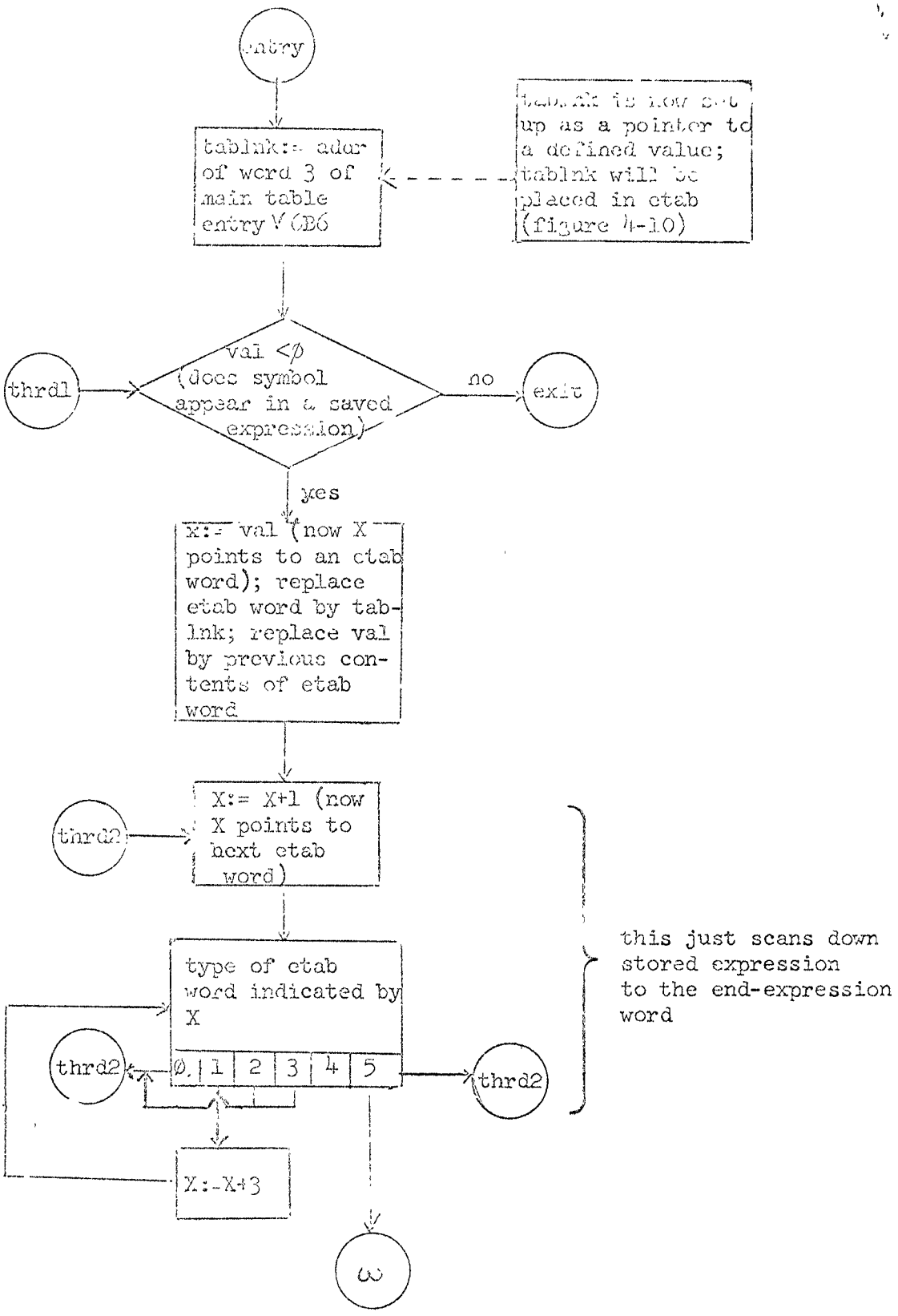


Figure 4-13: Flowchart of FOLEC (in CONTRL)
(cont'd)

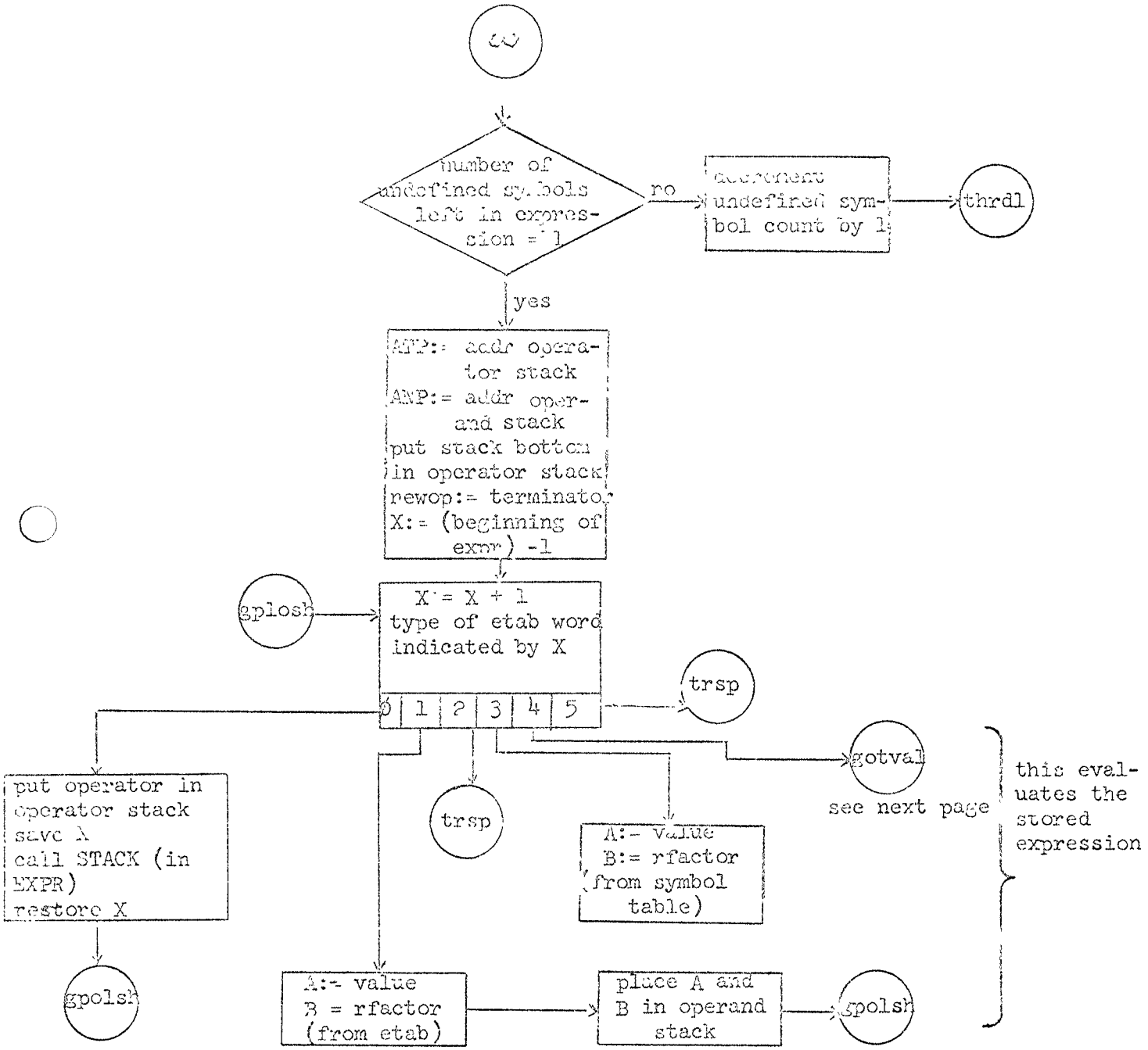
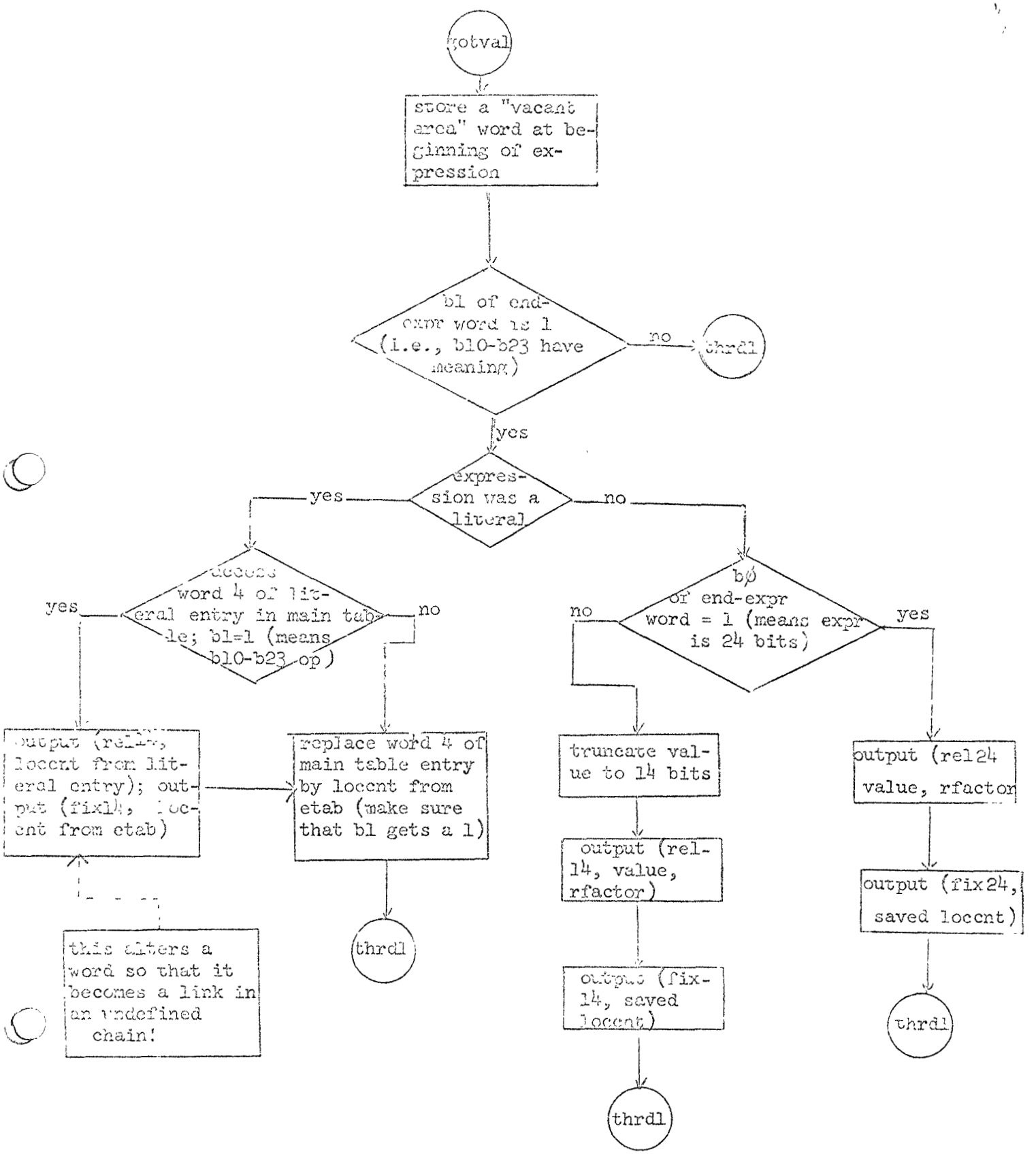


Figure 4-13: Flowchart of FOLEC (in CENTRL)

(cont'd)



The previous routines deal mostly with the actions when a symbol is defined. In contrast, the following routines handle references to undefined symbols and literals. At this point the scheme used for converting undefined expressions to Polish post-fix should be mentioned: just before EXPR applies an operator to its operands it checks that the operands are defined. If they are not, first the operand stack, and then the operator stack, are scanned (in the same direction as they were built up) and placed in the expression table. At the same time all operands in the operand stack are replaced by "place-holders" (see figure 4-8); these act as dummy operands to keep the format of the stacks in order. When an expression terminator is encountered and its action is taken, the undefined expression will be stored in the expression table in Polish post-fix.

The following flowcharts do not cover complete routines, but only parts of them:

EXPRI: Action when a symbol is delimited

EXPR: Action when an operator is to act on an undefined quantity

Action on a literal operator

Action when an expression terminator is encountered

END: Action on literals

Action on undefined expressions

Action on undefined symbols

Figure 4-14: EXPR: Action when a symbol is delimited

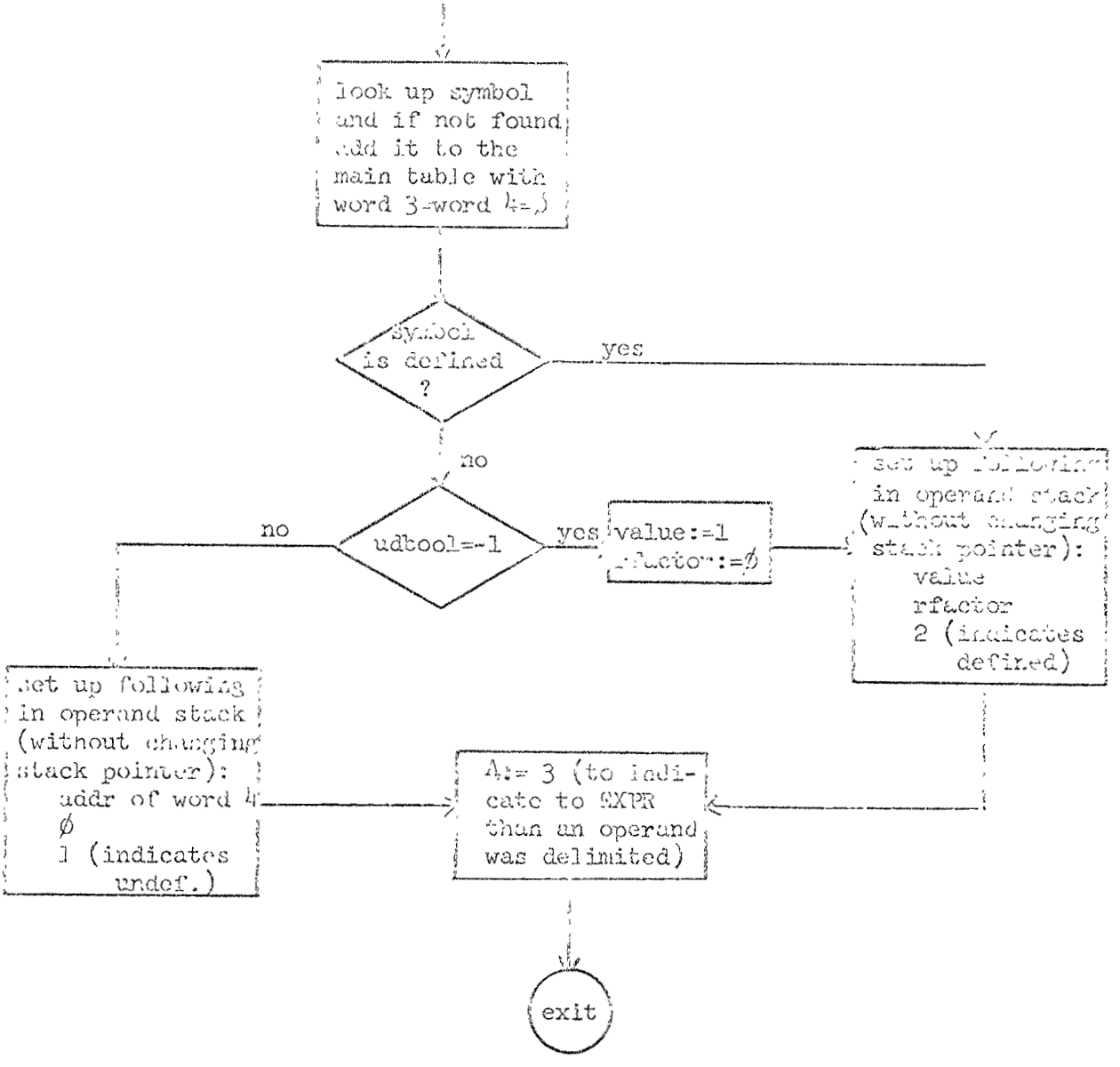
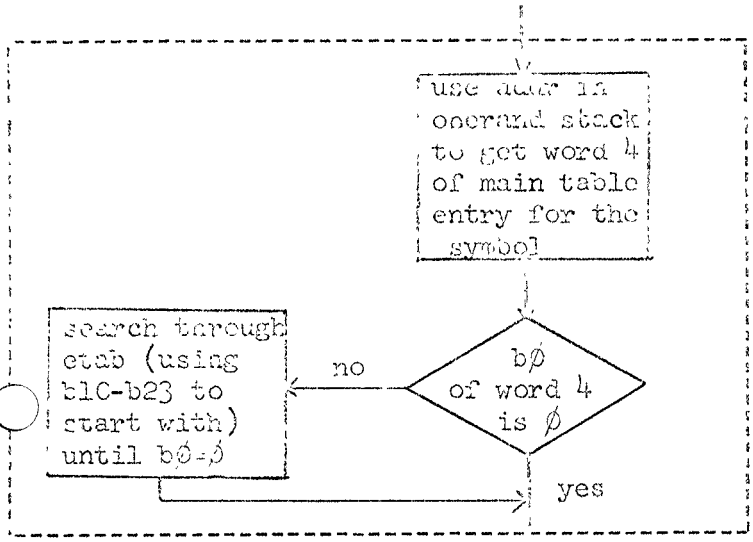
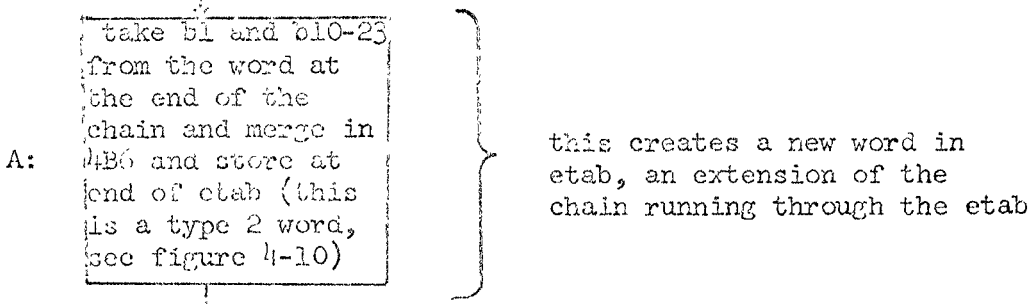


Figure 4-15: EXPR: Action when an operator is to act on an undefined quantity

The following action takes place when an undefined operand in the operand stack is to be stored in the expression table. The action enclosed in the dotted box is labelled because it is referred to in a later flowchart. (The flowchart below is part of the subroutine STASH in CENTRL.)





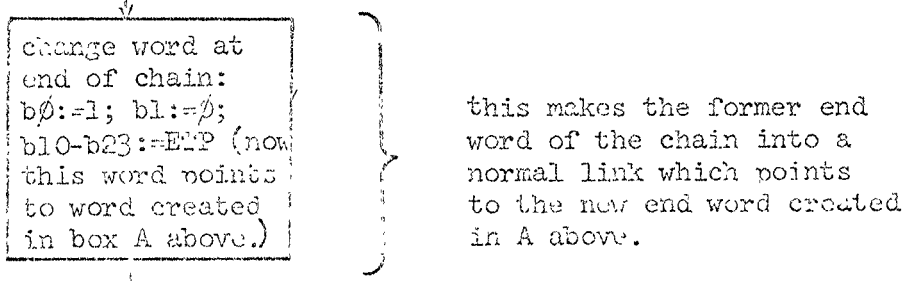


Figure 4-16: EXTR: Action on a literal operator

The flowchart below is of the subroutine EXTR in CENTRL. Note that if the operand of the literal operator is undefined then this subroutine is not executed; instead the literal goes in etab as for any other undefined expression.

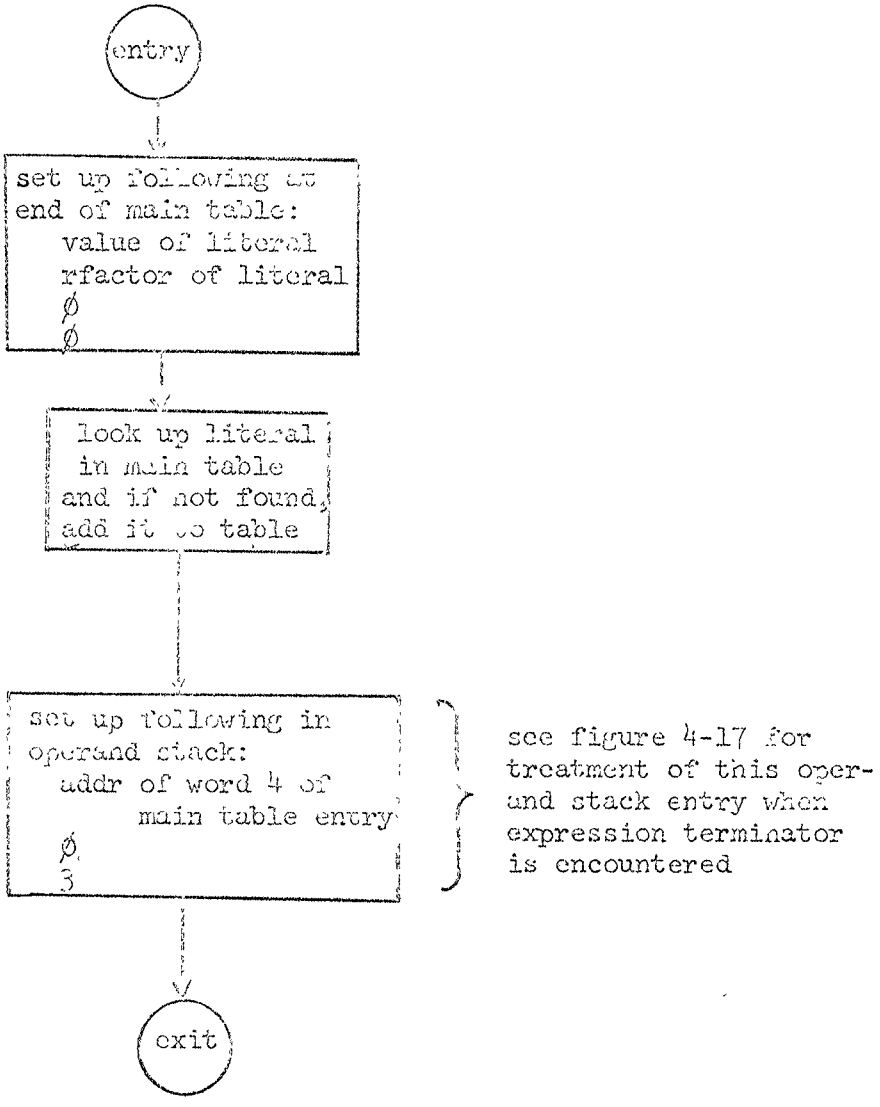


Figure 4-17: EXPR: Action when an expression terminator is encountered

The flowchart below is of that part of EXPR beginning just after the label EX.

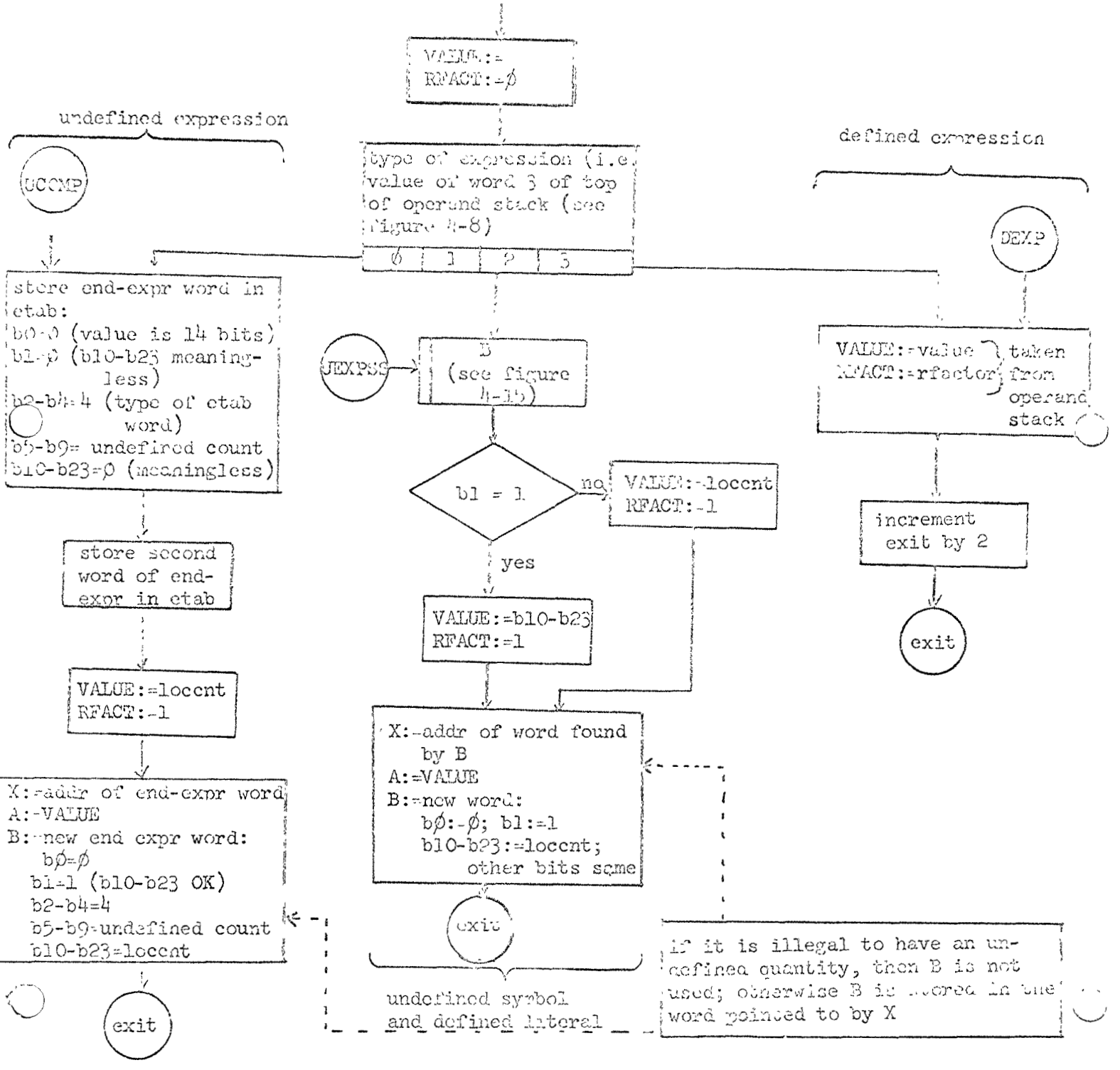


Figure 4-18: END: Action on literals

When END is encountered, the main table is scanned for literals, and the following action is taken on each literal (this is the routine LITERL in DIRECT).

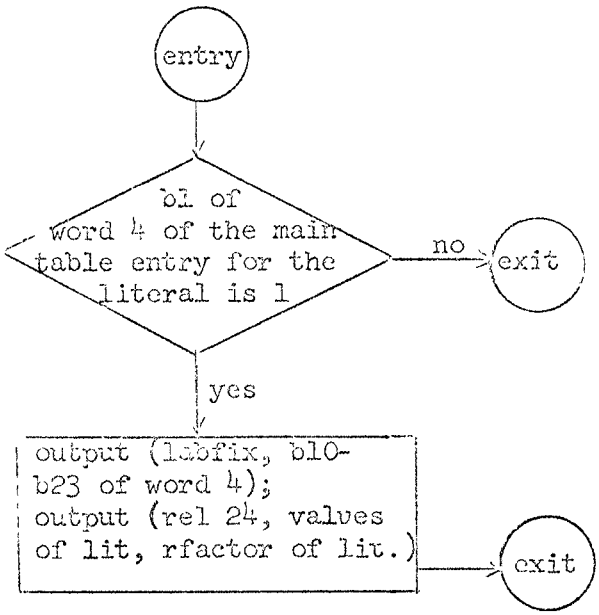


Figure 4-19: END: Action on undefined expressions

After the literals in the main table are output, the expression table is scanned. All expressions in this table are of the form ".X", where X is a single undefined symbol are output as literals; all other expressions in the table cause error messages. The following routine is executed for each symbol in the main table (this is the routine ETPROC in DIRECT).

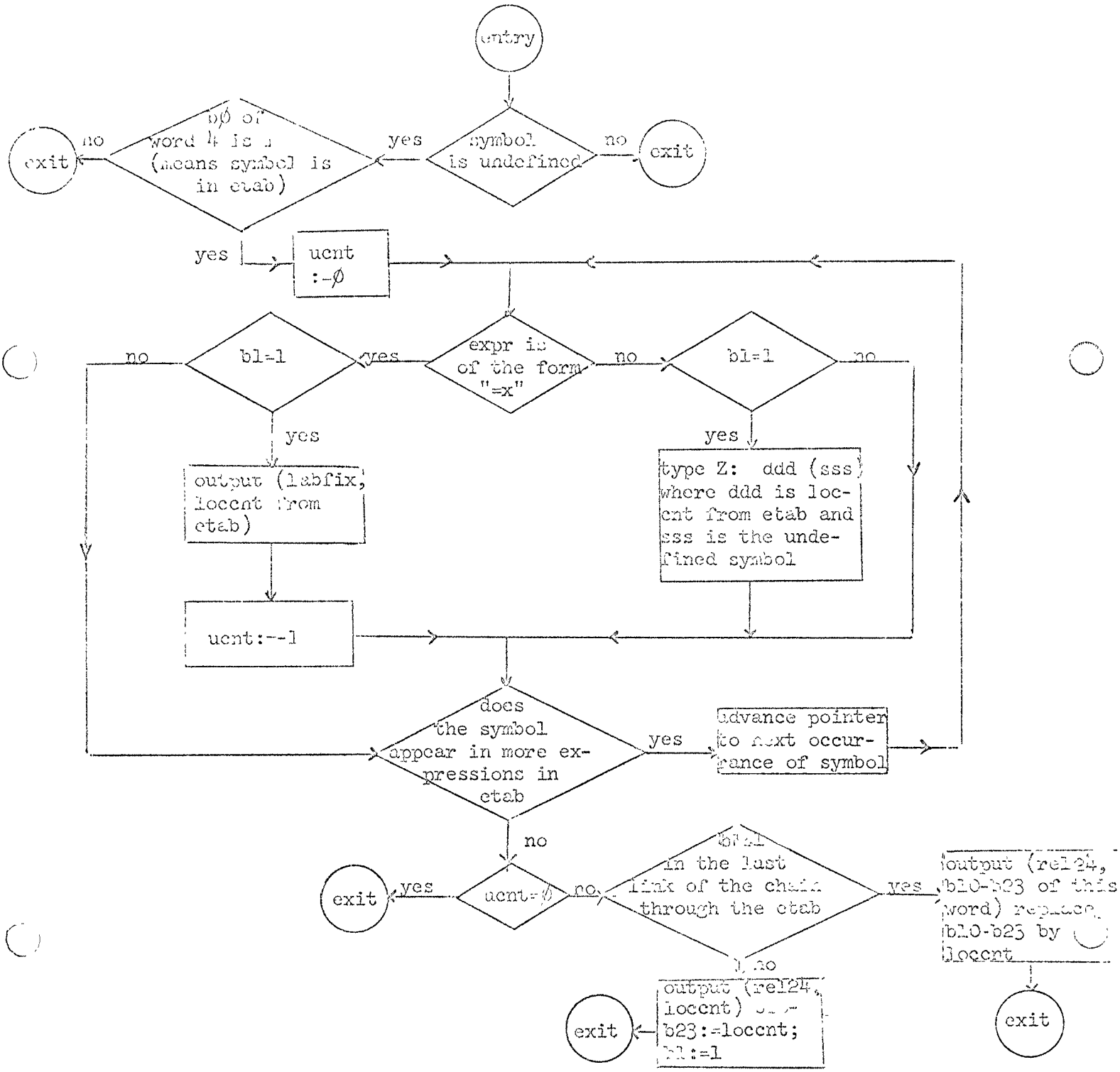
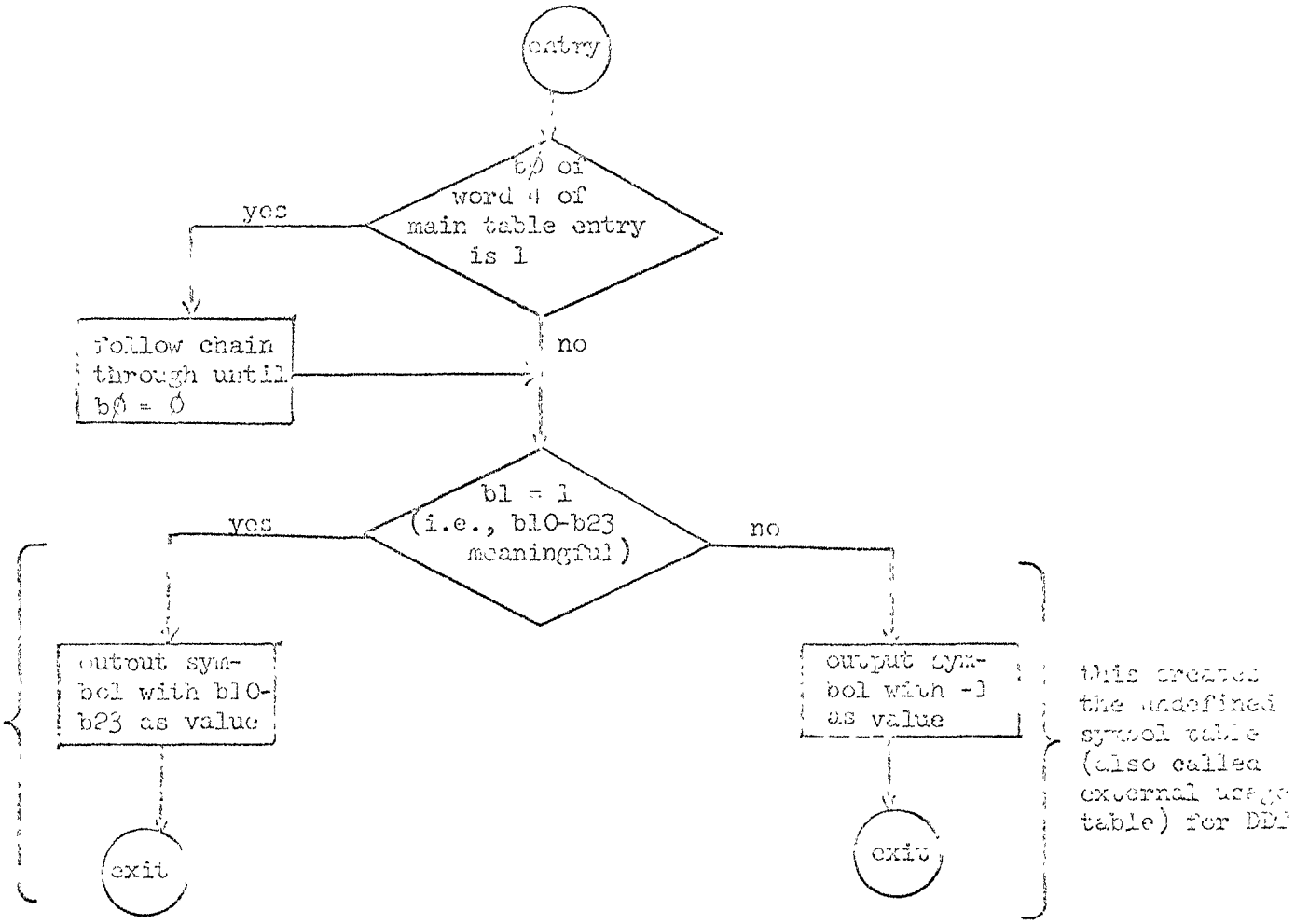


Figure 4-20: END: Action on undefined symbols

After the literals have been output and the expression table scanned, the main table is scanned for symbols. Each undefined symbol in the main table is processed by the following routine (USYM in DIRECT):



5.0 Statement processor (central loop of MARP), instructions, and non-macro directives

The central loop of MARP starts with the label CLOOP at the beginning of CENTRL. Following this label is a state table (STAB) of the same form as the one in EXPR, which checks the syntax of statements. In particular, the opcode of a statement is looked up and subsequent action is dependent on its type. The central loop processes a statement up to the operand field and then jumps to the appropriate action depending on the type of opcode (more details below). The routine jumped to then processes the operand field and scans up to a semi-colon or carriage return and then jumps back to CLOOP so that the next statement can be processed.

Just before the central loop jumps to a routine, it sets A as follows:

A=0 means there is no operand (if the opcode requires an operand then an error message is given by the central loop).

A=-1 means there is an operand waiting to be processed.

The routines jumped to take the following actions:

Instructions: The code for processing instructions is just after the central loop. This is rather straight-forward code, first checking for '←' and '/' and then calling EXPR to get the value of the operand. If there is a tag then EXPR is called again. The instruction is built up in INSTR and when a statement terminator is encountered it is output. (When the instruction processor is first entered, INSTR already contains the value of the opcode, possibly an indirect bit, etc., so only the values from the operand field need be merged in to complete the instruction.)

Macros: If the opcode is a macro call then a jump to MACALL in MAC is made; INSTR contains the address of the first word of the macro head (see figure 6-2) in string storage. More details of the processing of macro calls are given in section 6.

Directives: If the opcode is a directive then an ad hoc routine written specifically for the directive is jumped to. These routines are divided into non-macro directives (in DIRECT) and macro directives (in MAC), although some of the routines in MAC are not specifically connected with macros (like IF) but

involve string processing. The routines in EAS are described in section 6. Those in DIRECT are more or less straight-forward, and only a few are mentioned below. See the NARP listing for the others.

COPY: This is really a simple routine, but is mentioned because it uses a so-called short table (generated by the macro SHTAB). See the description of the routine SHTLK (in MISC) in the NARP listing for the format of these kinds of tables.

END: This directive is a little more complex than the others in DIRECT. There are two actions depending on whether NARP is being initialized (see section 7) or whether it is a normal assembly. In the latter case, several scans of the main table are made to output all the tables and other information needed by DDT.

EQU: This directive is of interest primarily because the code for handling equated symbols (routine ASCM, see figure 4-12) is located here.

ORG: The code may seem a bit obscure, but if it is followed with figure 4-2 at hand it should make sense. The action is slightly different if NARP is being initialized (see section 7).

6.0 Conditional assembly and macros

6.1 If statements (IF, ELSF, ELSE, ENDF)

Processing if statements is primarily a matter of deciding when to assemble MARP statements and when to skip them. The skipping is the central problem since it involves string handling, i.e., the statements to be skipped are looked upon as strings that are to be ignored until one of the patterns "IF", "ELSF", "ELSE", or "ENDF" appears in an appropriate place within a string. The skipping is done by a routine called PACKIT (described in more detail in section 6.4) which reads characters and switches on them to various actions. The usual action is to loop back and read the next character and switch on it. However, it is obvious that some of the actions must do a bit more or the ENDF closing off the IF will never be seen. Thus some actions change a mode, MMODE, which is used to detect the opcode field of a statement. The logic is rather simple, the first occurrence of a blank sets the mode to "after label field blank", and an "I" or an "E" encountered in this mode causes GNE to be called to delimit the symbol in the opcode field. After GNE returns, the delimited symbol is looked up in a short table containing the following:

IF
ELSF
ELSE
ENDF

If the symbol is found in this table then a special action is taken, otherwise PACKIT goes on switching on characters. The advantage of this scheme should be obvious: it is very fast since most of the time characters are simply skipped and GNE is called only rarely.

Following is the algorithm for processing if statements, written in pseudo-ALGOL.

```
procedure ifskp (elsf, else, endf); label elsf, else, endf;
begin comment  This procedure skips statements until it
    encounters one with ELSF, ELSE, or ENDF in the opcode field.
    It then exits to the corresponding label given as input.  Only
    ELSF, ELSE, and ENDF on the top level are considered, since
    all other occurrences belong to lower level IF-ENDF blocks
    (which are to be skipped);  mlev:=0;  comment mlev keeps track
```

of whether we are inside a lower level IF-ENDF block;
loop: PACKIT; comment This call will skip all statements until one occurs with "I" or "E" as the first character of the opcode field.; CNT; locate [illegible] and branch (ifx, elsfx, elsex, endfx, loop); comment if opcode is not one of the four special opcodes, the branch is to loop;
ifx: mlev: mlev +1; goto loop;
elsfx: goto if mlev =0 then elsf else loop;
elsex: goto if mlev =0 then else else loop;
endfx: mlev: = mlev -1; goto if mlev < 0 then endf else loop
end procedure ifskp;
comment The above procedure is not an exact replica of IFSKP in NARP since it shows some of the actions which really belong to PACKIT, but it conveys the main ideas. Following is the code for processing the directives IF, ELSF, ELSE, and ENDF.;

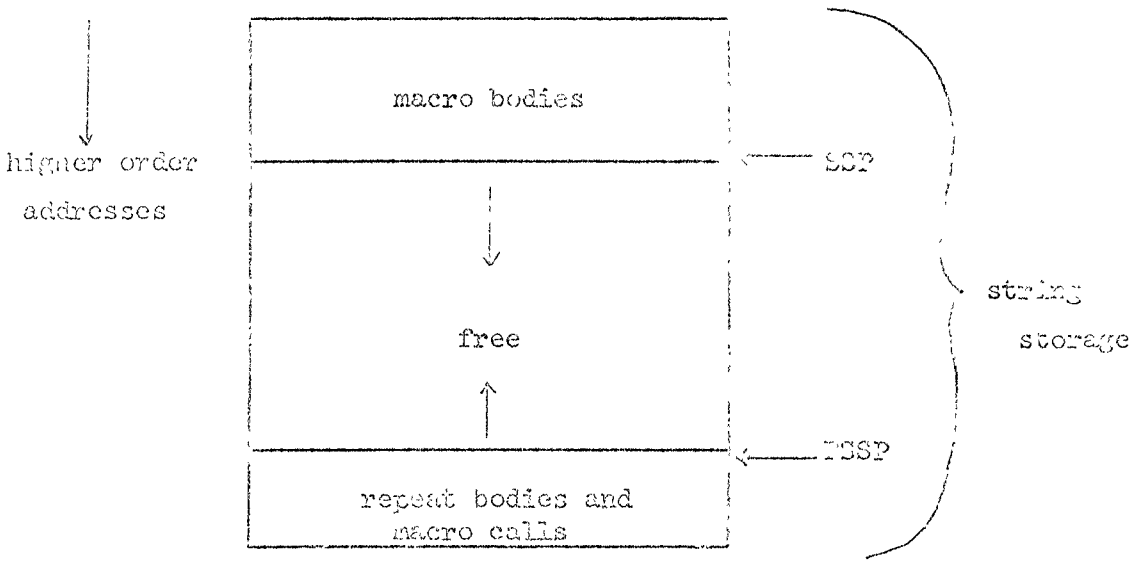
```
IF:     ifent: = ifent -1;  comment ifent is initially 0, so
        when it has a negative value an if statement is being
        processed;
ifl:    evaluate expression in operand field;
        if value of expression > 0 then goto CLOOP;  comment
        Thus the code immediately following the IF statement
        will be processed.  Note that the only thing which
        indicates that we are inside an if body is that
        ifent < 0.;
        ifskp     (ifl,   CLOOP,   endfl);
        comment     ELSF .   ELSE .    ENDF ;
ELSF:
ELSE:  if ifent= 0 then begin error ('S'); goto CLOOP; comment
        outside if body end;
elsel : ifskp (ELSF, elsel, endf);
        comment ELSF  ELSE  ENDF   ;

ENDF:   if ifent =0 then begin error ('S'); goto CLOOP; comment
        outside if body end;
endfl:  ifent: = ifent +1;  goto CLOOP;
```

Although it may not seem at first as if the above code can handle

if statements in their entirety, the reader may try a few examples to convince himself that indeed it does. The scheme may look simple, but it can handle if statements nested to any level, and yet, only one word of memory is needed to keep track of what is going on, namely ifcnt (note that -ifcnt is the level of the if statement being processed at any given moment).

6.2 Organization of string storage

Whereas if statements require only a certain amount of string handling the processing of repeat statements and macros is almost solely concerned with string handling. Things become more complicated because strings must be saved in memory for later use. The area of memory allotted for these strings (string storage or ss) is organized as follows:



The bounds of ss are kept track of by two pointers, SSP and RSSP, both of which always point to a word that contains useful information.

When a macro is defined (by MACRO and ENDM), the macro body is stored in the top part (i.e., lower order addresses) of ss, after the last macro body. SSP is advanced accordingly (SSP always gets larger as there is currently no garbage collection for macro bodies). On the other hand, when a repeat body or macro call (macro calls contain arguments which must be stored until the call is complete) is stored, the strings will only be needed for a short time, so they are stored at the bottom of ss. As soon as the repeat terminates or the macro call is finished, RSSP is restored to its previous value so that the repeat body or macro arguments are discarded. Thus RSSP gets both larger and smaller.

6.3 Repeat statements (RPT, CRPT, ENDR)

Handling repeat statements is done in two phases: storing the repeat body, and then letting NARP process the statements of the repeat body as many times as specified by the repeat head.

The main problem of the first phase is to find the ENDR matching the opening RPT or CRPT. The alert reader will notice that this is precisely the same problem as finding the ENDF to match an IF; the main difference is that the characters between are not skipped but are stored in ss. The routine PACKIT (now the name should have some mnemonic value for the reader) is used to store the characters in ss and to find the terminating ENDR.

Initially, the repeat is stored in the lower order addresses of ss, just after the last macro body, but as soon as the ENDR is encountered, the entire repeat is moved to the higher order addresses of ss and ESSP is altered accordingly. See figure 6-1 for the precise format of the repeat in ss. Of course, the controlled variable portion of the stored repeat is built up before PACKIT is called to find ENDR.

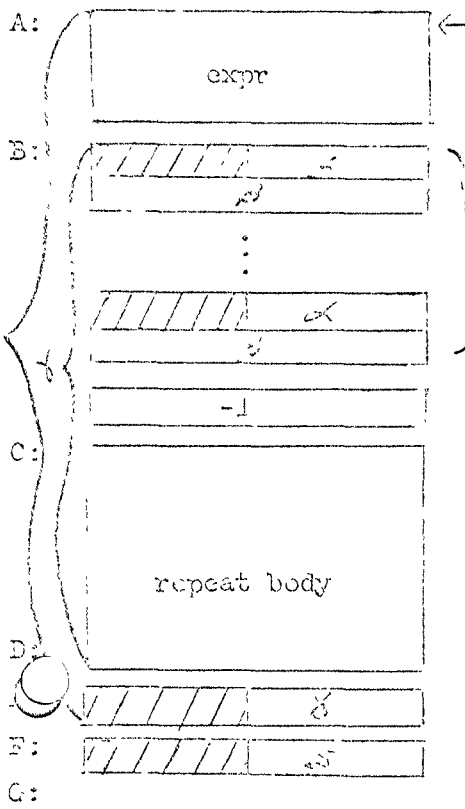
As soon as the repeat statement is all set up in ss as in figure 6-1, the expr area is examined (this may involve the evaluation of an expression) to see if the repeat body is to be processed at least once. If so, the current input pointer is stacked and then fixed up so it points to the first character of the repeat body. A jump to CLOOP is made and NARP is off processing the repeat body. When the special endrpt character (which replaces the symbol "ENDR" in ss) is encountered, the following action takes place:

ERPT: Use the input pointer (which contains address D; see figure 6-1) to locate V and thus find the beginning of the controlled variables.

Work through the controlled variable area, incrementing each symbol. At the end of this process address C is available, so stick it in STXT.

ERPT3: Use the input pointer to locate F and thus find the beginning of the expr area. Use the first character of this area to decide what action to take:

Figure 6-1: Format of a repeat body



expr: In general this is a string of characters, the first character indicating the kind of expr:

char	value	meaning
⊥	∅	arith expr: expr is only one word; the remaining 16 bits form a binary number, a count that is decremented each time the repeat body is processed until it becomes zero.
!	1	ev expr: expr is only one word; the remaining 16 bits form a binary number, a bound on the first controlled variable; the repeat body is processed until the value of the first controlled variable passes this bound (due to a syntax error there may be no controlled variables).
"	2	expr: the following characters are an expression terminating in a carriage return (expr is then a variable number of words).

controlled variables: This area corresponds to the increment list of a repeat statement; it may be empty.

α: address of the fourth word of a main table entry for a symbol

β: increment to be added to the symbol indicated by α each time repeat body is processed

repeat body: This is the string of characters comprising the repeat body. The last character of the string is always a special endrpt character (value 204B); it is always preceded by a carriage return or a semi-colon.

γ: Count of words from first word of controlled variable area through last word of repeat body. This count is used to find the controlled variables when the last character of the repeat body has been processed.

δ: Count of words from first word of expr through δ word. This count is used to find the beginning of the entire stored repeat when the controlled variables have all been incremented.

BSSP is shown with the value it has just after the repeat is stored; it may take on smaller values during the processing of the repeat body.

ϕ : goto SKMP
1 : goto BOUND
2 : goto EVALU8

SKMP: Subtract 1 from count and leave count in A; goto ERPT4

BOUND: Check if first controlled variable is past bound and leave ϕ in A if so, otherwise / ϕ in A; goto ERPT4

EVALU8: Evaluate expression and leave value in A;

ERPT4: if A / ϕ then set up input pointer to beginning of repeat body (using STEXT) and goto CLOOP else change BSSP to point to C, get old input pointer from stack, and goto CLOOP;

By trying an example the reader should convince himself that this scheme handles nested repeats, as well as macro calls within repeats and vice versa. The only information that need be saved (outside the ss) to make these nested structures work is the current input pointer (see section 3-1).

6.2 Storing strings (PACKIT routine)

In the preceding discussion the routine PACKIT has been mentioned several times and even described to a certain extent, in particular in section 6-1. However this description was oriented to the use of PACKIT in processing IF statements, whereas most of the features of PACKIT are used for processing repeat statements and macro definitions.

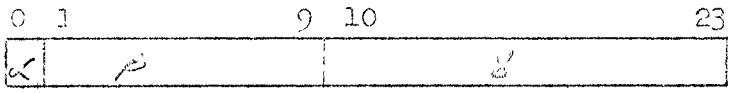
The input to PACKIT is implicit:

1.) The input pointer is positioned at the beginning of a statement.

2.) SOMTAB contains the address of a short table.

3.) DTAB has been modified.

(The last two lines above will be clarified shortly.) When entered, PACKIT reads characters and dispatches on them via DTAB. The format of an entry in DTAB is shown here:



α : $\emptyset$ means non-alphanumeric character, 1 means alphanumeric

β : An integer used by SSNSS (described in section 6.5)

γ : Address of a piece of code in PACKIT

See the NARP listing (in MAC) for the specific DTAB entries. This table is in MAC because in creating it, arithmetic is performed on labels defined in MAC. As soon as NARP is started, however, the table is moved to alterable memory (in TEMP) because some entries in DTAB are changed from time to time.

For most characters, the piece of code indicated by γ is PACKM, which simply packs the character into string storage. (Note: The basic packing routine is PACK, an alterable routine located in TEMP after the basic input routines; when IF is using PACKIT, the routine PACK is altered so that it doesn't pack any characters.) As described in section 6.1, a rather simple mode mechanism indicates whether the opcode field is being processed or not. Now, inputs 2.) and 3.) mentioned above can be clarified:

2.) The short table indicated by SCRTAB containing the symbols which are of interest to the routine calling PACKIT:

if statements	repeat statements	macro definitions
IF	REP	MACRO
ELSF	CRPT	END
ELSE	ENDR	dummy argument
ENDF		generated

For macro definitions, the dummy argument and generated may be different for each macro; this table is in alterable memory because it is, of course, changed depending on the macro.

3.) SCRTAB is modified so that the first characters of the symbols in the above tables cause dispatches to special pieces of code:

if statements	repeat statements	macro definitions
I	R	M
E	C	E
	E	d
		g

Except for "d" and "g" the dispatch is to SPECL in PACKIT which calls CMI (if MMODE is right) and does a lookup in the table indicated by SCRTAB. If the lookup is successful, a branch is made to a piece of code indicated by the table. For "d" and "g", MMODE is not interrogated, but instead a check is made to see if the preceding character was alphanumeric. If not, CMI is called, etc.

The mechanisms described above are designed essentially to enable a scanning algorithm that looks at one character at a time to look occasionally at a wider universe, namely a symbol. However there are lesser tasks to be considered:

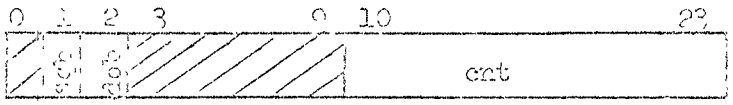
"($" is converted to a special character

".&" is removed (after changing ATTAN to ϕ)

blanks are compressed.

The first two tasks mentioned above are rather straight-forward and can be easily followed in the MAP listing. The last one however requires some comment since the conventions about blanks between quotes and between parentheses are a bit complex (see the

NARP reference manual). The decision on whether to compress blanks or not is made by interrogating BRAKB, which has the following structure:



sqb: single quote boolean: 1 when between single quotes

dqb: double quote boolean: 1 when between double quotes

cnt: parentheses count: > 0 means between parentheses, value is level of nesting

If BRAKB is zero (normal state) then SQ%135 should be negative so that GNLC compresses blanks.

6.5 Macros (MACRO, ENDM, macro calls)

The directive MACRO signals the beginning of a macro definition. The symbol in the label field is stored in the main table as an opcode, its value being the address of the macro in string storage. If a dummy symbol and a generated symbol appear, they are stored in the table MACT (in DTAB) and then PACKIT is called to pack away the macro body in ss and to find the closing ENDM. See figure 6-2 for the format of a macro as stored in ss.

The interesting part of macro processing is macro expansion. When a macro is called, the arguments for the call are stored in ss along with pointers to them, the number of arguments, information about generated symbols, and a pointer to the enclosing macro call (if there is one). See figures 6-3 and 6-4. As for repeat bodies, the arguments are first stored just after the last macro body in ss, and after they have all been read in, are moved to the bottom of ss (higher order addresses). The routine for reading arguments from the source and storing them in ss is SSNSS. It reads one character at a time and switches on DTAB (using a different part of a DTAB word than PACKIT). The logic of SSNSS is rather simple, its main task is to find an argument terminator; see the NARP listing.

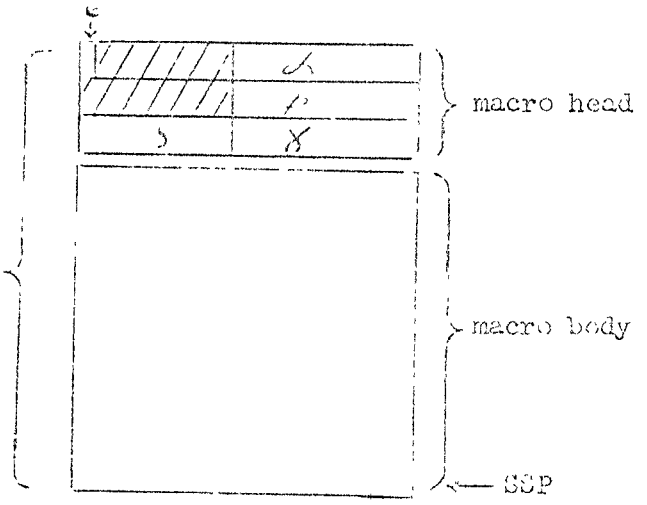
Once the arguments have been collected, the current input pointer is stacked and then altered to point to the first character of the macro body. A jump to CLOOP is then made and NARP is off processing the macro. During this processing the directives NARG and NCHR may appear. These are handled quite simply:

NARG: The word ARGACC always points to the current macro call (it is $\emptyset$ if NARP is not expanding a macro), where the number of arguments is stored.

NCHR: The routine SSNSS is called, since an operand for NACR looks exactly like a macro argument. Note that this directive can also appear outside macros.

Of more interest are the special characters which may be encountered during macro expansion; they are of course stored by PACKIT during macro definition:

Figure 6-2: Format of a stored macro body



macro head:

α: address of fourth word of a main table entry for a macro opcode.

β: count of words used by macro head and macro body

γ: base of generated symbols (initial value is $-\delta$).

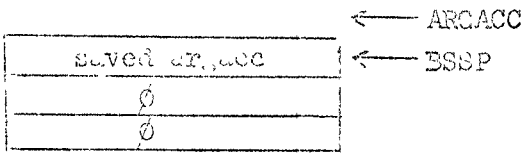
δ: increment to add to γ

ϵ: 0 - macro is active; 1 - macro is garbage

macro body: This is the string of characters comprising the macro body. The last character of the string is always a special endmac character (value 203B); it is always preceded by a carriage return or a semi-colon.

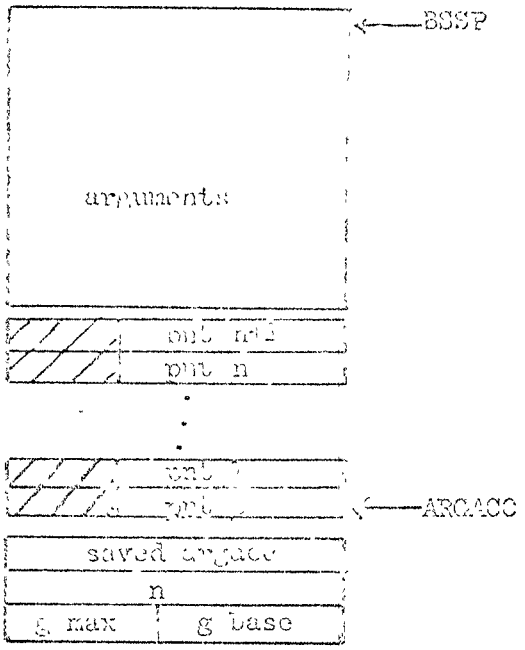
(Note: The elements α, β, and ϵ in the macro head are to be used by a garbage collector which currently does not exist.)

Figure 6-3: Format of a call of a macro with no dummy argument



The only piece of information that actually needs to be saved in this case is ARGACC; however, two dummy words are introduced so that the format is compatible with a call of a macro with a dummy argument (see figure 6-4).

Figure 6-4: Format of a call of a macro with dummy arguments



arguments: Each argument is stored as "(<arg string>),", where <arg string> is the argument as given in the source program but with parentheses removed. Thus ARG(A, (X + Y)) is stored as

(A),
((X + Y)),

Another way to say it is that if an argument is given in the source program has no enclosing parentheses then they are supplied before the argument is stored. ---- Note that the arguments are stored one after the other with no gaps. Null strings are stored as "(),". Argument ∅ (the label field) always is stored.

pointers: Each argument is pointed to by a pointer, a 16-bit character address which points at the opening left parenthesis of the argument. Note that there is an extra pointer (pnt n+1) which would point to argument n+1 if there were one more argument.

saved argacc: This is the previous contents of ARGACC. When the macro call is completed, this value will be placed back in ARGACC so that it points to a higher level macro call.

n: This is the number of arguments supplied by the macro call ($n \geq 0$).

g max: This is the maximum value that a subscript of a generated symbol may have ($1 \leq g\ max \leq 1023$). This value is the same as y in figure 6-2.

g base: This is the base value for generated symbols for this call. Thus the generated symbol G(2) will result in G∅x, where x is a digit string representing the value of g base + 2.

(illegible) becomes (illegible) where (illegible)
D(ABC) becomes (illegible) where (illegible)
C(ABC) becomes (illegible) where (illegible)
DMD: becomes $ where (illegible)

When one of the above characters is encountered, a branch is made to the appropriate routine in MAC:

DIG$ ⟶ goto DIG
DUMDUM ⟶ goto DUM
GENRAT ⟶ goto GEN
ENDMAC ⟶ goto EMAC

The actions of DIG and GEN are very similar: the following expression is evaluated and then converted to a digit string which is stored in an array in TEMP. The input pointer is stacked and then altered to take input from this array. —— The routine EMAC is even simpler; it uses the saved argacc (see figures 6-3 and 6-4) to restore ARGACC to its former value, resets DSCP so the macro call disappears from ss, and unstacks the input pointer. —— Of the special character routines, DUM does the most work, primarily because it must worry about all the special notations for dummy arguments (see the NARP reference manual). See figure 6-5 for a flowchart of the main action of DUM: setting DPNTR to the address of the first character of the argument or piece of argument that is specified, and setting DUMLN to the length of the character string specified.

One fine point whould be mentioned: Sometimes when one of the special characters is encountered, it should not cause a special action but should simply be passed on. Here is an example:

```
A   MACRO   D
B   NARG
    RPT     (I=1,B)
    DATA    D(I)
    ENDR
    ENDM
```

When the macro body is stored, the string "D(" inside the repeat body will be converted to $, where $= DUMDUM. Now, when the macro is expanded, the repeat will be encountered, and its repeat body

will be stored in the bottom of ss. During this storing, & will be encountered. If at this point the routine DOM is executed (i.e., the expression "I" is evaluated) then the repeat will be stored as

```
REPT    (I-1, B)
DATA    XYZ
ENDR
```

assuming that D(1) = "XYZ". Obviously this is not what is wanted. To achieve the desired result, & is simply stored with the repeat body and no expression evaluation is done. The decision of whether to process & or simply pass it on (here & may be CENTAT or DIOS as well as DUMDUM) is made by interrogating PXPAT. It may have one of the following values:

-1 normal (i.e., not storing anything, so process &)
0 storing a macro
1 storing a repeat or skipping inside an IF statement

The decision is made according to the table below:

special symbol / process	DUMDUM	CENTAT	DIOS
storing a macro body	expand	expand	store
storing a repeat body	store	store	store
skipping inside an if statement	store	store	store
normal	expand	expand	expand

Consider the following string from some macro body:

```
ADD     ABC+DEF.&D(IJ+1)
```

During expansion when the & replacing "D(" is encountered, the expression "IJ+1" will be evaluated. This will involve calls of

Figure 6-5: DSM: Flowchart for dsl' ... an array ref.

(See MACRO listing for description of the subroutines DSUP and ARGBC.)

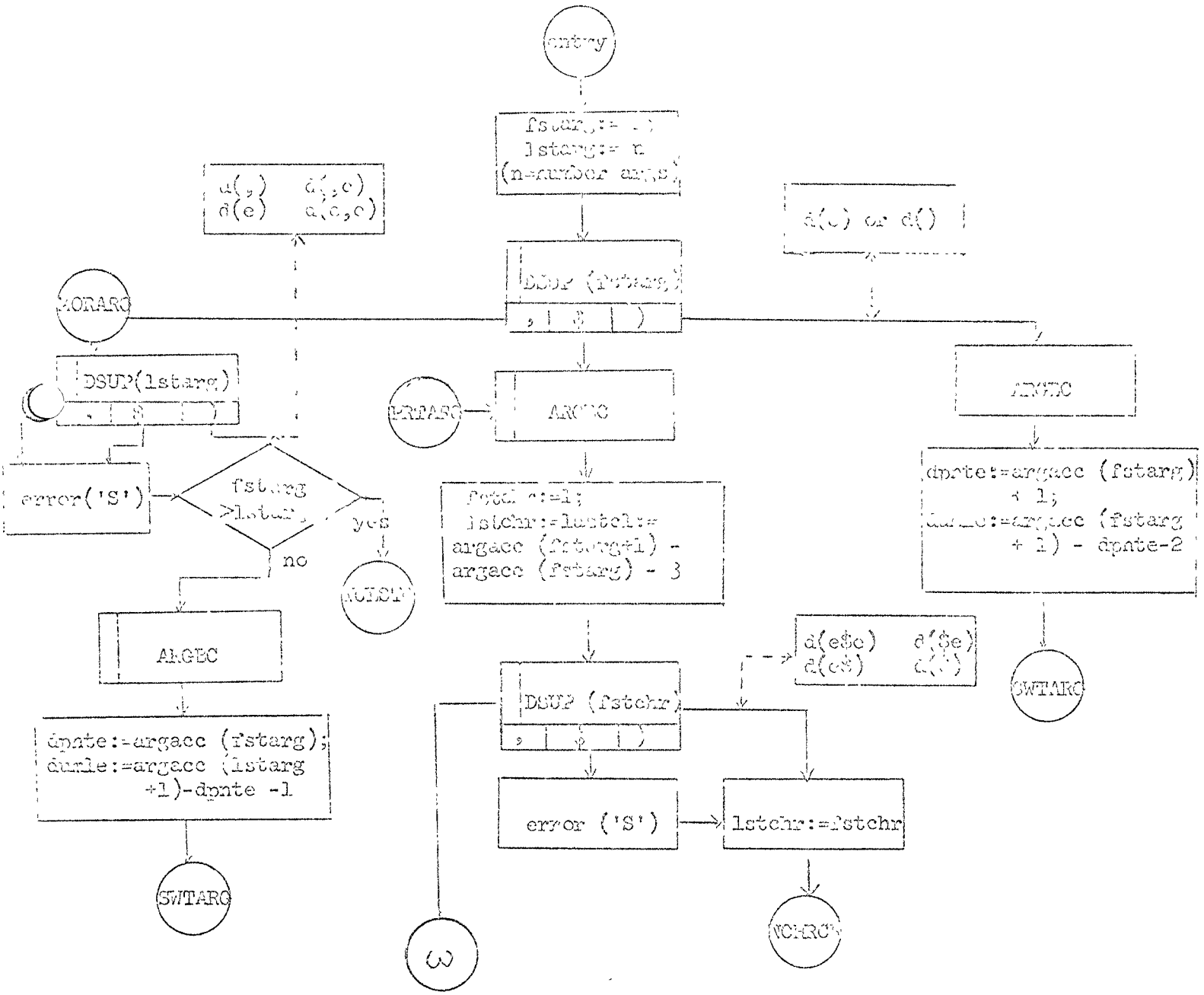
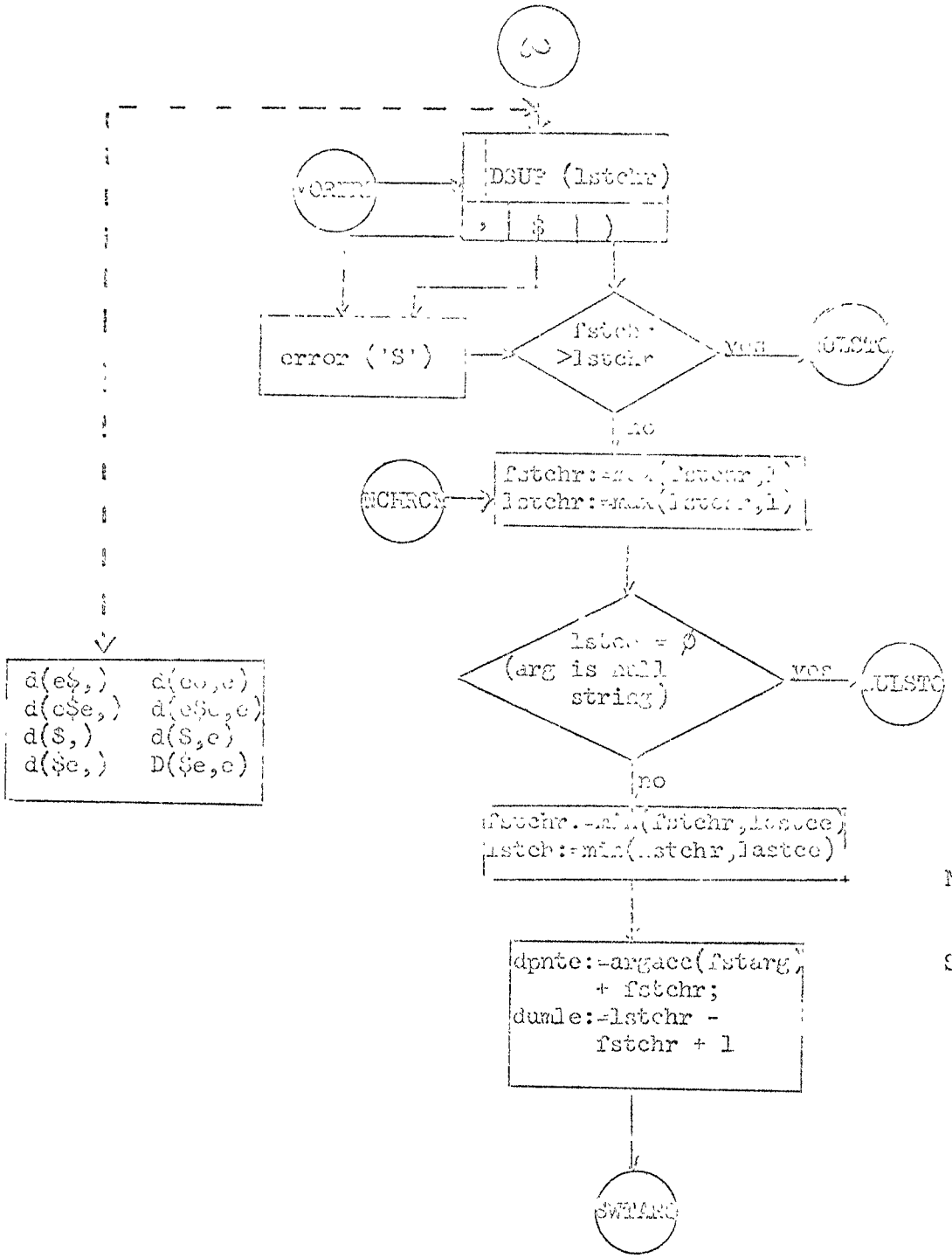


Figure 6-5: DUM: Flowchart for delimiting an argument (cont'd)



NULSTG: No change of input source.

SWTARG: Switch input source to an argument.

CAT and EXPR. However, at the moment & is encountered, CAT is busy delimiting a symbol (the first part of which is "DEF") and EXPR is busy evaluating an expression (the first part of which is "ABC+DEF"). Thus CAT and EXPR must be recursive subroutines. The character stack in CAT and the operand and operator stacks in EXPR work fine during recursion, but all the working cells of these routines, as well as their return addresses must be saved somewhere. This is the function of the pile: whenever DCM, GEN, or DIG is entered, all subroutine working and work cells that must be preserved (of course, routines other than EXPR and CAT must be considered) are placed on the pile by PSH (in MISC); upon exit from these routines, CRA (also in MISC) takes information off the pile and places it in the appropriate cells. This scheme also enables constructs like D(ABC+D(2)) to be processed, since DCM, GEN, and DIG are recursive.

The reader should note that enough information is stored with a macro call so that macro calls within macro calls are handled quite naturally. The key to this is the saving of ARGACC with the macro call; thus when one call is finished, ARGACC is restored to its value before the call, which may mean that it points to a higher level call. This same scheme also handles recursive macro calls, with absolutely no extra machinery added. The reader should try a few examples; as an exercise hand-assemble the following program, observing how ss changes:

```
A     MACRO     D
      IF        D(1)='STOP'
      DATA      D(2)
      ELSE

      RPT       2
      DATA      D(2)
      A         'STOP',ALPHA
      ENDR

      ENDIF
      ENDM

      A         'GO',BETA

      END
```

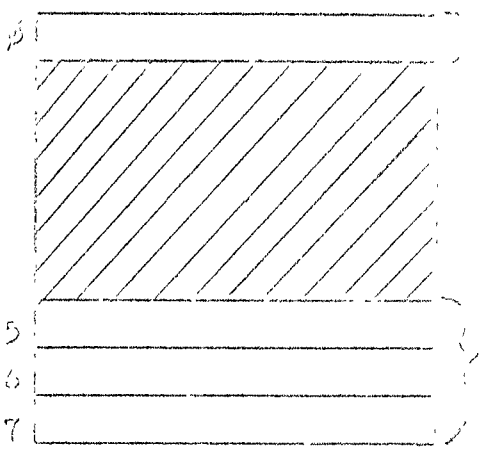
The generated code is

```
DATA    [illegible]
DATA    [illegible]
DATA    [illegible]
DATA    [illegible]
```

7.0 Initializing, storing and co[illegible] NARP (cont. from page 7-2)

After the symbolic version of NARP has been assembled, it is loaded by DDT, core looks like

image I



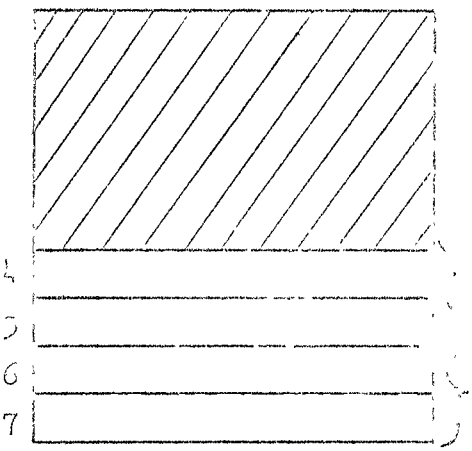
first block of working area (reserved), containing allocated routines and initialized terms

read-only code of NARP

At this point, none of the pre-defined opcodes (XWD, MOVE, etc.) are in the main table, except for OPD. In order to define these opcodes, NARP should be started at INIT (see pages 7-1 and 7-2 for details), and a special program containing definitions of all the pre-defined opcodes and pre-defined symbols should be assembled. At the end of this assembly, block 0 will contain all the definitions.

If TSS could handle sub-systems which consisted partly of read-write blocks and partly of read-only blocks, core would be all set up for creating the sub-system NARP. However, since TSS only handles read-only blocks, it is necessary to move block 0 to block 4 before placing NARP on the drum as a sub-system. This is done automatically at the end of the assembly of the initialization program, so core looks like

image II



first block of working area; now contains all pre-definitions.

read-only code of NARP

This is the core image that goes on the drum as the NARP subsystem; of course all four blocks are read-only.

Now, when a user calls NARP, core image II is brought into memory, and NARP is started at START, which is the location in block 5. This code moves block 4 back to block ϕ in a BLT loop (so block ϕ is read-write) and then changes the relabelling so block 4 is thrown away. Now core looks like image I above, except that block ϕ contains all predefinitions. The specified source file can now be assembled.

The processing of the directive FREEZE is intimately tied in with initializing and starting NARP. Every initialization program should end with a FREEZE so that when NARP is started it always assumes that it is entered at the COMMON entry (the only difference between the START entry and the COMMON entry is that the former must move block 4 to block ϕ). There are two pieces of information that must be saved to implement FREEZE: the values of initialized temps (which may change during one assembly, and must therefore be restored before the next assembly), and pointers to the main table and string storage. The first piece of information is saved just once when NARP is initialized; the second piece is saved every time FREEZE appears. Both pieces of information are restored every time NARP is started or continued (remember that before the START or COMMON entry of NARP is used, the INIT entry has been used, and FREEZE has occurred at the end of the initialization program).

Figures 7-1 and 7-2 should give enough details about the processes described above. The file opening logic may seem a bit obscure, but the idea behind it is to first collect the names of all the files that are to be opened and then open them in the order they were given. Thus, a user can type in the file names and walk away from his teletype while the files are opened (which can take time if the tape must be moved) and the assembly is done. The file opening code will have to be changed somewhat to implement the description of how to start NARP as given in the reference manual.

See section 8 for some comments on how the initialization scheme will have to be changed to accommodate pre-defined macros.

Figure 7-1: Flowchart for initializing [illegible]

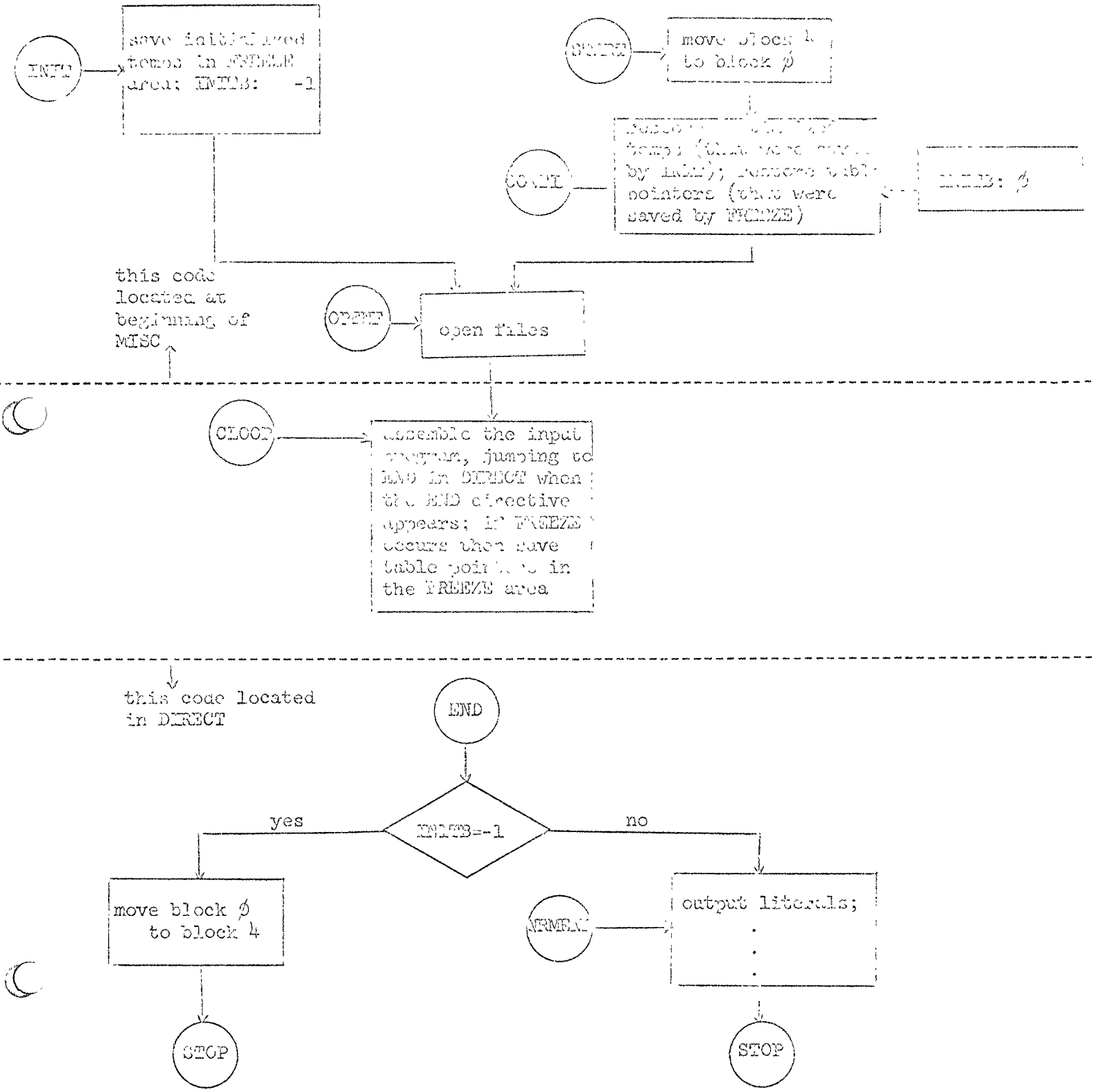


Figure 7-2: Algorithm for initializing, starting, and continuing MTP

```
INIT:     move block 4 to MTP;
          check block to see that table contains [illegible];
          initialize the upper bound of the pibs (pib [illegible] length;
          see section 7.3); save initialized temps in initial area,
          goto START; for the INITD is -1;

START:    move block 4 to block ∅;
          get rid of block 4;

CONTIN:   restore initialized temps that were saved by [illegible];
          restore table pointers that were saved by FREEZE;
          combine free area in [illegible],
          scan main table, setting word 4 of all literal entries to zero;
          scan user table, setting word 4 of all undefined symbol entries
               to zero, as well as removing a mark left over from listing
               the undefined symbols in alphabetical order;
          INITD := 0;
          goto CHKMT;

FREEZE:   save table pointers in FREEZE area;
          goto CLOOP;

END;      if INITD ≥ 0 then goto NRMEND;
          check MYCONF and MTP to see if initialization program was okay;
          get rid of all blocks between block ∅ and first read-only block;
          close all files;
          type "init complete";
          move block ∅ to block 4;
          brs 10;

NRMEND:   output literals;
               :
               :
          brs 10;
```

The following macros are used by the above routines:

saving and restoring initialized temps:

TMOVE : moves the initialized temps defined by the macro TI.

ITMOVE : moves the initialized words that do not lie in a contiguous block but are spread throughout the working area.

saving and restoring table pointers:

IRMVE : moves initial reference tables, MTP, SSP, and WD1-WD4.

8.0 Peculiarities, bugs and suggested improvements

Following is a list of miscellaneous notes; what it is not arranged in any particular order.

1.) It would be nice to be able to have more than one MAP program in the same symbolic file and have MAP1 call MAP2 or vice versa, the other with no user intervention. To implement this, before closing files in END, read the next character and check if it is 137B. If not then assume it is the first character of a new program and assemble it.

2.) Introduce a new directive which would allow the user to change the size of the tables used by MAP (hash table, SS, etc. This in conjunction with the STAT option of MAP1, (see reference manual) which prints how full the tables are, would allow a programmer to adjust the tables to suit him. In conjunction with this, a more elaborate message should be printed when a table overflows so the programmer knows which table it was.

3.) The bit in the symbol table to designate that a symbol is generated is superfluous; the forget bit can be used. Also, ASGN should interrogate CNBCOL and set the generated (or forget bit) accordingly. (Right now C(3) EQU 1 will not be marked as generated.)

4.) An empty macro body or repeat body will cause MAP to go wild. This is because the routine EXSP always expects to find a carriage return or semi-colon. To correct the problem, pack a carriage return in SS before calling PACKTE, and then change things so the input pointer points to the second, not the first, character of a macro body or repeat body.

5.) Throw out the directives DEC and OCT and make a more general directive RADIX which allows the number radix to be set to 2, 3, ..., 10.

6.) Perhaps EXPR should recognize =x, where x is a single undefined symbol, as a special case instead of just sticking it in the undefined expression table. The reason for this is that every occurrance of =x, where x is external, will be stuck in the expression table.

7.) Because NARP doesn't keep track of whether a [illegible] symbol is used as a 24-bit value and when it is used as a 24-bit value, all fix-ups of undefined chains must be done [illegible] by DDT. If the programmer writes something like the following, he will get into trouble:

			After DDT finishes loading:
	DATA	X	[illegible]
	ADD	X	MQA 077777B
X	FOO	[illegible]	
	END		

The trouble, of course, is that the leftmost 18 bits of ADD X get modified, so ADD becomes MQA.

8.) The pre-defined symbol :X'':0: would be more aesthetically pleasing if it were written as :0:.

9.) The action of the relational operators "=" and "#" should be changed so that their operands do not have to have the same relocation factor, and thus not only are the values of the operands compared for equality, but the relocation factors too. Then inside a macro, a statement like

```
IF      B(T)=0
```

will be legal (and will be the right thing), no matter whether B(T) is absolute or relocatable. At present, if B(T) is relocatable then the error message "?" is typed. This can be quite annoying.

10.) The OPD directive has provisions for allowing a macro to be defined while processing an initialization program. This should be removed (i.e., so that OPD can only define instructions and directives) since macro names are, of course, defined by MACRO. (The option exists only because of a moment of fuzzy thinking on the author's part.)

11.) When pre-defined macros are added to the initialization program, NARP will have to be changed slightly. Right now, it is assumed that there is only one block of initialized stuff, namely block 0. Since string storage is in another block, adding pre-defined macros will create more than one block of initialized stuff. This means NARP will need more than the four blocks it now occupies on the drum.

12.) A garbage collection scheme [illegible] table should be considered. [illegible] in the table is [illegible] least [illegible] should be set back. [illegible] collection in mind. When a [illegible] code for [illegible] DATA will have [illegible] that if WARP does not [illegible] the [illegible] operation was not [illegible]

13.) WARP is not prepared to handle [illegible]:

```
        M     MACRO    ...    ENDM
              PRINT
        M     MACRO    ...    ENDM
              END
            @CONTINUE WARP.

              M
              END
```

The opcode M will point to nonsense in the string storage when the second package is assembled. There is no telling what the call of M will do.

14.) A garbage collection scheme for the string storage should be coded (see routine [illegible]). Of course, we need only garbage collect macro bodies, not repeat bodies. See [illegible] in listing for some warnings. Also, remember to adjust the pointers in the input pointer stack after a garbage collection. The format of a macro body in ss (see figure 6-2) has been made to make garbage collection relatively easy.