DECUS
PROGRAM LIBRARY

DECUS NO.	10-118
TITLE	BLISS REFERENCE MANUAL (A Basic Language for Implementation of System Software for the PDP-10)
AUTHOR	W. A. Wulf, D. Russell, A. N. Habermann, C. Geschke, J. Apperson, D. Wile, R. Brender*
COMPANY	Computer Science Department Carnegie-Mellon University Pittsburgh, Pennsylvania
DATE	January 15, 1970 (Revised August 15, 1970) (Revised November 9, 1970) (Revised April 7, 1971)
SOURCE LANGUAGE	BLISS

* Digital Equipment Corporation, Maynard, Mass. 01754

001-326-002-01

BLISS REFERENCE MANUAL

A Basic Language for Implementation of System Software for the PDP-10

W. A. Wulf
D. Russell
A. N. Habermann
C. Geschke
J. Apperson
D. Wile
R. Brender*

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

January 15, 1970

(Revised August 15, 1970)
(Revised November 9, 1970)
(Revised April 7, 1971)

*Digital Equipment Corporation, Maynard, Mass. 01754

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

PREFACE

This manual is a definitive description of the BLISS language as implemented for the PDP-10. BLISS is a language specifically designed for writing software systems such as compilers and operating systems for the PDP-10. While much of the language is relatively "machine independent" and could be implemented on another machine, the PDP-10 was always present in our minds during the design, and as a result BLISS can be implemented very efficiently on the 10. This is probably not true for other machines.

We refer to BLISS as an "implementation language". This phrase has become quite popular lately, but apparently does not have a uniform meaning. Hence it is worthwhile to explain what we mean by the phrase and consequently what our objectives were in the language's design. To us the phrase "implementation language" connotes a higher level language suitable for writing production software; a truly successful implementation language would completely remove the need and/or desire to write in assembly language. Furthermore, to us, an implementation language need not be machine independent--in fact, for reasons of efficiency, it is unlikely to be.

Many reasons have been advanced for the use of a higher level language for implementing software. One of the most often mentioned is that of speeding up its production. This will undoubtedly occur, but it is one of the less important benefits, except insofar as it permits fewer, and better programmers to be used. Far more important, we believe, are the benefits of documentation, clarity, correctness and modifiability. These were the most important goals in the design of BLISS.

Some people, when discussing the subject of implementation languages, have suggested that one of the existing languages, such as PL/I, or at most

a derivative of one, should be used; they argue that there is already a proliferation of languages, so why add another. The only rational excuse for the creation of yet another new language is that existing languages are unsuitable for the specific applications in mind. In the sense that all languages are sufficient to model a Turing machine, any of the existing languages, LISP for example, would be adequate as an implementation language. However, this does not imply that each of these languages would be equally convenient. For example, FORTRAN can be used to write list processing programs, but the lack of recursion coupled with the requirement that the programmer code his own primitive list manipulations and storage control makes FORTRAN vastly inferior to, say, LISP for this type of programming.

What, then, are the characteristics of systems programming which should be reflected in a language especially suited for the purpose? Ignoring machine dependent features (such as a specific interrupt structure) and recognizing that all differences in such programming characteristics are only ones of degree, three features of systems programming stand out:

1. Data structures. In no other type of programming does the variety of data structures nor the diversity of optimal representations occur.
2. Control structures. Parallelism and time are intrinsic parts of the programming system problem.*
3. Frequently, systems programs cannot presume the existence of large support routines (for dynamic storage allocation, for example).

---

* Of course, parallelism and time are intrinsic to real time programming as well.

These are the principal characteristics which the design of BLISS attempts to address. For example, taking point (3), the language was designed in such a way that no system support is presumed or needed, even though, for example, dynamic storage allocation is provided. Thus, code generated by the compiler can be executed directly on a "bare" machine. Another example, taking point (1), is the data structure definition facility. BLISS contains no implicit data structures (and hence no presumed representations for structures), but rather provides a method for defining a representation by giving the explicit accessing algorithm.

One final point before proceeding with the description of the language--namely, the method of syntax specification. The syntax is given in BNF, for example

escapeexpression → EXITBLOCK escapeexpression | EXITLOOP escapeexpression

escapeexpression → | e

where: (1) lower case words are metalinguistic variables, and (2) the 'empty' construct is represented by a blank (as in the first alternative of the second rule above).

TABLE OF CONTENTS

I. LANGUAGE DEFINITION

1.1 Modules

A module is a program element which may be compiled independently of other elements and subsequently loaded with them to form a complete program.

module → MODULE name (parameters) = e ELUDOM

A module may request access to other modules' variables and functions by declaring their names in EXTERNAL declarations. A module permits general use of its own variables and ROUTINEs by means of GLOBAL declarations. These lines of communication between modules are linked by the loader prior to execution. A complete program consists of a set of compiled modules linked by the loader.

The 'name' in a module declaration is used to identify that module and must be unique in its first four characters from any other global names which are to be linked together to form a complete program. The 'parameters' field of a module definition is used to control the compilation (see section II.1.4). See section IV-1.3 for other uses of the module name.

1.2 Blocks and Comments

A block is an arbitrary number of declarations followed by an arbitrary number of expressions all separated by semicolons and enclosed in a matching begin-end or '('-')' pair.

block → BEGIN blockbody END | (blockbody)

compoundexpression → BEGIN expressionsequence END | (expressionsequence)

blockbody → declarations; expressionsequence

declarations → declaration | declaration; declarations

expressionsequence → | e | e; expressionsequence

comment → | ! restofline endoflinesymbol|% stringwithnopercent %

Comments may be enclosed between the symbol ! and the end of the line on which the ! appears. However, a ! may appear in the quoted string of a literal, or between two % symbols, without being considered the beginning of a comment. Likewise, a % enclosed within quotes will be considered part of a string.

As in Algol the block indicates the lexical scope of the names declared at its head. However, in contrast to Algol, there is an exception. The names of GLOBAL variables and ROUTINEs have a scope beyond the block and although they are declared within the module, the effect, for a module citing them in an EXTERNAL declaration, is as if they were declared in the current block. This violation of block structure has implications with respect to allowed references, particularly in connection with declared registers. These implications, and a corresponding set of restrictions, will be discussed in connection with the affected declarations.

1.3 Literals

The basic data element is a PDP-10 36 bit word. However, the hardware provides the capability of pointing to an arbitrary contiguous field within a word and so a 36 bit word may be regarded as a special case of the "partial word". Literals are normally converted to a single word.

literal → number | quotedstring | plit

number → decimal | octal | floating

decimal → digit | decimal digit

octal → # oit | octal oit

floating → decimal.decimal | decimal.decimal exponent | decimal.exponent

exponent → E decimal | E + decimal | E - decimal

digit → 0|1|2 --- |9

oit → 0|1|2 --- |7

numbers (unsigned integers) are converted to binary modulo 2^{36} residue -2^{35}. The binary number is 2's complement and is signed. Octal constants are prefixed by the sharp sign, #. Floating numbers must have an embedded decimal point and no embedded blanks!

quotedstring → leftadjustedstring | rightadjustedstring

leftadjustedstring → 'string'

rightadjustedstring → "string"

Quoted-string literals may be used to specify bit patterns corresponding to the 7-bit ASCII code for visable graphic characters on the external I/O media. Two types of single-word strings are provided for left or right

justification of the string within a word. Normally quoted strings are limited to five characters and the unused bit positions are filled with zeroes.

Within a quoted string the quoting character is represented by two successive occurrences of that character.

1.3.1 Pointers to Literals - "plit"s

A plit is a pointer to a literal word whose contents are specified at compile time; e.g., plit 3 is a pointer to a word whose contents will be set to 3 at load time.

```
plit → plit plitarg
plitarg* → load-time-expression |
           long-string |
           triple
triple → (triple-item-list)
triple-item-list → triple-item | triple-item, triple-item-list
triple-item → load-time-expression |
              long-string |
              duplication-factor: plitarg
duplication-factor → compile-time-expression
```

*Note: "plit (3)+4" has 2 parses: plit load-time-expression
and plit triple + expression

The latter choice is used. Hence, "plit (3)+4" is the same as "(plit 3)+4".

A plit may point to a contiguously stored sequence of literals - long strings and nested lists of literals are also allowed. The value of

```
plit (3,5,7,9)
```

is a pointer to 4 contiguous words containing 3,5,7 and 9 respectively. A long string (> 5 characters) is also a valid argument to a plit:

```
plit  'THIS ALLOCATES 5 WORDS'
```

allocates 5 words of 7-bit ASCII characters with 3 pad characters of zero to the right and the last bit turned on.

The arguments to plits need only be constant at load time; plits are themselves literals, thus nesting of plits is allowed (with the inner plits allocated first):

```
external A,B,C;
bind y = plit (A, plit (B,C), plit 3, 'A LONG STRING', 5+9*3);
```

is such that:

```
.y[0] = A<0,36>; ..y[1] = B<0,36>; .(.y[1]+1) = C<0,36>
..y[2] = 3;  .y[3] = 'A LON'; .y[4] = 'G STR'; .y[5] = 'ING' or 1;
.y[6] = 32;
```

In addition, any argument to a plit can be replicated by specifying the number of times it is to be repeated; e.g.

```
plit (7:3)
```

produces a pointer to 7 contiguous words, each of which contains the value 3. Duplicated plits are allocated once, identical plits are not pooled - hence,

```
bind  x = plit (3: plit A, plit A, 2: (2,3));
```

is such that:

```
..x[0] = ..x[1] = ..x[2] = ..x[3] = A<0,36>;
 .x[0] =  .x[1] =  .x[2] ≠  .x[3];
 .x[4] =  .x[6] = 2; .x[5] = .x[7] = 3;
```

Note: the length of every plit (in words) is stored as the word preceding the plit. Hence, in the last example, .x[-1] = 8.

1.4 Names

Syntactically an identifier, or name, is composed of a sequence of letters and/or digits, the first of which must be a letter. Certain names are reserved as delimiters, see Appendix A. Semantically the occurrence of a name is exactly equivalent to the occurrence of a pointer to the named item. The term "pointer" will take on special connotation later with respect to contiguous sub-fields (bytes) within a word; however, for the present discussion the term may be equated with "address". This interpretation of name is uniform throughout the language and there is no distinction between left and right hand values. Contrast this with Algol where a name usually, but not always, means "contents of".

The pointer interpretation requires a "contents of" operator, and "." has been chosen. Thus .A means "contents of location A" and ..A means "contents of the location whose name is stored in location A". To illustrate the concept, consider the assignment expression

p11 ← e

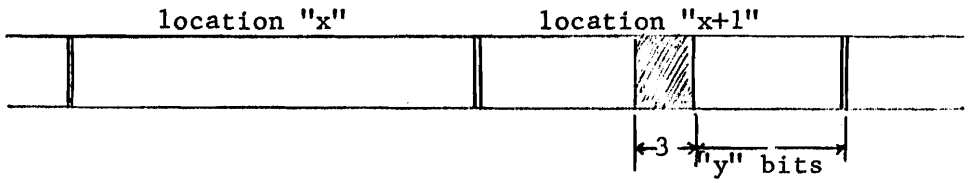
This means "store the value computed from e into the location whose pointer is the value of p11". (Further details are given in 2.2.) Thus the Algol statement "A := B" is written "A ← .B". It is impossible to express in Algol BLISS expressions such as: "A ← B", "A ← ..B", ".A ← .B", etc.

1.5 Pointers

As explained in 1.4, the value of a name is a pointer which names a location in memory. However, pointers are more general than mere addresses since they may name an arbitrary contiguous portion of a word, and may, further, involve index modification and indirect addressing. (For full details, the reader should refer to the PDP-10 System Reference Manual.) The most general form of pointer specifies five quantities; an example is $\epsilon_0 <\epsilon_1,\epsilon_2,\epsilon_3,\epsilon_4>$, where ϵ_0 is computed modulo 2^{18} and forms the base word address (Y field); ϵ_1,ϵ_2, are computed modulo 2^6 and form the position, size fields respectively (P, S fields); ϵ_3 is computed modulo 2^4 and forms the index field (X field); ϵ_4 is computed modulo 2 and forms the indirect address bit (I field). Each of $\epsilon_1,\epsilon_2,\epsilon_3,\epsilon_4$ may optionally be omitted, in which case a default value is supplied. $\epsilon_1,\epsilon_3,\epsilon_4$ have defaults of 0, but ϵ_2 has the default of 36. Thus, for example, the expression

```
(x+1)<.y,3>
```

defines a three bit field in the first location beyond x. The position of this three bit field is ".y" bits from the right end of the word.



1.6 The "contents of" Operators

The interpretation placed on identifiers in Bliss coupled with the dot operator discussed earlier allow a programmer direct access to, and control over, fields within words, to pointers to such fields which are themselves stored within memory, to chains of such pointers, etc. Two additional "contents of" operations besides the dot are provided which are more efficient in certain cases, but which are defined in terms of the dot and pointer operations. These operators are @ and \, and are defined by the following (where t is a temporary):

$$@\in \equiv .\in < 0, 36, 0, 0 >$$
$$\backslash\in \equiv .(t \leftarrow \in) < 0, 36, .t < 18, 4 >, .t < 22, 1 >>$$

Thus, both @∈ and \∈ specify a full 36 bit value. @∈ uses only the rightmost 18 bits of ∈ as the absolute address from which to fetch the value. \∈ interprets the rightmost 23 bits of ∈ as an indirect bit, index register field and base address. Whichever form is used, the compiler attempts to optimize the code produced; thus, for example, identical code is produced for .x, @x, and \x, if they occur in an expression.

Suppose that the assignment "X ← Y < 3, 15, R1, 0>;" has been executed, that is a pointer has been stored in X (that pointer has P=3, S=15, X=R1, I=0), and further that register R1 contains two. Now:

(1) Z ← .X stores the value of X, i.e., the pointer, into Z

(2) Z ← ..X stores the value of the fifteen bit field (which ends three bits from the right) on the second word following Y into Z

(3) Z ← @ .X stores the value of Y into Z

(4) Z ← \ .X stores the value of the second word following Y into Z

(5) .X ← 5 stores 5 into the relevant fifteen bit field of the second word following Y

2.1 Expressions

Every executable form in the BLISS language (that is, every form except the declarations) computes a value. Thus all commands are expressions and there are no "statements" in the sense of Algol or Fortran. In the syntax description e is used as an abbreviation for expression.

e → simpleexpression | controlexpression

2.2 Simple Expressions

The semantics of simpleexpressions is most easily described in terms of the relative precedence of a set of operators, but readers should also refer to the BNF-like description in 4.1. The precedence number used below should be viewed as an ordinal, so that 1 means first and 2 second in precedence. In the following table the letter ϵ has been used to denote an actual expression of the appropriate syntactic type, see 4.1.

Precedence	Example	Semantics
1	compoundexpression	The component expressions are evaluated from left to right and the final value is that of the last component expression.
1	block	
1	$\epsilon_0(\epsilon_1,\epsilon_2,\ldots,\epsilon_n)$	A function call, see 3.4.
1	name$[\epsilon_1,\epsilon_2,\ldots,\epsilon_n]$	A structure access, see 3.5.
1	name	A pointer to the named item, see 1.4.
1	literal	Value of the converted literal, see 1.3.
2	ϵ<pointer parameters>	A partial word pointer, see 1.5.
3	$.\epsilon$	Value (possibly partial word) pointed at by ϵ.
3	$@\epsilon$	Equivalent to $.\epsilon$<0.36.0.0>.
3	$\backslash\epsilon$	Equivalent to .(t←ϵ)< 0,36,.t< 18,4>, .t<22,1>>.
4	$\epsilon_1\uparrow\epsilon_2$	ϵ_1 shifted logically by ϵ_2 bits; left if ϵ_2 positive; right if ϵ_2 negative. (Shifts are modulo 256.)
5	$\epsilon*\epsilon$	Product of ϵ's.
5	ϵ_1/ϵ_2	ϵ_1 divided by ϵ_2.
5	ϵ_1 MOD ϵ_2	ϵ_1 modulo ϵ_2.
6	$-\epsilon$	Negative of ϵ.
6	$\epsilon+\epsilon$	Sum of ϵ's.
6	$\epsilon_1-\epsilon_2$	Difference between ϵ_1 and ϵ_2.

[Note all integer arithmetic is carried out modulo 2^{36} with a residue of -2^{35}.]

Precedence	Example	Semantics
5	ϵ_1 FMPR ϵ_2	Floating product of ϵ_1 and ϵ_2.
5	ϵ_1 FDVR ϵ_2	Floating divide of ϵ_1 by ϵ_2.
6	FNEG ϵ_1	Floating negate of ϵ_1.
6	ϵ_1 FADR ϵ_2	Floating sum of ϵ_1 and ϵ_2.
6	ϵ_1 FSBR ϵ_2	Floating difference of ϵ_1 and ϵ_2.
7	ϵ_1 EQL ϵ_2	$\epsilon_1 = \epsilon_2$
7	ϵ_1 NEQ ϵ_2	$\epsilon_1 \neq \epsilon_2$
7	ϵ_1 LSS ϵ_2	$\epsilon_1 < \epsilon_2$
7	ϵ_1 LEQ ϵ_2	$\epsilon_1 \leq \epsilon_2$
7	ϵ_1 GTR ϵ_2	$\epsilon_1 > \epsilon_2$
7	ϵ_1 GEQ ϵ_2	$\epsilon_1 \geq \epsilon_2$

[Truth is represented by 1, falsity by 0.]

8	NOT ϵ	bitwise complement of ϵ
9	ϵ AND ϵ	bitwise and of ϵ's
10	ϵ OR ϵ	bitwise inclusive or of ϵ's
11	ϵ XOR ϵ	bitwise exclusive or of ϵ's
11	ϵ EQV ϵ	bitwise equivalence of ϵ's
12	$\epsilon \leftarrow \epsilon_2$	The value of this expression is identical to that of ϵ_2, but as a side effect this value is stored into the partial word pointed to by ϵ_1; with associative use of $\leftarrow$, the assignments are executed from right to left: thus $\epsilon_1 \leftarrow \epsilon_2 \leftarrow \epsilon_3$ means $\epsilon_1 \leftarrow (\epsilon_2 \leftarrow \epsilon_3)$.

There is no guarantee regarding the order in which a simpleexpression is evaluated other than that provided by precedence and nesting: thus (R ← 2; @ R * (R ← 3)) may evaluate to 6 or 9.

The reader should refer to the PDP-10 reference manual for a complete definition of the arithmetic operators under various special input value conditions.

2.3.1 Control Expressions

The controlexpressions provide sequencing control over the execution of his program; there are five forms:

controlexpression → conditionalexpression | loopexpression | choiceexpression | escapeexpression | coroutineexpression

The general goto statement has deliberately been omitted from the language to improve readability and structuring of programs.

2.3.2 Conditional Expressions

conditionalexpression → IF e_1 THEN e_2 ELSE e_3

e_1 is computed and the resulting value is tested. If it is odd*, then e_2 is evaluated to provide the value of the conditional expression, otherwise e_3 is evaluated.

conditionalexpression → IF e_1 THEN e_2

This form is equivalent to the IF-THEN-ELSE form with 0 replacing e_3. However, it does introduce the "dangling else" ambiguity. This is resolved by matching each ELSE to the most recent unmatched THEN as the conditional expression is scanned from left to right.

---

* Only the least significant bit of e_1 is tested; a zero bit is interpreted as false and a one bit as true. Thus any odd integer value is interpreted as true and any even value as false.

2.3.3 Loop Expressions

The value of each of the six loop expressions is -1, except when an EXITLOOP is used, see 2.3.4.

loopexpression → WHILE e_1 DO e_2

The e_1 is computed and the resulting value is tested. If it is odd, then e_2 is computed and the complete loopexpression is recomputed; if it is even, then the loopexpression evaluation is complete.

loopexpression → UNTIL e_3 DO e_2

This form is equivalent to the WHILE-DO form except that e_1 is replaced by NOT(e_3).

loopexpression → DO e_2 WHILE e_1

The expressions e_2,e_1 are computed in that sequence. The value resulting from e_1 is tested: if it is odd, then the complete loop expression is recomputed; if it is even, then the loopexpression evaluation is complete.

loopexpression → DO e_2 UNTIL e_3

This form is equivalent to the DO-WHILE form except that e_1 is replaced by NOT(e_3).

loopexpression → INCR name FROM e_1 TO e_2 BY e_3 DO e_4

This is a simplified form of the Algol 68 for-loop. The "name" is declared to be a REGISTER or a LOCAL for the scope of the loop. The expression e_1 is computed and stored in name. The expressions e_2 and e_3 are computed and stored in unnamed local memory which for explanation purposes we shall name U_2 and U_3. Any of the phrases "FROM e_1" "TO e_2" or "BY e_3" may be omitted--

in which case default values of $e_1 = 0$, $e_2 = 2^{35}-1$, $e_3 = 1$ are supplied. The following loopexpression is then executed:

```
BEGIN  REGISTER name; LOCAL U2,U3; U2 ← e2; U3 ← e3;
       UNTIL .name GTR .U2 DO (e4; name ← .name +.U3)
END
```

The final form of a loopexpression is:

loopexpression → DECR name FROM e_1 TO e_2 BY e_3 DO e_4

This is equivalent to the INCR-FROM-TO-BY-DO form except that the final loop is replaced by

```
BEGIN  REGISTER name; LOCAL U2,U3; U2 ← e2; U3 ← e3;
       UNTIL .name LSS .U2 DO (e4; name ← .name - .U3)
END
```

If any of the FROM, TO, or BY phrases are omitted from a DECR expression, default values of $e_1 = 0$, $e_2 = -2^{35}$, and $e_3 = 1$ are supplied. Notice that in both forms the end condition is tested before the loop, hence the loop is potentially executed zero or more times.

2.3.4 Escape Expressions

The various forms of escapeexpressions permit control to leave its current environment. They are intended for those circumstances when other controlexpressions would have to be contorted to achieve the desired effect.

escapeexpression → environment level escapevalue | RETURN escapevalue

environment → EXIT | EXITBLOCK | EXITCOMPOUND | EXITLOOP | EXITCOND
EXITCASE | EXITSET | EXITSELECT

level → | [e]

escapevalue → | e

Each of these expressions conveys to its new environment a value, say $\in$, obtained by evaluating the escapevalue, which may optionally be omitted implying $\in = 0$. The levels field, which must evaluate to a constant, say n, at compile time, determines the number of levels of the specified control environment to be exited; the levels field may optionally be omitted in which case one level is implied. The maximum number of levels which may be exited in this way is limited by the current function (routine) body or the outermost block.

RETURN	terminates the current function, or routine, with value $\in$.
EXITBLOCK	terminates the innermost n (where n is the value of the "levels" field) blocks, yielding a value of $\in$ for the outermost one exited.
EXITCOMPOUND	terminates the innermost n compound expressions, yielding a value of $\in$ for the outermost one exited.
EXITLOOP	terminates the innermost n loop expressions, yielding a value of $\in$ for the outermost one exited.
EXITCOND	terminates the innermost n conditional expressions, yielding a value of $\in$ for the outermost one exited.
EXIT	terminates the innermost n control scopes (whether blocks, compounds, conditionals, or loops with $\in$ as the value of the outermost.

EXITCASE terminates the n innermost case expressions yielding a value of $\in$ for the outermost of these.

EXITSET terminates the n innermost set expressions, yielding a value of $\in$ for the outermost of these.

EXITSELECT terminates the n innermost select expressions, yielding a value of $\in$ for the outermost of these.

2.3.5 Choice Expressions

choiceexpression → CASE elist OF SET expressionset TES

elist → e | e, elist

expressionset → |e|; expressionset | e ; expressionset

Let us suppose that the actual e's within the elist are $\epsilon_1, \epsilon_2, \ldots, \epsilon_m$ and that the actual expressions within the expressionset are $\Pi_0; \Pi_1; ,,,; \Pi$. Then the expressions $\Pi_{\epsilon_1}, \Pi_{\epsilon_2}, \ldots \Pi_{\epsilon_m}$ are executed in that order. The value of the case expression is that of Π_{ϵ_m}.

choiceexpression → SELECT elist OF NSET nexpressionset TESN

nexpressionset → | ne | ne; nexpressionset

ne → e:e

This form is somewhat similar to the case expression except that the expressions in the nexpressionset are not thought of as being sequentially numbered--instead each expression in the nexexpressionset is tagged with an "activation" expression. Suppose we have the following select expression

SELECT ϵ_1, ϵ_2, ϵ_3 OF NSET ϵ_4: ϵ_5; ϵ_6: ϵ_7; ϵ_8: ϵ_9; ϵ_{10}: ϵ_{11} TESN

then the execution proceeds as follows: first ϵ_1, ϵ_2, ϵ_3 are evaluated, then ϵ_4, ϵ_6, ϵ_8 and ϵ_{10} are evaluated; correspondingly ϵ_5 is evaluated if and only if ϵ_4 is equal to one of ϵ_1, ϵ_2, or ϵ_3. Similarly ϵ_7 is evaluated if and only if ϵ_6 is equal to one of ϵ_1, ϵ_2, or ϵ_3, etc. The order of comparison of ϵ_4, ϵ_6, etc. is from left-to-right, and the value of the select expression is the last of ϵ_5, ϵ_7, etc. to be evaluated (or -1 if none is evaluated).

In place of one of the selection expressions, ϵ_4, ϵ_6, etc. one of the two reserved words OTHERWISE or ALWAYS may be used, e.g., "ALWAYS:ϵ_9". The expression following an "OTHERWISE:" will be executed just in the case that none of the preceding selection criteria were satisfied. The expression following an "ALWAYS:" will always be executed independent of the selection criteria. In the following example

```
z ← SELECT  .x,.y OF
        NSET
        1:ϵ[1]
        7:ϵ[2]
OTHERWISE: ϵ[3]
       36: ϵ[4]
   ALWAYS: ϵ[5]
       94: ϵ[6]
        TESN;
```

(1) ϵ^1 will be executed if .x=1 or .y=1, then (2) ϵ^2 will be executed if .x=7 or .y=7, then (3) ϵ^3 will be executed in the case neither ϵ^1 nor ϵ^2 was executed, i.e., .x≠1, .y≠1, .x≠7, and .y≠7, then (4) ϵ^4 will be executed if .x=36 or .y=36, then (5) ϵ^5 will always be executed, and finally (6) ϵ^6 will be executed if .x=94 or .y=94. The value assigned to z will be that of ϵ^5 unless .x=94 or .y=94 in which case the value assigned to z will be that of ϵ^6.

Note that although OTHERWISE and ALWAYS may be placed in any nset-element, it makes no sense to use more than one OTHERWISE or to use an OTHERWISE after an ALWAYS since in these cases the latter OTHERWISE's can have no effect.

2.3.6 Co-routine Expressions

The body of a function or routine may be activated as a co-routine and/or asynchronous process; the additional syntax is

coroutineexpression → CREATE e_1 (elist) AT e_2 LENGTH e_3 THEN e_4 |
EXCHJ (e_6, e_7)

The effect of a 'create' expression is to create a context, that is an independent stack, for the routine (function) named by e_1, with parameters specified by the elist, at the location whose address is specified by e_2 and of size e_3 words. Control then passes to the statements following the 'create'. When two or more such contexts have been established, control may be passed from any one to any other by executing an exchange-jump, EXCHJ (e_6, e_7)*, where the value of e_6 must be the stack base, e_2, of a previous 'create' expression. The value of e_7 is made available to the called routine as the value of its own EXCHJ which caused control to pass out of that routine. Thus the value of the EXCHJ operation is defined dynamically by the co-routine which at some later time re-activates execution of the current co-routine.**

Should a process, the body of which is necessarily that of a function (or routine), execute a 'return', either explicitly or implicitly, the expression e_4 (following the 'then' in the 'create' expression of the creating process) is executed in the context of the created process. The normal responsibilities of e_4 include making the stack space used for the created context available for other uses and performing an EXCHJ to some other process.

The facilities described above, namely 'create' and 'exchj', are adequate either for use directly as co-routine linkages or for use as primitives in constructing more sophisticated co-routine facilities with macros

* Note that the 1st EXCHJ to a newly created process causes control to enter from its head with actual parameters as set up by the CREATE.

** The value e_7 is not available to the called routine on the 1st EXCHJ to it.

and/or procedures. It should be noted in the context that if the created processes are functions (rather than routines) the resulting processes continue to have access to lexically global variables which may be local to an embracing function (access to lexically local variables which have been declared 'own' is available in either case). In such a case the resulting structure is a stack tree in which all segments of the tree below the lexical level of the (function) process are available to it.

Two additional complexities are added if the create and exchj are to be used for asynchronous, and possibly parallel, execution of processes. One is synchronization, by which we man a mechanism by which a process can coordinate its execution with that of one or more others. A typical example of the need for synchronization occurs when two processes, independently update a common data base, and each must be sure that the entire updating process is complete before any other process attempts to use the data base. The second complexity arises in connection with interrupts, and in particular from the fact that certain operations must not be interrupted (some exchj operations for example). It is possible that certain situations require synchronization mechanisms but do not need to be concerned about the interrupt problem--as for example, a user program with asynchronous processes, which is 'blind' to interrupts, and which some monitor systems view as a single 'job'.

The nature of "appropriate" synchronization primitives and mechanisms for temporarily blinding the processor to interrupts (or interrupts in a certain class) are highly dependent upon the nature of the processes being used and the operating system, or lack of one, underlying the Bliss program. As a consequence, no syntax for dealing with either problem is included in

the language; in any case, the amount of code necessary for these facilities is quite small.

The co-routine user is well advised to read and understand the material on the run-time representation of Bliss programs contained in section IV.

3.1 Declarations

All declarations, except MAP and SWITCH, introduce names each of which is unique to the block in which the declaration appears. Except with STRUCTURE and MACRO declarations, the name introduced has a pointer bound to it.

The declarations are:

```
declaration → functiondeclaration|structuredeclaration|
              binddeclaration|macrodeclaration|
              allocationdeclaration|mapdeclaration
```

Before proceeding with a detailed discussion of the declarations we shall give an intuitive overview of the effect of these declarations.

3.1.1 Storage (an introduction)

A Bliss program operates with and on a number of storage "segments". A storage segment consists of a fixed and finite number of "words", each of which is composed of a fixed and finite number of "bits" (36 for the PDP-10). Any contiguous set of bits within a word is called a "field". Any field may be "named", the value of a name is called a "pointer" to that field. In particular, an entire word is a field and may be named.

In practice a segment generally contains either program or data, and if the latter, it is generally integer numbers, floating point numbers, characters, or pointers to other data. To a Bliss program, however, a field merely contains a pattern of bits.

Segments are introduced into a Bliss program by declarations, called allocation declarations, for example:

```
global g;
own x,y [5], z;
local p [100];
register r1, r2 [3];
function f(a,b) = .a↑.b;
```

Each of these declarations introduces one or more segments and binds the identifiers mentioned (e.g., g, x, y, etc.) to the name of the first word of the associated segment. (The function declaration also initializes the segment named "f" to the appropriate machine code.)

The segments introduced by these declarations contain one or more words, where the size may be specified (as in "local p[100]"), or defaulted to one (as in "global g;"). The identifiers introduced by a declaration

are lexically local to the block in which the declaration is made (that is, they obey the usual Algol scope rules) with one exception - namely, "global" identifiers are made available to other, separately compiled modules. Segments created by own, global, and function declarations are created only once and are preserved for the duration of the execution of a program. Segments created by local and register declarations are created at the time of block entry and are preserved only for the duration of the execution of that block. Register segments differ from local segments only in that they are allocated from the machine's array of 16 general purpose (fast) registers. Re-entry of a block before it is exited (by recursive function calls, for example) behaves as in Algol, that is, local and register segments are dynamically local to each incarnation of the block.

There are two additional declarations whose effect is to bind identifiers to names, but which do not create segments; examples are:

```
external  s;
bind      y2 = y+2, pa = p+.a;
```

An external declaration binds one or more identifiers to the names represented by the same identifier declared global in another, separately compiled module. The bind declaration binds one or more identifiers to the value of an expression at block entry time. At least potentially the value of this expression may not be calculable until run time - as in 'pa = p+.a' above.

3.1.2 Data Structures (an introduction)

Two principles were followed in the design of the data structure facility of Bliss:

- the user must be able to specify the accessing algorithm for elements of a structure,
- the representational specification and the specification of algorithms which operate on the information represented must be separated in such a way that either can be modified without affecting the other.

The definition of a class of structures, that is, of an accessing algorithms to be associated with certain specific data structures, may be made by a declaration of somewhat the following form:

structure <name>[<formal parameter list>] = ∈

Particular names may then be associated with a structure class, that is with an accessing algorithm, by another declaration of somewhat the form:

map <name> <name list>

Consider the following example:

```
begin
   structure ary2[i,j] = (.ary2+(.i-1)*10+(.j-1));
   own x[100],y[100],z[100];
   map ary2 x:y:z;
   .
   .
   .
   x[.a,.b] ← .y[.b,.a];
   .
   .
   .
end;
```

In this example we introduce a very simple structure, ary2, for two dimensional (10x10) arrays, declare three segments with names 'x', 'y', and 'z' bound to them, and associate the structure class 'ary2' with these names. The syntactic forms "x[ϵ_1,ϵ_2]" and "y[ϵ_3,ϵ_4]" are valid within this block and denote evaluation of the accessing algorithm defined by the ary2-structure declaration (with an appropriate substitution of actual for formal parameters).

Although they are not implemented in this way, for purposes of exposition one may think of the structure declaration as defining a function with one more formal parameter than is explicitly mentioned. For example, the structure declaration in the previous example,

```
structure ary2[i,j] = (.ary2+(.i-1)*10+(.j-1));
```

conceptually is identical to a function declaration

```
function ary2(f0,f1,f2) = (.f0+(.f1-1)*10+(.f2-1));
```

The expressions "x[.a,.b]" and "y[.b,.a]" correspond to calls on this function - i.e., to "ary2(x,.a,.b)" and "ary2(y,.b,.a)".

Since, in a structure declaration, there is an implicit, un-named formal parameter, the name of the structure class itself is used to denote this "zero-th" parameter. This convention maintains the positional correspondence of actuals and formals. Thus, in the example above, ".ary2" denotes the value of the name of the particular segment being referenced, and 'x[.a,.b]' is equivalent to:

```
(x+(.a-1)*10+(.b-1))
```

The value of this expression is a pointer to the designated element of the segment named by x.

In the following example the structure facility and bind declaration have been used to encode a matrix product ($z_{i,j} = \sum_{k=1}^{10} x_{ik} y_{kj}$). In the inner block the names 'xr' and 'yc' are bound to pointers to the base of a specified row of x and column of y respectively. These identifiers are then associated with structure classes which allow one-dimensional access.

```
begin
    structure  ary2[i,j] = (.ary2+(.i-1)*10+(.j-1)),
               row[i] = (.row+.i-1),
               col[j] = (.col+(.j-1)*10);
    own        x[100],y[100],z[100];
    map        ary2 x:y:z;
    .
    .
    .
    incr i from 1 to 10 do
       begin bind xr = x[.i,1], zr = z[.i,1]; map row xr:zr;
       incr j from 1 to 10 do

          begin
             register t; bind yc=y[1,.j]; map col yc;
             t ← 0;
             incr k from 1 to 10 do t ← .t+.xr[.k]*.yc[.k];
             zr[.j] ← .t;
          end;
       end;
       .
       .
       .
end
```

3.1.3 The Actual Declaration Syntax

The example declarations in the preceding two sub-sections are valid Bliss syntax; however, they do not reflect the complete power of the declarative facilities. The following sections (3.2 - 3.5) are definitive presentations of the actual syntax and semantics of these declarations. The actual declarations presented in the following sections differ from the examples given previously in that they admit greater interaction between the allocation declarations and structure declarations.

3.2 Memory Allocation

There are five basic forms of allocation declaration:

```
allocation declaration → allocatetype msidlist
allocatetype → GLOBAL|REGISTER|OWN|LOCAL|EXTERNAL
msidlist → msidelement|msidelement, msidlist
msidelement → structure sizedchunks
structure → | structurename
sizedchunks → sizedchunk|sizedchunk: sizedchunks
sizechunk → idchunk|idchunk [elist]
idchunk → name|name:idchunk
```

As with most other declarations, the allocation declarations introduce names whose scope is the block in which the declarations occur. REGISTER and LOCAL declarations cause allocation of storage at each block entry (including recursive and quasi-parallel ones), and corresponding de-allocation on block exit. Storage for OWN and GLOBAL declarations is made once (before execution begins) and remains allocated during the entire execution of the program. EXTERNAL declarations do not allocate storage, but cause a linkage to be established to storage declared with the same name in a GLOBAL declaration of another module. Space for allocation is taken from core for LOCAL, OWN, and GLOBAL declarations, and from the machine's high speed registers for REGISTER declarations.

The initial contents of allocated memory is not defined and should not be presumed.

Each msidelement defines a set of identifiers and simultaneously maps these identifiers onto a specified structure. (If the structure part is empty, the default structure 'vector' is assumed, see section 3.5). Each sizedchunk allows, by interaction with the associated

structure of the msidelement, specification of the size of the segment to be allocated - and the values of the "undotted structure formals" to be used in accessing an instance of the structure (again, see 3.5).

3.3 Map Declaration

map declaration → MAP msidlist

The map declaration is syntactically and semantically similar to an allocation declaration except that no new storage or identifiers are introduced. The purpose of the map declaration is to permit re-definition of the structure and elist information associated with an identifier (or set of identifiers) for the scope of the block in which the map declaration occurs.

3.4 Bind Declarations

bind declaration → BIND equivalencelist
equivalencelist → equivalence | equivalence, equivalencelist
equivalence → msidelement = e

A bind declaration introduces a new set of names whose scope is the block in which the bind declaration occurs, and binds the value of these names to the value of the associated expressions at the time that the block is entered. Note that these expressions need not evaluate at compile time.

3.5 Structures

structure declaration → STRUCTURE name structureformallist = structuresize e_1

structureformallist → | [namelist]

structuresize → | [e_2]

Structure declarations serve to define a class of data structures by defining an explicit "access algorithm", e_1, to be used in accessing elements of that structure. The class of structures introduced by such a declaration is given a name which may be used as the structure name in an allocation declaration or map declaration.

The names in the structure formal list are formal parameter identifiers which are used in two distinct ways:

1. "dotted" occurrences of the formal names positionally correlate with the values of elist elements at the site of a structure access. (Recall that a structure access is syntactically p1 → name [elist].) These are referred to as "access formals" and "access actuals" respectively.

2. "undotted" occurrences of the formal names positionally correlate with the values of the elist elements at the site of the declaration which associated the variable name with the structure class. These are referred to as "incarnation formals" and "incarnation actuals" respectively.

In addition to the explicit formal names, the structure name, in "dotted" form, is used as an access formal to denote the name of the specific segment being accessed (that is, to denote the pointer to the base of the segment).

If present, the structure size, i.e., [e], is used to calculate (from the incarnation actuals) the size of the segment to be allocated by an allocation declaration. After substitution of incarnation actuals, this expression must evaluate to a constant at compile time.

The simple example of a two-dimensional array given in section 3.1.2 might now be written:

```
begin
   structure ary2[i,j] = [i*j](.ary2+(.i-1)*j+(.j-1));
   own ary2 x:y:z[10,10];
   .
   .
   .
   x[.a,.b] ← .y[.b,.a];
   .
   .
   .
end;
```

The default structure VECTOR, mentioned in section 3.2 is defined by

```
structure vector [i] = [i] (.vector + .i);
```

If defaulted, the size part of a structure declaration is defaulted to the product of the incarnation actuals.

3.6 Functions

function declaration → FUNCTION name (namelist) = e |
FUNCTION name = e |
ROUTINE name(namelist) = e |
ROUTINE name = e

The FUNCTION and ROUTINE declarations define the name to be that of a potentially recursive and re-entrant function whose value is the expression e. The syntax of a normal subroutine-like function call is

p1 → p1 (elist) | p1 ( )
elist → e | elist, e

where p1 is a primary expression. Clearly, p1 must evaluate to a name which has been declared as a FUNCTION or ROUTINE either at compile time or at run time. The names in the namelist of the declaration define (lexically local) the names of formal parameters whose actual values on each incarnation are determined by the elist at the call site. All parameters are implicitly Algol "call-by-value"; but notice that call-by-reference is achieved by simply presenting pointer values at the call site. Parentheses are required at the call site even for a ROUTINE or a FUNCTION with no formal parameters since the name on its own is simply a pointer to the function or routine. Extra actual parameters above the number mentioned in the namelist of the function (or routine) declaration are always allowed; however, too few actual parameters can cause erroneous results at run time.* A ROUTINE differs from a FUNCTION in having an abbreviated and hence faster prolog. Restriction: a routine may not refer directly to local variables declared outside it, nor may it call a FUNCTION.

*Note: If extra parameters are presented, and say, n are expected, then the rightmost n actual will correspond to the formal parameters. See section IV for details of the access mechanism.

function declaration → GLOBAL ROUTINE name (namelist) = e |

GLOBAL ROUTINE name = e

A ROUTINE name is like an OWN name in that its scope is limited to the block in which it is declared and its value is already initialized at block entry. The prefix GLOBAL changes the scope of the ROUTINE to that of the outer block of the program enveloping all the modules. Note that this inhibits a GLOBAL ROUTINE from access to REGISTER names declared outside it. This is in addition to the other limitations of ROUTINES cited on the previous page.

Functions and routines may also be activated as co-routines and/or asynchronous processes, and indeed, the body of a single function may be used in any or all of these modes simultaneously. (See 2.3.6.)

function declaration → FORWARD nameparlist

nameparlist → namepar | nameparlist, namepar

namepar → name (e)

FORWARD's tell the compiler how many parameters, given by e*, are expected by an undeclared function (or routine) name which will be declared later in the current block. The compiler permits the number of actual parameters in a function (or routine) call to be greater than or equal to the number of formals declared.

*Clearly e must evaluate to a constant at compile time.

3.7 Simple Macros

A limited macro facility is provided to improve the usability of the language. This facility provides simple replacement of a macro keyword (and arguments) by a suitably defined string (with appropriate actual string substitution for the formal parameters). Nested macro calls are permitted. Recursive macro calls and nested macro definitions are not permitted.

```
macrodeclaration → MACRO macdefinitionlist
macdefinitionlist → macdefinition |
                    macdefinitionlist, macdefinition
macdefinition → name1 (namelist) = stringwithout$ $ |
                name2 = stringwithout$ $
```

The stringwithout$ is scanned for occurrences of atoms that match elements of the namelist (if any). The first $ terminates the macdefinition without exception.

```
macrocall → name1 (balancedstringlist) |
            name2
balancedstringlist → balancedstring |
                     balancedstringlist, balancedstring
```

A balancedstring is any string for which the number of right brackets ("(", "[", or "<") in the string equals or exceeds the number of corresponding left brackets. This includes the null string. A balancedstring is associated with the formal parameter in the corresponding ordinal position in the macdefinition.

Note that

1. "Extra" balancedstrings will be simply ignored, but parsed as described above.
2. Null balancedstrings are accepted.
3. The macrocall may present fewer balancedstrings than the macrodefinition, in which case the null string will be used for the "missing" arguments.
4. A macrocall must have a balancedstringlist if the macrodefinition had a namelist.

The expanded string from a macro replaces the macrocall in the program prior to lexical processing and scanning resumes at the head of this string. Hence macrocalls may be nested. Indeed, parts of a "nested" call may come from the actual parameter(s) of the containing macro, from the body of the containing macro or even from the text following the containing macro.

As with other declarations, macros have a scope given by the block in which they are defined - with this exception: Any macro being expanded at the end of a block will, in effect, be purged but its expansion will run to completion. This might occur, for example, if a macro contained an END as in:

```
BEGIN
   MACRO  QQSV = END  B ← "TQ" $;
   QQSV
END
```

This may lead to anomolous behavior depending on the specific program.

Macros may be used to provide names to bit fields so as to improve readability.

```
MACRO EXPONENT = 27,8 $;
MACRO MANTISSA = 0,27 $;
MACRO SIGN = 35,1 $;
LOCAL X;
X <SIGN> ← 0; X <EXPONENT> ← 27; X <MANTISSA> ← .I;
```

Macros may be used to extend the syntax in a limited way.

```
MACRO NEG = 0 GTR $;
MACRO UNLESS(X) = IF NOT(X) $;
```

Macros may be used to effect in-line coding of a function.

```
MACRO ABS(X) = BEGIN REGISTER TEMP;
               IF NEG(TEMP ← X) THEN -.TEMP ELSE .TEMP END $;
! HERE THE ACTUAL PARAMETER SUBSTITUTED FOR X MAY NOT INCLUDE THE
!                                                    NAME TEMP.
```

II. SPECIAL LANGUAGE FEATURES

The previous chapter describes the basic features of the BLISS language. In this chapter we describe additional features which are highly machine and implementation dependent.

1.1 Special Functions

A number of features have been added to the basic BLISS language which allow greater access to the PDP-10 hardware features. These features have the syntactic form of function calls and are thus referred to as "special functions". Code for special functions is always generated in line.

1.2 Character Manipulation Functions

Nine functions have been specified to facilitate character manipulation operations. They are:

scann (ap)	copynn (ap_1, ap_2)
scani (ap)	copyni (ap_1, ap_2)
replacen (ap, $\in$)	copyin (ap_1, ap_2)
replacei (ap, $\in$)	copyii (ap_1, ap_2)

incp (ap)

For each of these $\in$ is an arbitrary expression, and ap is an expression whose value is a pointer to a pointer. The second of these pointers is assumed to point to a character in a string.

scann (ap) is a function whose value is the character from the string.

scani (ap) is like scann except that, as a side effect, the string pointer is set to point at the next character of the string before the character is scanned.

replacen (ap, $\in$) is a function whose value is $\in$ and which, as a side effect, replaces the string character by $\in$.

replacei (ap, $\in$) is similar to replacen except that the string pointer is set to point at the next character of the string before the value of $\in$ is stored.

copynn (ap_1, ap_2)
copyni (ap_1, ap_2)
copyin (ap_1, ap_2)
copyii (ap_1, ap_2) — these functions are similar in that they each effect a copy of one character from a source string (pointed at by $.ap_1$) to a destination string (pointed at by $.ap_2$) and have as value the character copied. They differ in that copynn advances neither pointer, while copyni advances $.ap_2$, copyin advances $.ap_1$, and copyii advances both. In each case the pointer is advanced before the copy is effected.

incp (ap) advances .ap to the next character

Suppose that a string (of 7 bit ASCII characters) is stored in memory beginning at location S. The string is terminated by a null (zero) character. The following skeletal code will transform it into a 6-bit string with blanks deleted:

```
begin
register p7, p6, c;
p7 ← (s-1) <1, 7>; p6 ← (s-1) <0,6>;
while (c ← scani (p7)) neq 0 do
      if .c neq " " then replacei (p6, .c);
...
end;
```

1.3 Machine Language

It is possible to insert PDP-10 machine language instructions into a Bliss program in the syntactic form of a special function

$$op\ (\epsilon_1,\ \epsilon_2,\ \epsilon_3,\ \epsilon_4)$$

where

- op is one of the PDP-10 machine language mnemonics (see table below).
- ϵ_1 is an expression whose least significant 4 bits will become the accumulator (A) field of the compiled instruction. This expression must yield a value at compile time of a declared register name or a literal.
- ϵ_2 is an expression whose least significant 18 bits will become the address (Y) field of the compiled instruction.
- ϵ_3 is an expression whose least significant 4 bits will become the index (X) field of the compiled instruction.
- ϵ_4 is an expression whose least significant bit will become the indirect (I) bit of the compiled instruction.

(A table of machine language instruction mnemonics follows. Defaults for ϵ_1-ϵ_4 are 0.)

The 'value' of these machine language instructions is uniformly taken to be the contents of the register specified in the accumulator (A) field of the instruction. (This makes little sense in a few cases, but was adopted for uniformity.)

In order for the compiler to conserve space during compilation, the mnemonics for the machine language operators are not normally preloaded into the symbol table. Therefore, in order to use this feature of the language, it is necessary for the programmer to include one of the following special declarations

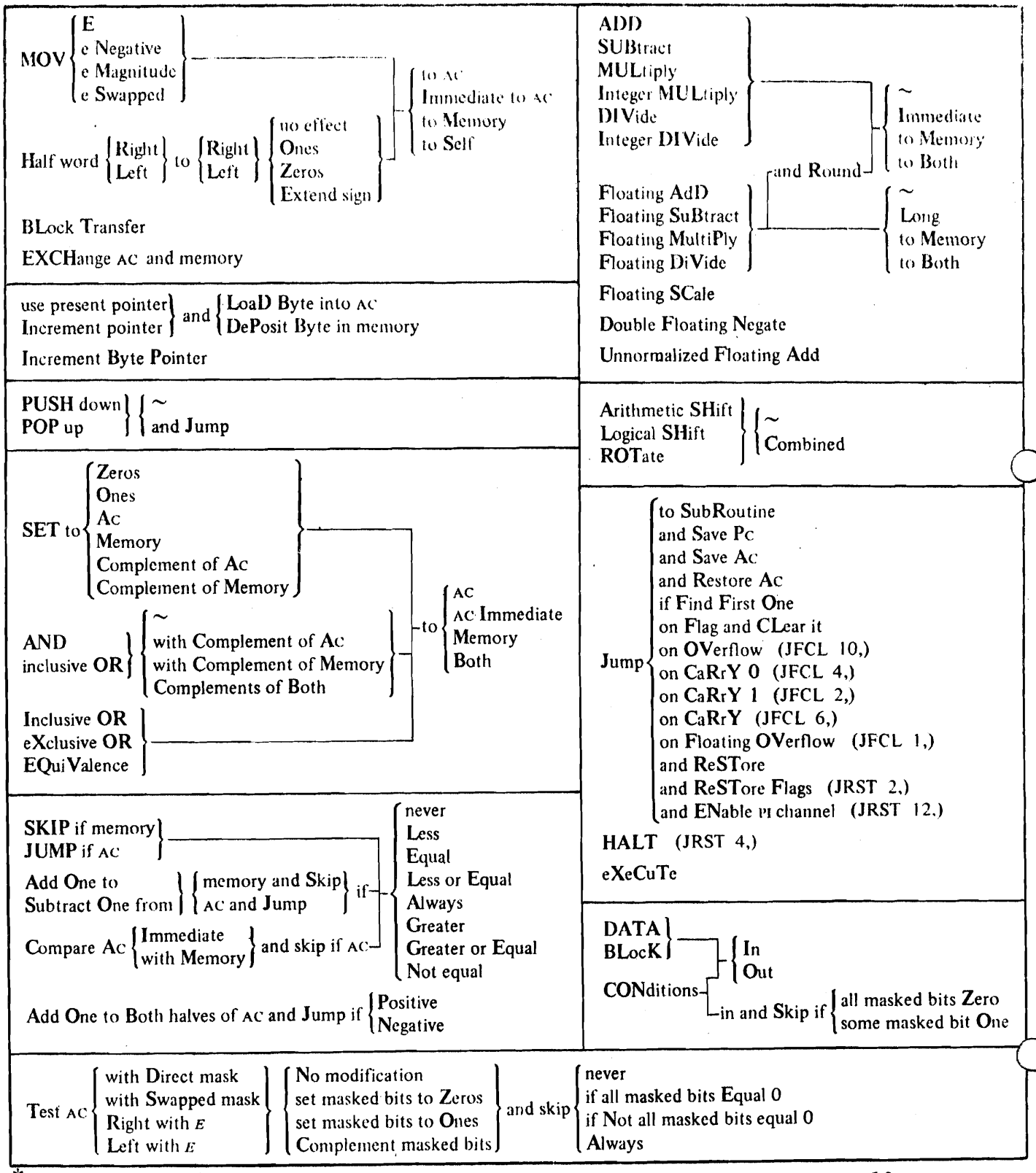
```
declaration  → MACHOP  mlist  | ALLMACHOP
mlist        → name  = e | mlist , name  = e
```

in the head of a block which embraces occurrences of these special functions.

(Note: The e's in an mlist must be the high order nine bits of the actual values of the machine operation and must evaluate at compile time.) Symbol table space for these names is released when the block in which the declaration occurs is exited.

NOTE: The description of fields ϵ_2, ϵ_3, ϵ_4 needs some simplification in the case where ϵ_2 is a name. The compiler attempts to produce a single instruction for the machine language expression whenever possible. For example, consider the expression MOVEM(5,A) where A is a local variable. The compiler, noting that the index register has been defaulted to zero, produces a 22 bit address using the F register for the index register field of the instruction.

PDP-10 Instruction Mnemonic Table*



*Reproduced with permission of Digital Equipment Corporation from the PDP-10 Reference Handbook.

1.4 Compilation Control

The actions of the compiler with respect to a program may be controlled by specifications a) in the initial input string from a TTY, b) in the module head, c) by a special SWITCHES declaration. Not all actions can be controlled from each of these places, but many can. Some actions once specified have a permanent effect (such as whether to create a high segment or low segment program) while the effect of others can be modified (such as listing control). The table in section 1.4.4 gives a list of various compiler actions and the associated switch and/or source language constructs which modify those actions. This list is subject to change.

1.4.1 Command Syntax

The general format of the initial command to Bliss is:

objdev: file.ext,lstdev:file.ext ← sorcdev:file.ext,...,sorcdev:file.ext

The "objdev:file.ext" and/or "lstdev:file.ext" may be omitted with the implication that the corresponding file is not to be generated. The ".ext" may be omitted on any of the file specifications and the following defaults assumed:

object file:	REL
listing file:	LST
source file:	BLI

As with DEC CUSP's, switches of the form /x (x=A,B,...,Z) may be placed anywhere in a command string.

1.4.2 Module Head

As explained in I.1.1 the syntax for a module is

module → MODULE name(parameters) = e ELUDOM

The 'parameters' field may contain various information which will affect the compiler's action with respect to the current program. The syntax of this field is

parameters → parameter | parameter,parameters

The allowed forms of 'parameter' are given in tabular form in section II.1.4.4 under the column headed "module head syntax".

1.4.3 SWITCHES Declaration

declaration → SWITCHES switch list
switch list → switch | switch, switch list

The SWITCHES declaration allows the user to set various switches which control the compiler's actions. The effect of a SWITCHES declaration is limited to the scope of the block in which the declaration is made. The various allowed forms of 'switch' are given in tabular form in section II.1.4.4 under the column headed "SWITCHES DECLARATION".

1.4.4 Actions

COMMAND SWITCH	MODULE HEAD SYNTAX	'SWITCHES' DECLARATION	ACTION
/L	LIST	LIST	Enable listing of the source text. This switch is assumed true initially.
/K	NOLIST	NOLIST	Disable listing of the source text.
/N	NOERS	NOERS	Do not print error messages on the TTY.
/M	MLIST	MLIST	Enable listing of the machine code generated.
/H	HISEG	-	Make this module a highsegment module. Initially modules are assumed to be two segments.
/I	INSPECT	INSPECT	When true this switch will cause a special word to be emitted immediately prior to each function or routine body. This word contains information to facilitate a SIMULA-like inspection mechanism (see IV.1.4). The default initial value of this switch is false.
-	NOINSPECT	NOINSPECT	This sets the inspection switch false.
/S	-	-	Enable listing of compiler statistics. Information relevant to the implementation will be printed at the end of compilation.
/X	SYNTAX	-	Syntax check only! No code will be generated - this speeds the compilation process and is therefore useful during the initial stages of program development.
-	DREGS=e	-	'e' specifies the number of 'declared'-type registers to be used. Unless specified this value is defaulted to a small number (three at the time of this writing).

COMMAND SWITCH	MODULE HEAD SYNTAX	'SWITCHES' DECLARATION	ACTION
-	RESERVE(e_1,...e_n)	-	Registers with absolute names e_1,...,e_n are reserved (usually for inter-module communication).
/O	OPTIMIZE	OPTIMIZE	Because of the possibility of computed addresses in Bliss programs, it is not possible for the compiler to determine whether optimization of sub-expressions is possible across ";"'s in a compound expression. Therefore the compiler operates in two modes - one in which it does optimize such common sub-expressions and one in which it does not. When the 'optimize' switch is true the compiler attempts to optimize across a ";". The default mode is for the switch to be true.
/U	NOOPTIMIZE	NOOPTIMIZE	Sets the optimization switch (see above) to false.
/E	EXPAND	EXPAND	Give trace of macro expansions.
-	NOEXPAND	NOEXPAND	Turn off trace of macro expansion. This is default initial state.
-	SREG = e	-	The user may use these to choose specific registers to be used as the S, V, B, and F, respectively.
-	VREG = e	-	
-	BREG = e	-	
-	FREG = e	-	
/C			Print a cross-reference to all identifiers at the end of compilation (assumes a listing is being printed).
/R	NORSAVE RSAVE	NORSAVE RSAVE	The compiler normally generates code to save all declarable registers around an EXCHJ operation. This default may be overriden by a /R, or NORSAVE. RSAVE reverts to the default.
/V	LOSEG	LOSEG	Force entire compilation into the low segment.

COMMAND SWITCH	MODULE HEAD SYNTAX	'SWITCHES' DECLARATION	ACTION
-	STACK (see text at right)	-	The syntax of the module head permits automatic allocation and initialization of the run-time stack. The syntax is STACK|STACK(<literal size>)| STACK=<explicit-stack> where <explicit-stack>::=<stype> <s-name-sz> <stype>::=GLOBAL|OWN|EXTERNAL <s-name-sz>::=(<ID><ss-OPTN>) <ss-OPTN>::=/<literal> The defaults are 'STACK'= STACK=OWN(STACK,#1000) 'STACK(lit)'= STACK-OWN(STACK,lit) etc.
/G	-	-	All routine names are forced to be 'global'.
	ENTRIES-(n_1,,n_m)		An 'entry' block is created at the beginning of the '.REL' file for the names n_1,n_2,...n_m. These names must subsequently be declared 'global' in the module. This permits FUDGE2 to be used to create a library.

1.1 Registers

The sixteen registers are divided into three main classes:

1. Reserved registers:

These registers are declared in the module head. Their scope is the entire module and they may also be accessed from within any global routine. They are never saved.

2. Bliss run-time registers:

After the reserved registers have been allocated, the lowest four remaining addresses are assigned as the run-time registers. In particular, if there are no reserved registers, 0 through 3 are assigned as the S, B, F, and V registers respectively. The names SREG, BREG, FREG, and VREG are available at the outermost blocks of the module and, as in the case of reserved registers, these names are accessible from within any global routine.

3. Temporary registers:

All the remaining registers fall into this class and are divided into two subclasses:

a. savable:

These registers are used for declared registers, control registers in incr-decr loops, and when necessary for computing temporary values. Any of these registers which are used in the body of a function or routine are saved in the prolog and restored in the epilog. Of course if F is not a global routine and F is within the scope of

IV. RUN TIME REPRESENTATION OF PROGRAMS

1.0 Introduction

In order to make the fullest possible use of Bliss, it is important to understand the run-time environment in which Bliss programs run. The address space is occupied by various types of information:

(1) program

(2) constants

(3) static size variable areas (globals and owns)

(4) stacks

Programs are 'pure' (they do not modify themselves) therefore program and constant areas are placed in contiguous, write-protected regions and may be shared (see the 'HIGSEG' switch declaration, section II.1.4). Static variable storage and stack space are placed in readable/writable memory. The key to understanding the run-time environment in the stack configuration and register allocation is illustrated in Figure IV.1. Each process (co-routine) has its own stack configured as shown in IV.1.

of register R, then R is not preserved. The user must declare the size of this block of registers in the module head. (DREGS =). These registers are allocated from the highest addresses.

b. non-savable:

These are the registers used for calculating intermediate results. They are saved at the call site of a function or routine only if they contain a needed result and are never saved in the prolog or epilog.

Comments:

a. If one wishes to load a collection of Bliss modules together, they must request precisely the same reserved registers and request the same number of savable temporaries.

b. The two classes of temporary registers are managed quite differently in that the savable registers obey a stack discipline (to minimize saving and restoring) and the non-savable are used in round-robin fashion (to lengthen the life of intermediate results). The present version of the compiler requires a minimum of 4 non-savable registers--i.e., the maximum value of DREGS = 8 - # of reserved regs. In general the compiler can produce better code if DREGS is kept to the minimum value which the lexical scope of declared registers and/or incr-decr loops allow.

1.2 The Stack and Functions

The first 17_{10} locations of each stack are reserved for state information (registers plus program counter) for a process when it is inactive. The use of these cells is explained more fully in 1.4. The configuration above these 17 state words depends upon the depth of nesting of function calls, but each such nested call involves a similar (not identical) use of the stack; Figure IV.1 illustrates a typical stack configuration after several nested functional calls. At a time when one of these functions is executing

(1) The S-register points to the highest assigned cell in the stack; the S-register is used to control the allocation of the stack area.

(2) The F-register points to the 'local base of stack'; below* the F-register are the parameters to the function and the return address. The stack cell actually pointed to by the F-register contains the previous value of the F-register at the time at which the current function was entered.

(3) The calling sequence which is used to enter a function (or routine) is

```
PUSH   S,p1          ;  push 1st parameter onto the stack
PUSH   S,p2          ;  push 2nd parameter onto the stack
 ...                 ...
PUSH   S,pn          ;  push nth parameter onto the stack
PUSHJ  S,FCN         ;  jump to the called function
SUB    S,[nooooon]   ;  delete the parameters
```

(4) Above the F-register are stored the "displays", $D_1 \ldots D_f$.

*'below' in the sense of decreasing address values.

One display is used for each lexical nesting of the declaration of the function which is currently executing. The value of the displays are the F-register values for the most recent recursive entries for the lexically embracing functions. The displays are needed and used to access variables global to the current functions but local to embracing functions. Such access is prohibited in routines, and consequently no displays are saved on a routine entry.

(5) Above the displays are saved any savable registers which are destroyed by the execution of the function body. These registers are restored before the function exits.

(6) Any local variables in the function are stored on top of the saved registers. Space is acquired/deleted for locals on block entry/exit by simply adding/subtracting a constant to the S-register. Some of these locals are automatically generated by the compiler.

(7) An excessive number of declared registers, or the evaluation of an unbelievably complex expression may exhaust the available registers, forcing the area above the locals to be used for storing partial results of an expression evaluation.

(8) The V-register is used to return the value of the function or routine.

Figure IV.2 illustrates the code generated surrounding the body of a function. The code surrounding a routine body is identical with the exception that the displays are never saved. In this illustration the S, B, F, and V registers are shown occupying physical registers 0-3. In practice other registers may be chosen if these registers are reserved in the module head.

Figure IV.1

Stack Structure and Registers for a Process

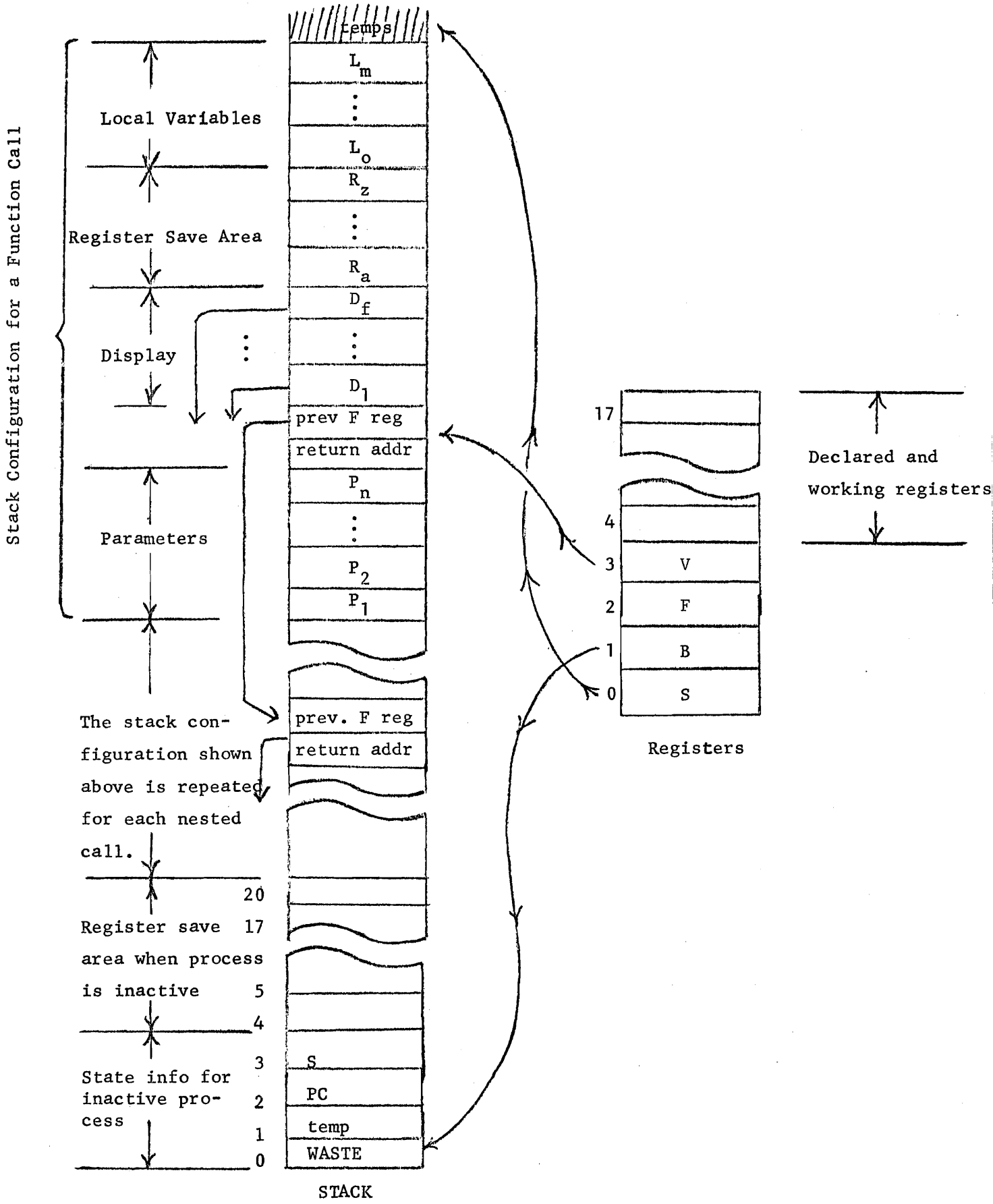


Figure IV.2

Function Prolog and Epilog

```
FCN:  PUSH   S,F        ;    save old F-register
      PUSH   S,1(F)     ;    copy display zero
      ...               ...
      PUSH   S,f(F)     ;    copy display f
      HRRZ   F,S        ;    set up new F
      SUBI   F,f        ;    subtract no. displays
      PUSH   S,F        ;    new display created
      PUSH   S,Ra       ;    save register
      ...               ...
      PUSH   S,Rz       ;    save register
      { BODY OF FUNCTION OR ROUTINE }
      POP    S,Rz       ;    restore register
      ...               ...
      POP    S,Ra       ;    restore register
      SUB    S,[(f+1)00000(f+1)] ; eliminate displays
      POP    S,F
      POPJ   S,
```

Not Generated For Routines

Figure IV.3

Block Entry and Exit

```
BENTER:   MOVEM    R1,l+1(F)          ; save in-use working registers
          ...                         ...
          MOVEM    Rj,l+j(F)          ; save in-use working registers
          ADD      S ,[n00000n]       ; INCR S-register by no. locals in blk

BEXIT:    SUB      S,[(n+j)00000(n+j)] ; DECR S-register by no. locals in blk
                                       ; (note: in-use regs left in stack,
                                       ; re-loaded only when used)
```

1.3 Access to Variables

This section briefly indicates the mechanisms by which generated code accesses various types of variables (formals, owns and globals, locals, etc.) The exact addressing scheme used by the compiler in any particular case is highly dependent upon the context; however, the following material should aid in understanding the overall strategy.

(a) OWN and GLOBAL variables are accessed directly.

(b) Formal parameters of the current routine are accessed negatively with respect to the F-register. If the current routine has n formals, then the ith one is addressed by

$$(-n + i - 2)(F)$$

(c) Local variables of the current routine are accessed positively with respect to the F-register. To access the ith local cell, one uses

$$(i + d + r + 1)(F)$$

where d is the number of displays saved and r is the number of registers saved on function entry.

(d) Formal parameters and local variables which are not declared in the currently executing function are accessed through the display. The appropriate display is copied into one of the working registers then accessed by indexing through that register in a manner similar to that shown in (b) or (c) above.

The first four characters of the name introduced in the module head is used to name various regions in the produced code. These names are declared "external" and therefore available in DDT. If 'XXXX' are the

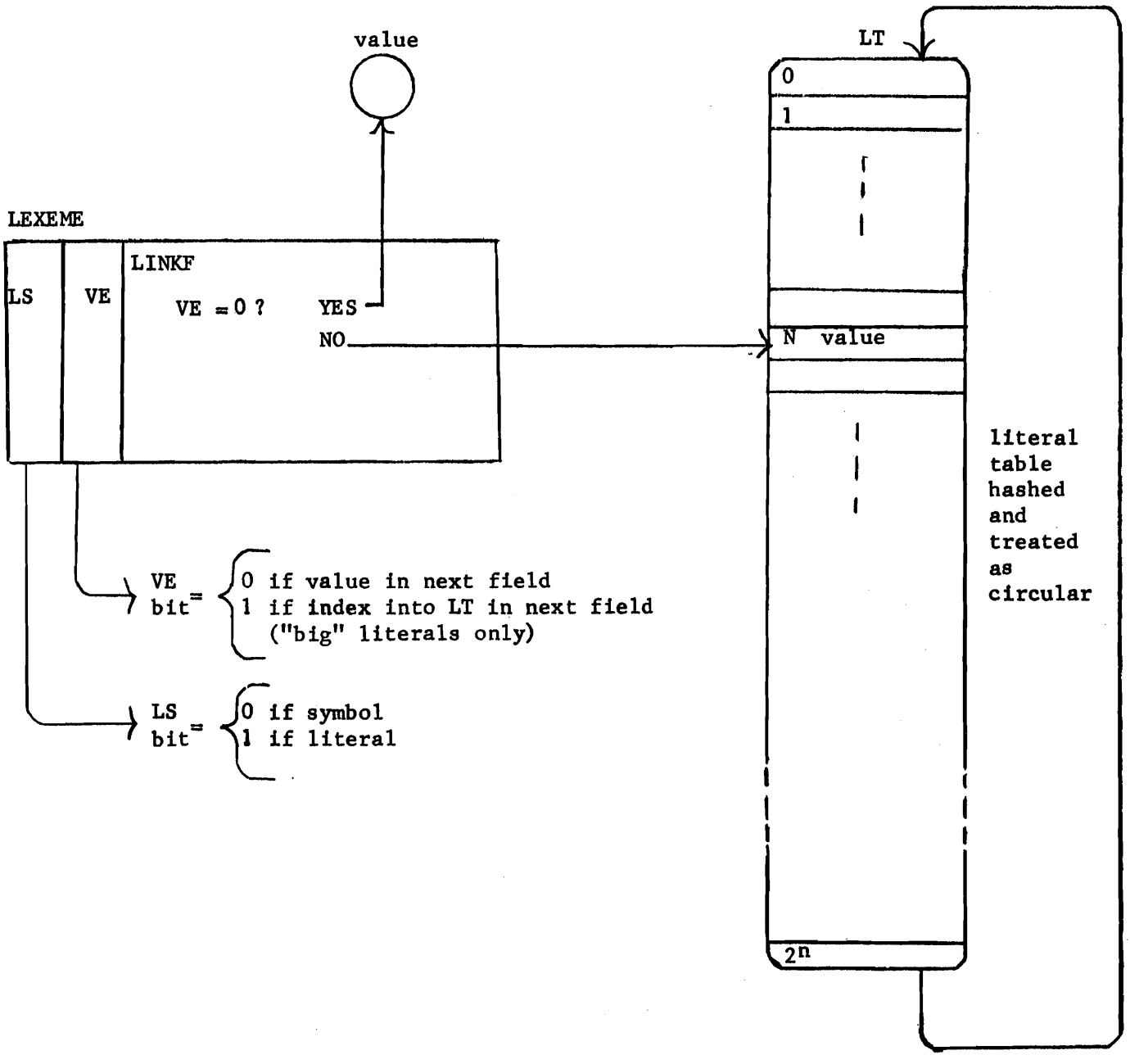
first four characters of the module name, then

XXXX	is the location of the first instruction in the main body of the module.
XXXX.F	is the location of the "literal" area which contains constants generated by the compiler.
XXXX.O	is the location of the "own" area in which is stored all variables declared 'own' in the module.
XXXX.G	is the location of the "global" area in which is stored all variables declared "global" in the module.
XXXX..	is the module name recognized by DOT.
XXXX.P	is the first location of the "plit" area.

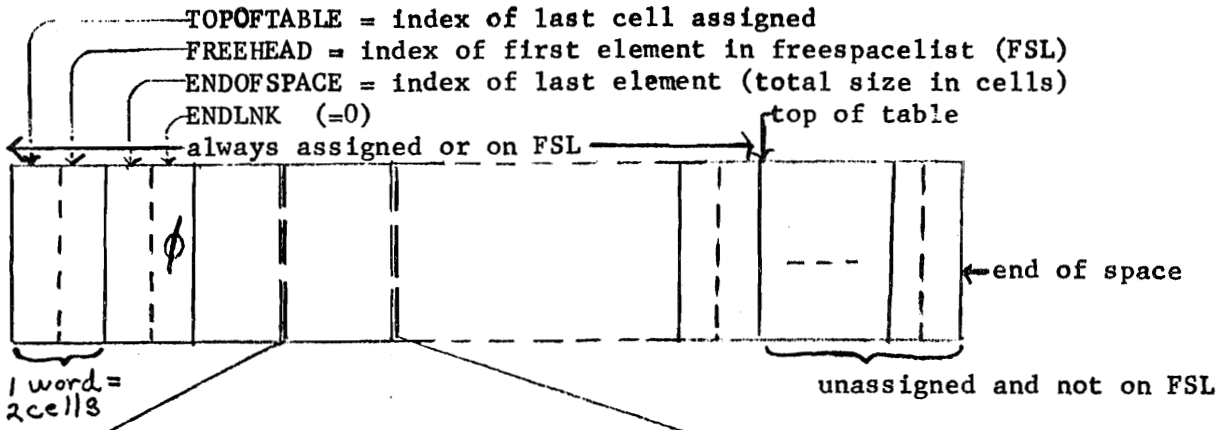
V. COMPILER IMPLEMENTATION

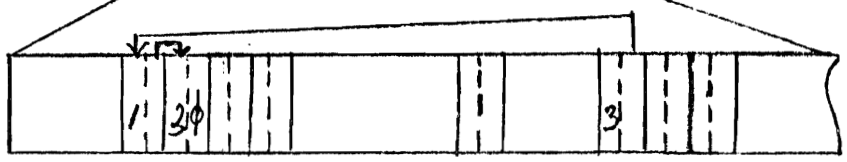
This table contains a description of the implementation of the Bliss compiler. At every instant of time this section will necessarily be incomplete and possibly erroneous. It will be extended and corrected as time permits and the compiler changes.

The initial contents of the section is a set of diagrams of the major tables in the compiler.



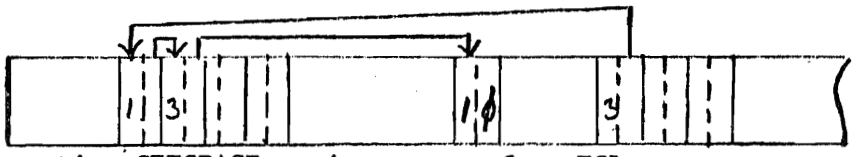
THE LITERAL TABLE



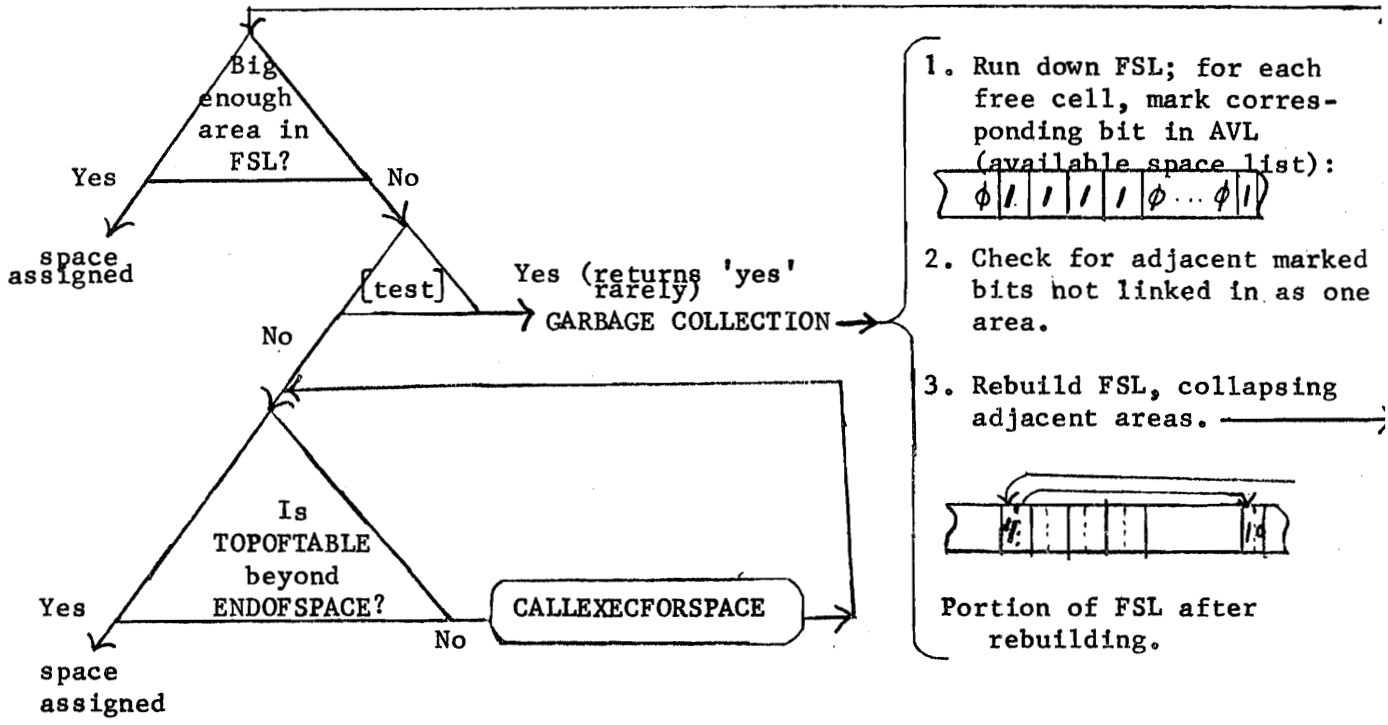


FSL = set of areas linked together such that 0th cell contains size of area, 1st cell has link to next area. (End of chain has 0 link.)

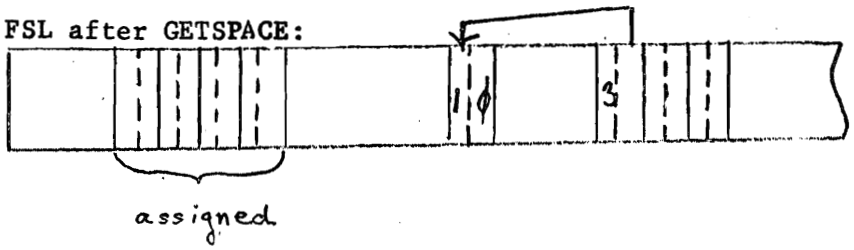
routine RELEASESPACE links areas into FSL:



routine GETSPACE assigns areas from FSL:

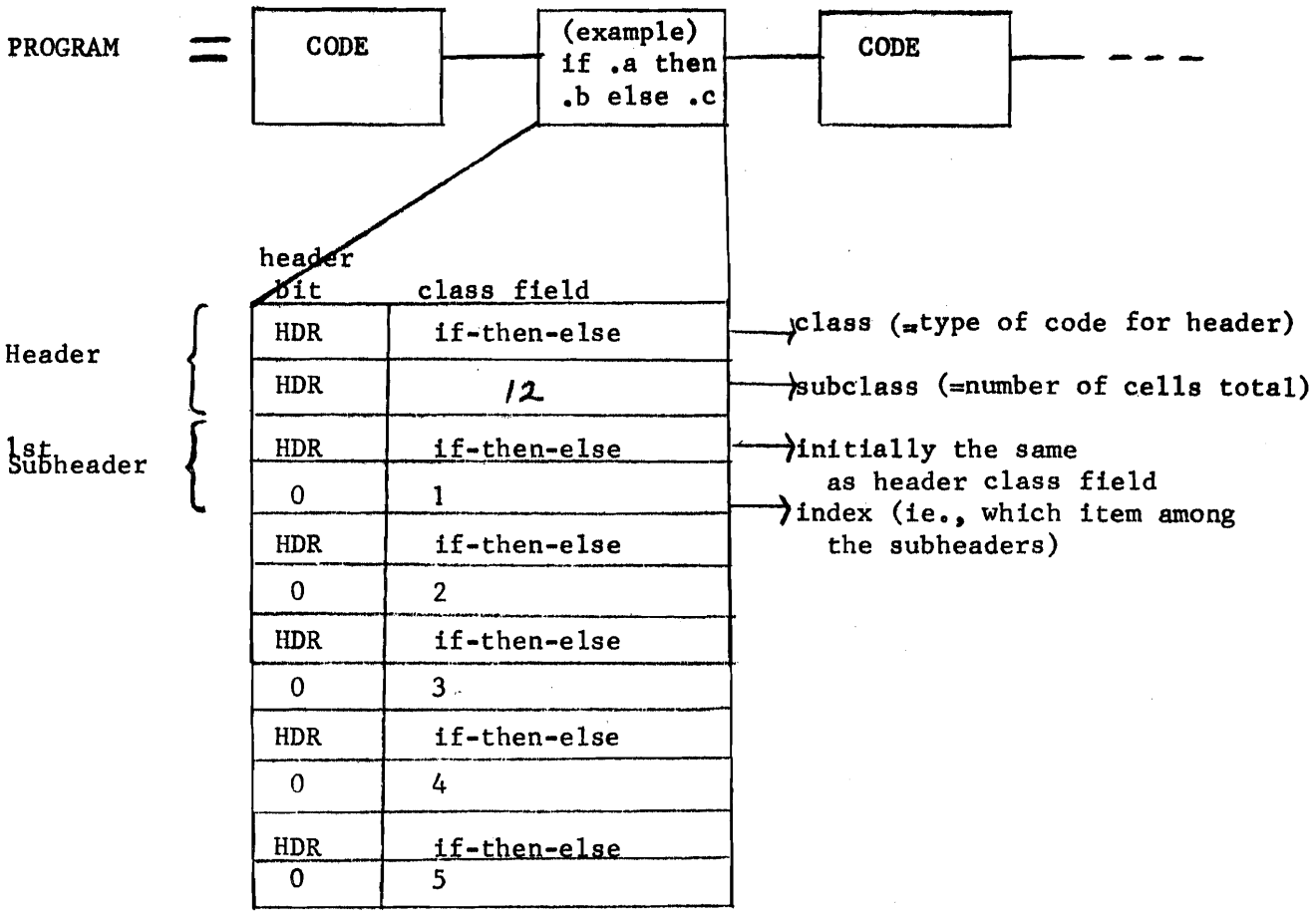


FSL after GETSPACE:

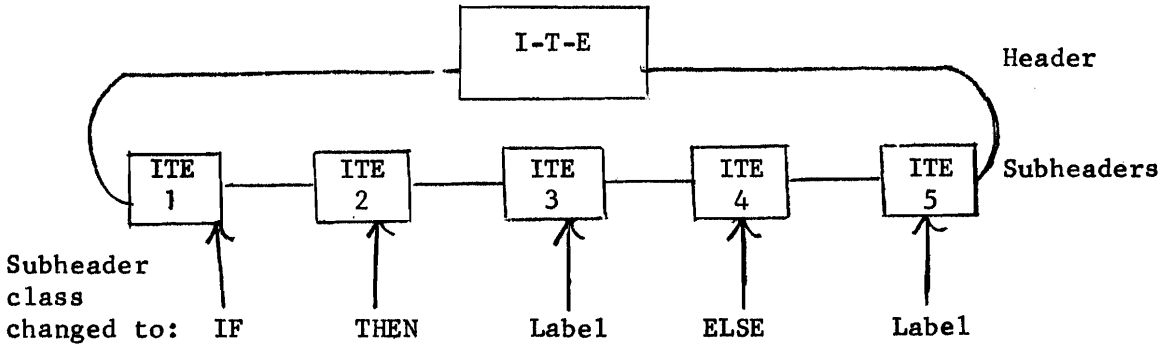


FREE SPACE LIST

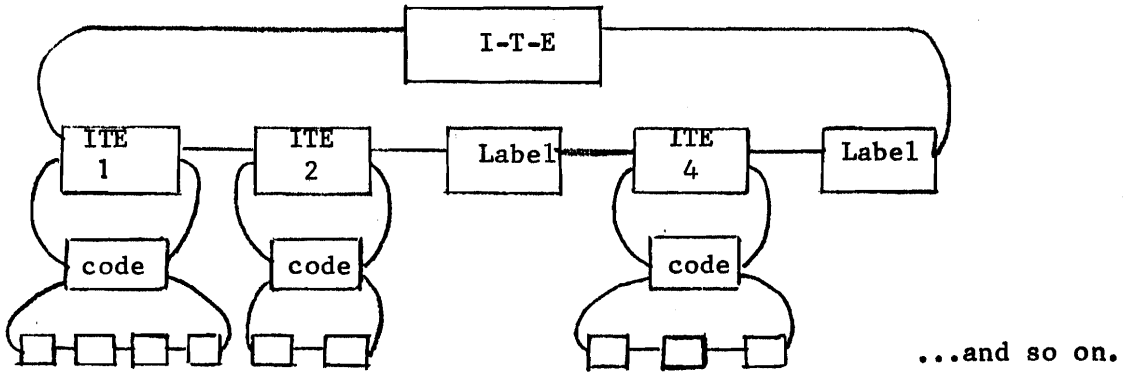
The entire program is a linked list, and is itself linked to global variable PROGRAM:



After processing, the above is equivalent to:

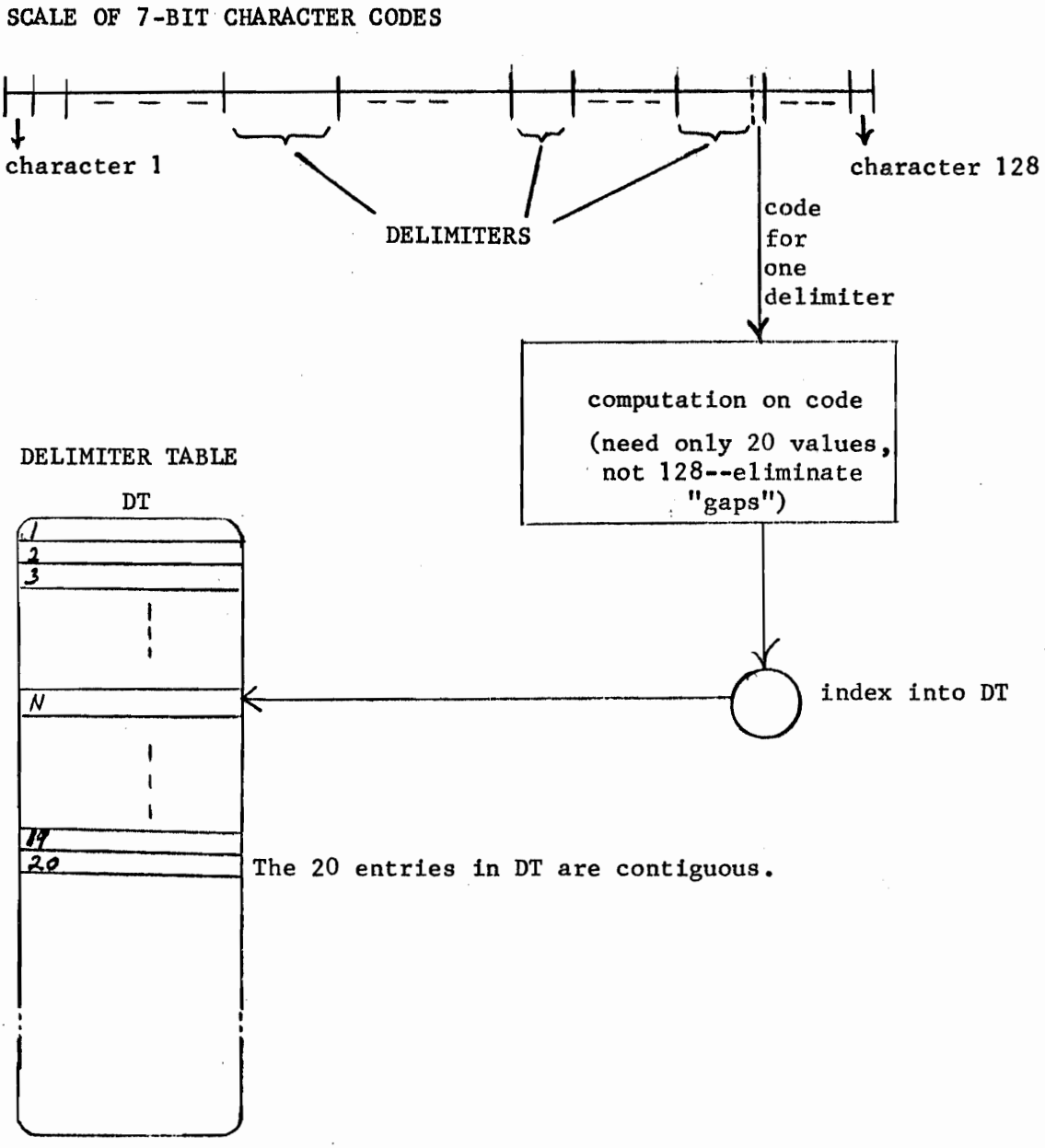


After more processing, the above may change to:

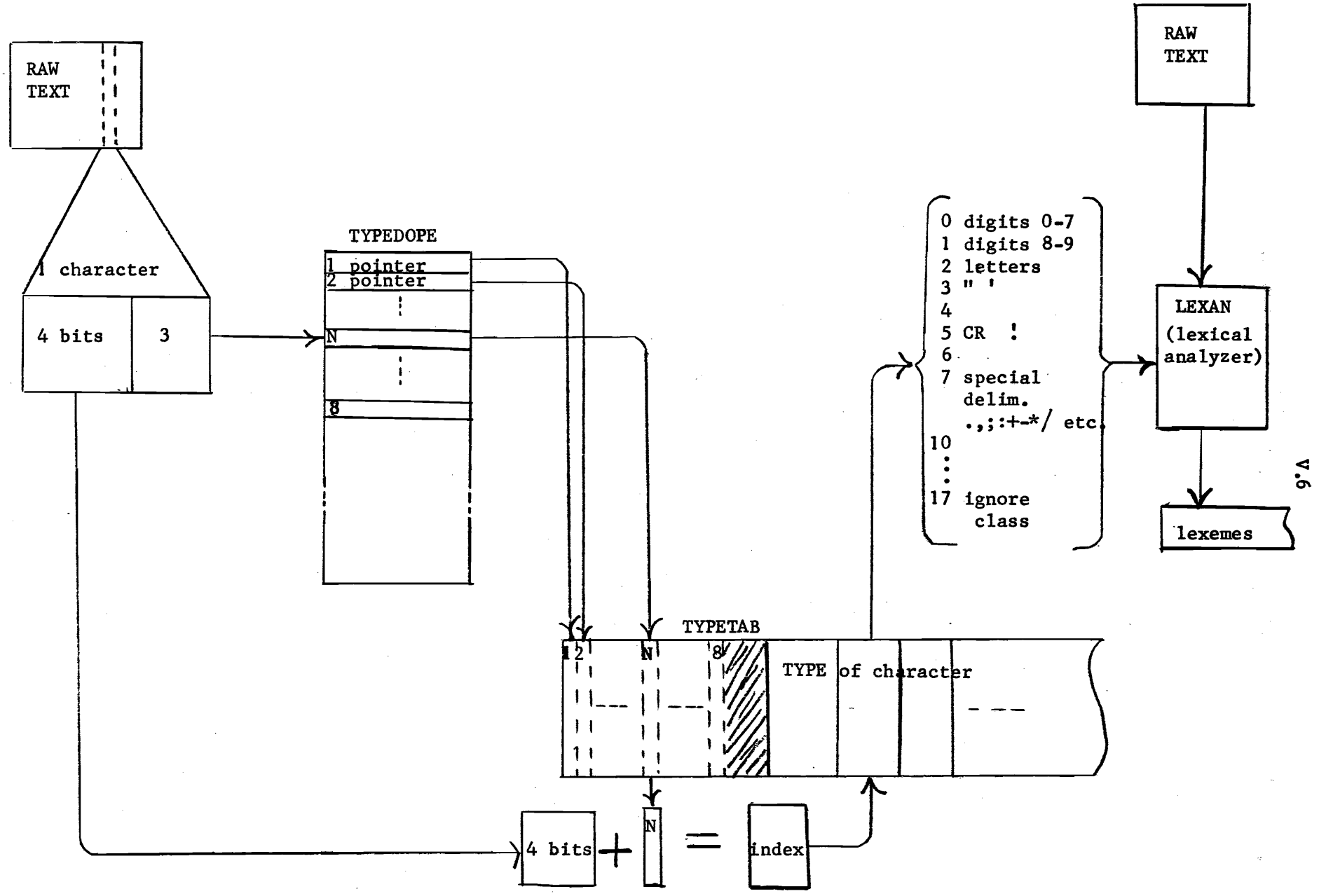


...and so on.

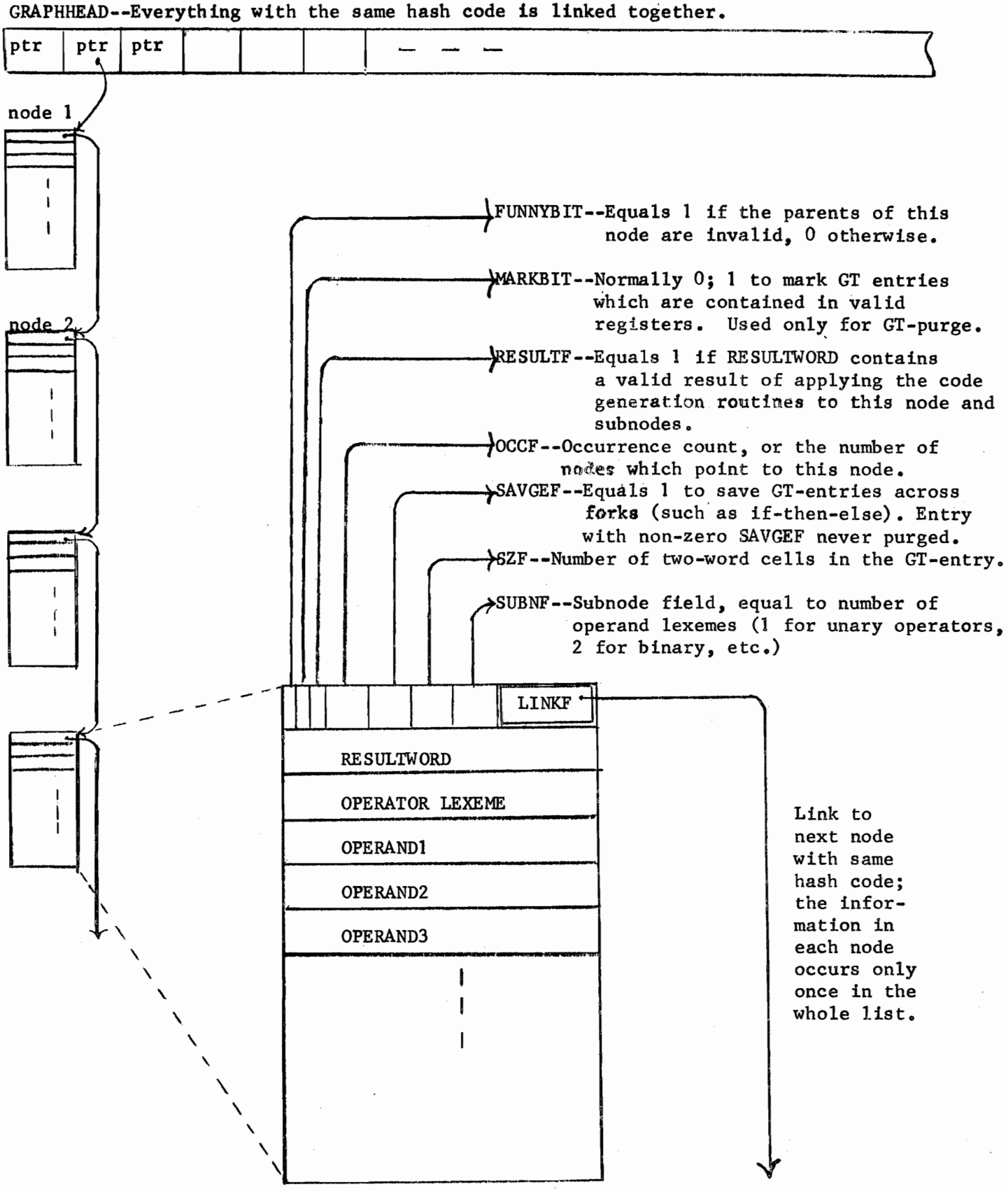
THE CODE TABLE



DELIMITER TABLE

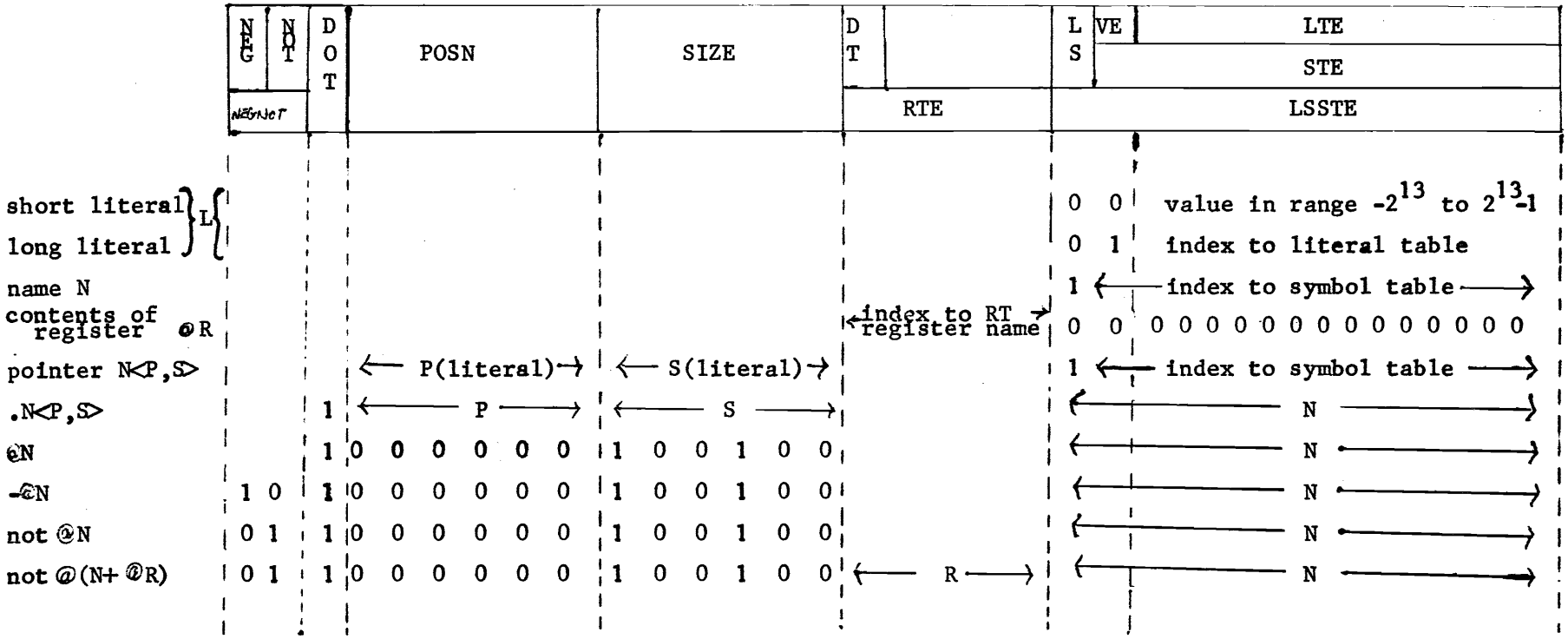


THE TYPE TABLE



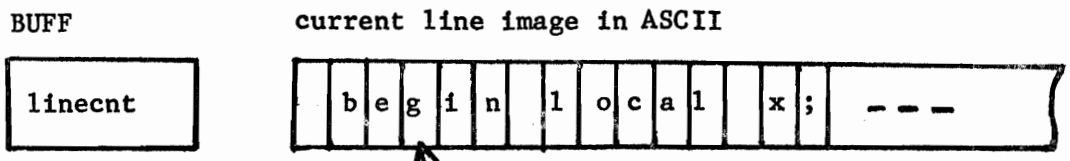
THE GRAPH TABLE

OPERAND LEXEME



NOTES:

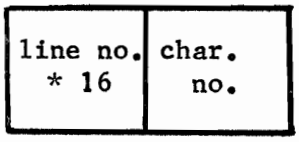
1. Neg and Not may not both be set.
   S=0 indicates P,S unset.
   The name of a declared register is a literal.
2. A graph-table lexeme is considered a special form of operand lexeme and is distinguished by a left-half equal to #077777.



Values associated with BUFF are:

(1) PBUFF, points to next character to be scanned

(2) CHAR, contains the character to be scanned



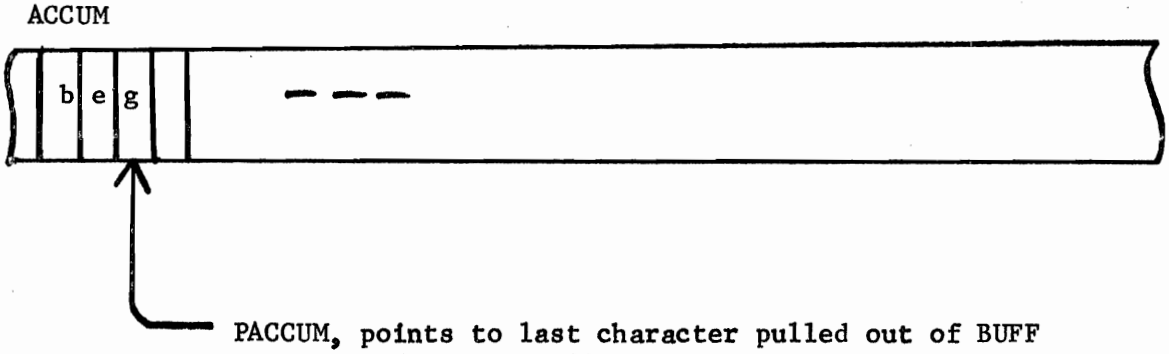
(3) NCBUFF, used for printing error messages-- points to place in line where error was detected

(4) VALIDBUF, = $\begin{cases} 0 \text{ if this line has been printed} \\ 1 \text{ if this line has } \underline{\text{not}} \text{ been printed} \end{cases}$

The sequence is:

READ a line
PROCESS that line
PRINT previous line (check VALIDBUF first)
READ next line
(repeat)

As characters are read, ACCUM is being built:

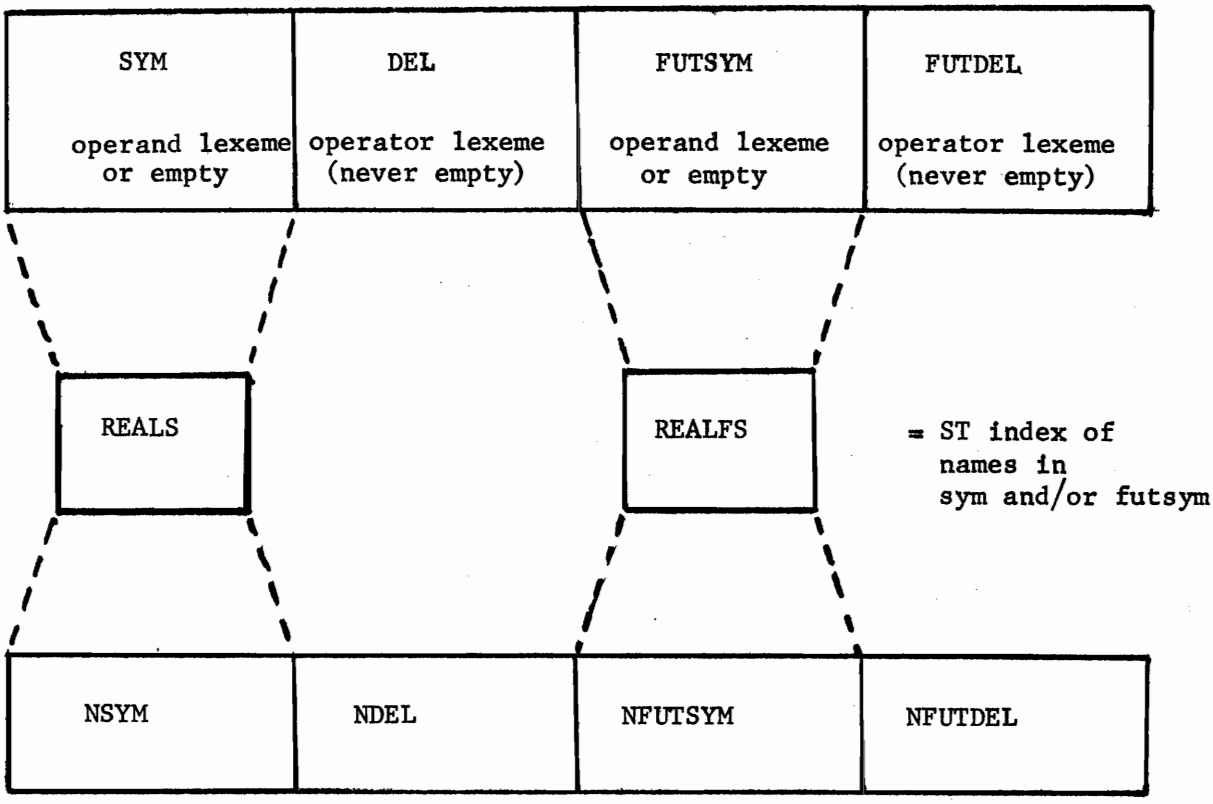


PACCUM, points to last character pulled out of BUFF and put into ACCUM

BUFF and ACCUM

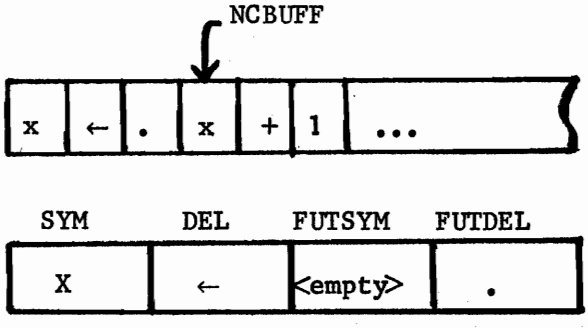
The major purpose of the lexical analyzer is to maintain the WINDOW; routine WRUND (read until next delimiter) keeps the WINDOW filled. Meanwhile, another window-like structure remembers the NCBUFF for lexemes in the WINDOW, for use when errors are detected.

WINDOW

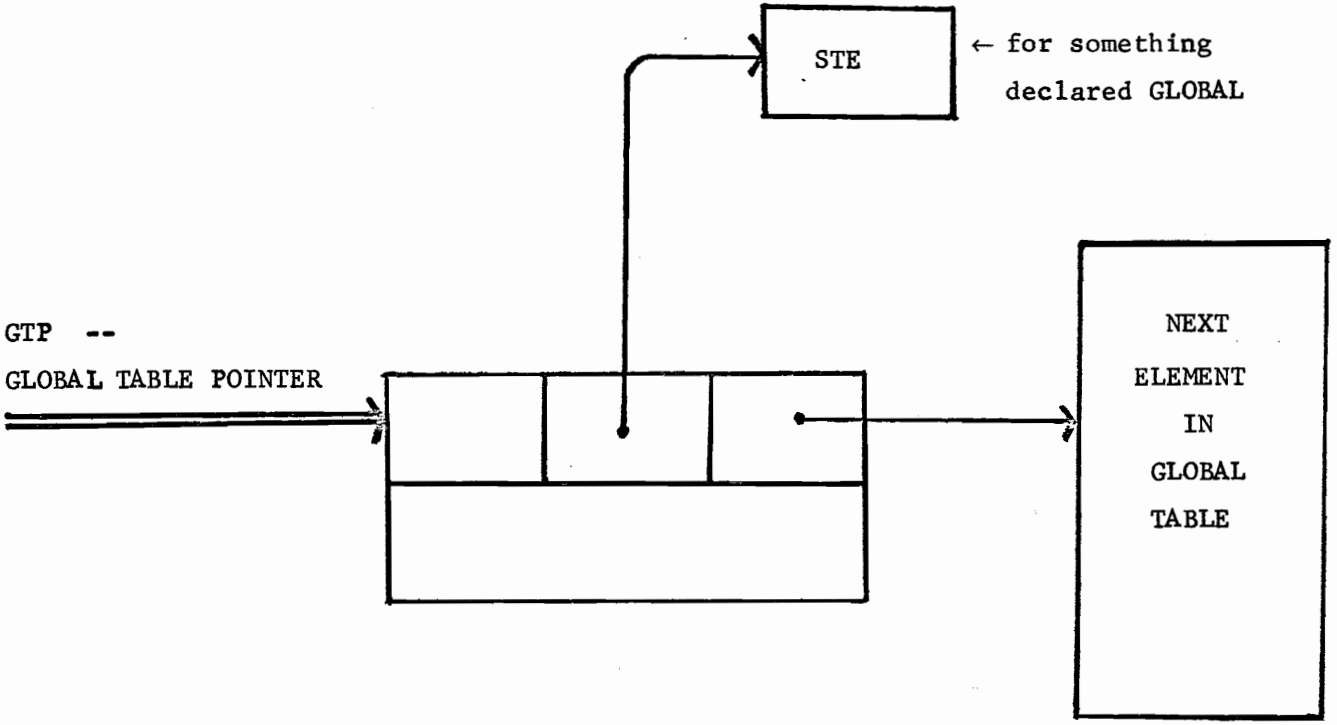


The above are values of NCBUFF for these atoms.

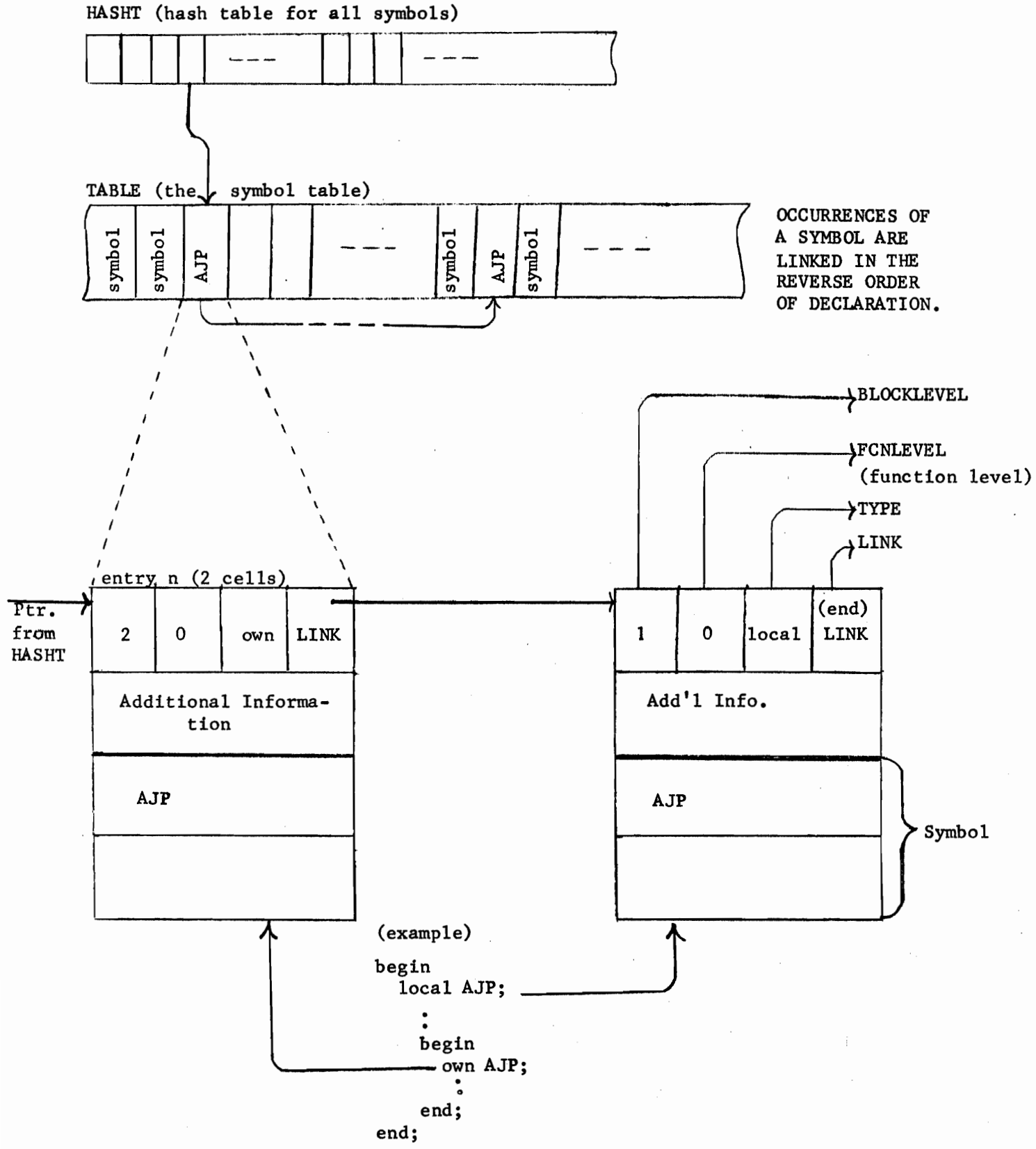
For x←.x+1 ... the BUFF and WINDOW look like:



THE WINDOW



THE GLOBAL TABLE

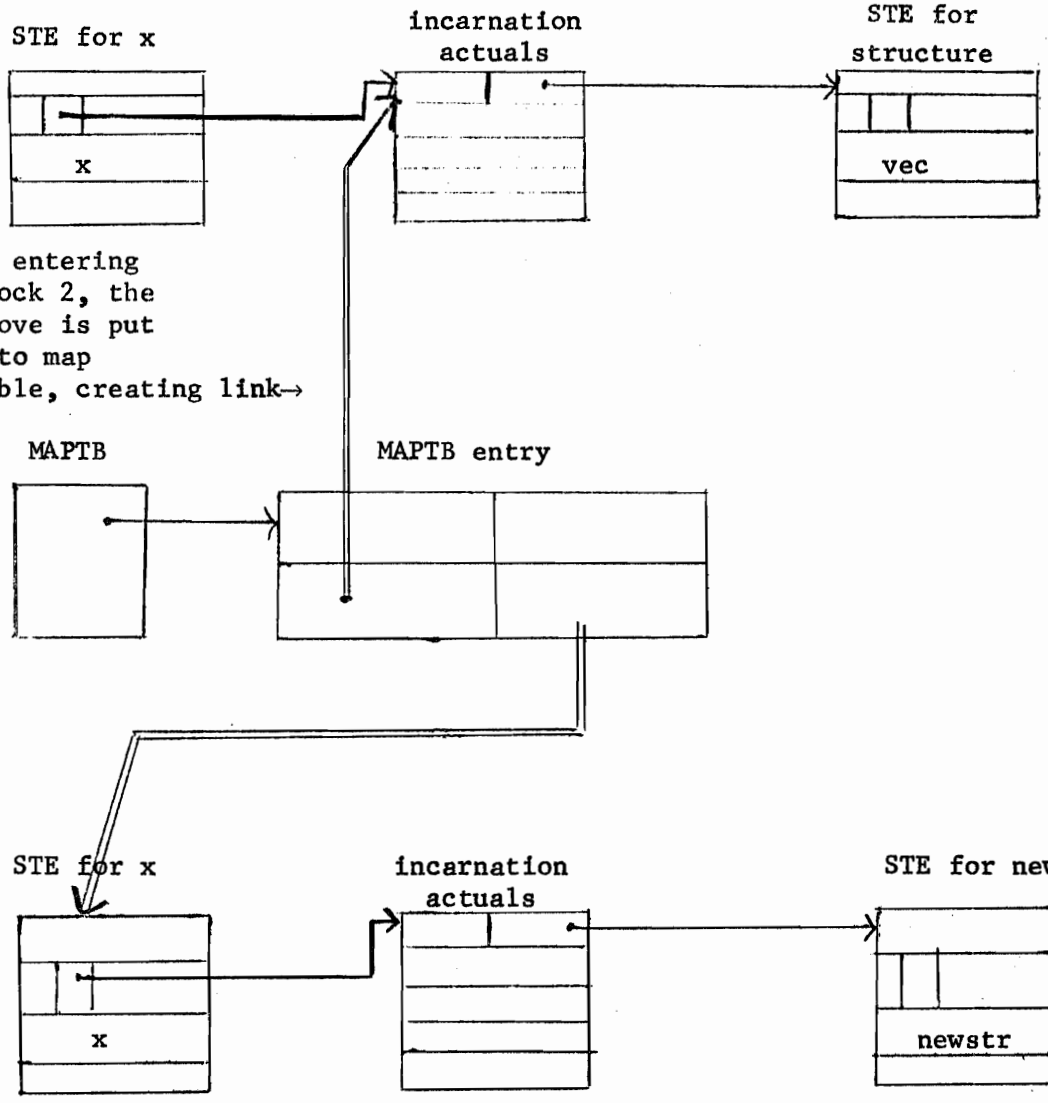


THE SYMBOL TABLE

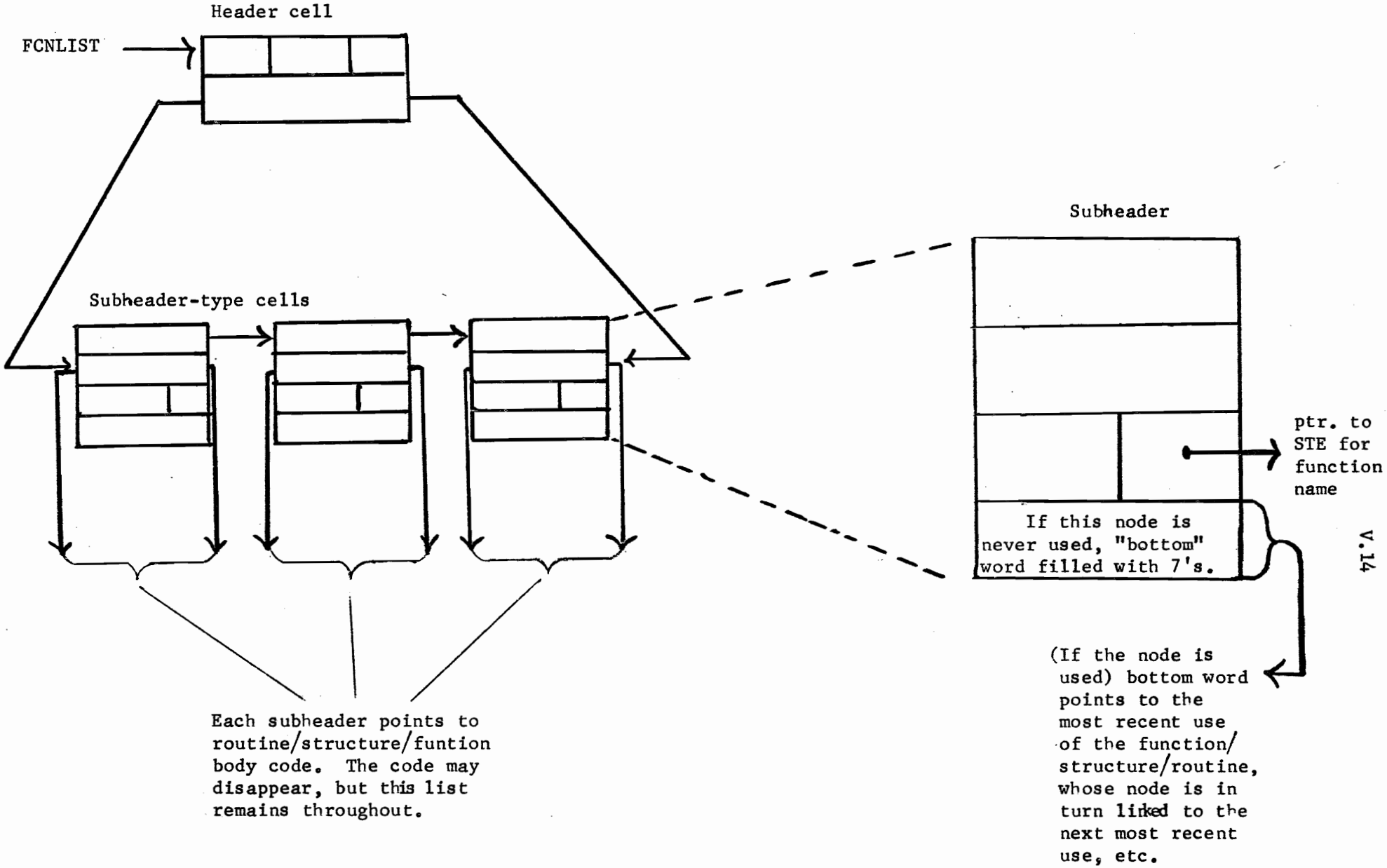
For the code:

```
structure vec[i] = (.vec+.i)<0,36>;
local x;
map vec x;
:
x[5]← 1;
    begin
    map newstr x;
    end;
x[3]← 4;
:
```

the STE would look like the following, until block 2 is entered:



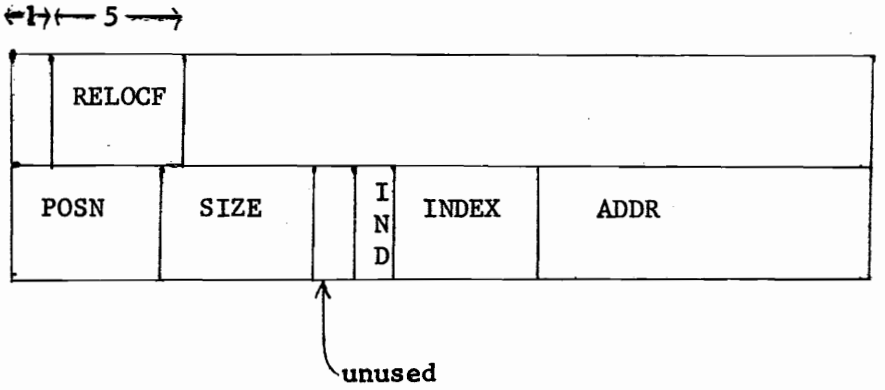
A unique map table exists for each block level; only "remapped" variables have entries in the table.



THE FUNCTION LIST

POINTER

RELOCF indicates whether ADDR is OWN, GLOBAL, etc.



POINTER TABLE

APPENDIX A: SYNTAX

module → MODULE name (parameters) e ELUDOM

block → BEGIN blockbody END | (blockbody)

compoundexpression → BEGIN expressionsequence END | (expressionsequence)

blockbody → declarations; expressionsequence

declarations → declaration | declaration; declarations

expressionsequence → | e | e; expressionsequence

comment → | ! restofline endoflinesymbol | % stringwithnopercent %

literal → number | quotedstring

number → decimal | octal | floating

decimal → digit | decimal digit

octal → # oit | octal oit

floating → decimal.decimal | decimal.decimal exponent

exponent → E decimal | E + decimal | E - decimal

digit → 0|1|2 --- |9

oit → 0|1|2 --- |7

quotedstring → leftadjustedstring | rightadjustedstring

leftadjustedstring → 'string'

rightadjustedstring → "string"

e → simpleexpression | controlexpression

simpleexpression → p11 ← e | p11

p11 → p10 | p11 XOR p10 | p11 EQV p10

p10 → p9 | p10 OR p9

p9 → p8 | p9 AND p8

p8 → p7 | NOT p7

p7 → p6 | p6 relation p6

p6 → p5 | - p5 | p6 + p5 | p6 - p5

p5 → p4 | p5 * p4 | p5 / p4 | p5 MOD p4

p4 → p3 | p4 ↑ p3

p3 → p2 | .p3 | @p3 | \ p3

p2 → p1 | p1 <pointerparameters>

p1 → literal |

name |

name [elist] =

p1 (elist) |

p1() |

block |

compoundexpression

relation → EQL | NEQ | LSS | LEQ | GTR | GEQ

pointerparameters → position, size modification

modification → | , index | , index, indirect

position → | e

size → | e

index → | e

indirect → | e

controlexpression → conditionalexpression | loopexpression |

choiceexpression | escapeexpression | coroutineexpression

conditionalexpression → IF e_1 THEN e_2 | IF e_1 THEN e_2 ELSE e_3

loopexpression → WHILE e_1 DO e_2

loopexpression → UNTIL e_1 DO e_2

loopexpression → DO e_2 WHILE e_1

loopexpression → DO e_2 UNTIL e_1

loopexpression → INCR name FROM e_1 TO e_2 BY e_3 DO e_4

loopexpression → DECR name FROM e_1 TO e_2 BY e_3 DO e_4

escapeexpression → environment level escapevalue | RETURN escapevalue

environment → EXIT | EXITBLOCK | EXITCOMPOUND | EXITLOOP | EXITCONDITIONAL
EXITCASE | EXITSET | EXITSELECT

level → | [e]

escapevalue → | e

choiceexpression → CASE elist OF SET expressionset TES

elist → e | e, elist

expressionset → |e|; expressionset | e ; expressionset

choiceexpression → SELECT elist OF NSET nexpressionset TESN

nexpressionset → | ne | ne; nexpressionset

ne → e:e

coroutineexpression → CREATE e_1 (elist) AT e_2 LENGTH e_3 THEN e_4 |
EXCHJ (e_4, e_5)

declaration → functiondeclaration|structuredeclaration|
binddeclaration|macrodeclaration|
allocationdeclaration|mapdeclaration

allocationdeclaration → allocatetype msidlist

allocatetype → GLOBAL|REGISTER|OWN|LOCAL|EXTERNAL

msidlist → msidelement|msidelement, msidlist

msidelement → structure sizedchunks

structure → | structurename

sizedchunks → sizedchunk|sizedchunk: sizedchunks

sizechunk → idchunk|idchunk [elist]

idchunk → name|name:idchunk

```
mapdeclaration → MAP msidlist

binddeclaration → BIND equivalencelist

equivalencelist → equivalence | equivalence, equivalencelist

equivalence → msidelement = e

structuredeclaration → STRUCTURE name structureformallist = structuresize

structureformallist → | [namelist]

structuresize → | [e]

function declaration → FUNCTION name (namelist) = e |
                       FUNCTION name = e
                       ROUTINE name(namelist) = e |
                       ROUTINE name = e

pl → pl (elist) | pl ( )

elist → e | elist, e

functiondeclaration → GLOBAL ROUTINE name (namelist) = e |
                      GLOBAL ROUTINE name = e

functiondeclaration → EXTERNAL nameparlist |
                      FORWARD nameparlist

        nameparlist → namepar | nameparlist, namepar

        namepar     → name (e)

macrodeclaration → MACRO definitionlist

    definitionlist → definition | definitionlist, definition

    definition → name  (namelist) = stringwithout$ $ |
                 name  = stringwithout$ $

macrocall → name  (balancedstringlist) | name

balancedstringlist → balancedstring | balancedstringlist, balancedstring

declaration → MACHOP mlist | ALLMACHOP

mlist → name = e | mlist, name = e
```

module → MODULE name(parameters) = e ELUDOM

parameters → parameter | parameter,parameters

declaration → SWITCHES switchlist

switchlist → switch | switch, switchlist

name → letter | name letter | name digit

letter → A|B|...|Z|a|b|...|z

APPENDIX B: INPUT-OUTPUT CODES*

The table beginning on the next page lists the complete teletype code. The lower case character set (codes 140-176) is not available on the Model 35, but giving one of these codes causes the teletype to print the corresponding upper case character. Other differences between the 35 and 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the console teletype, and the definitions bear no necessary relation to the use of the codes in conjunction with the PDP-10 software.

The line printer has the same codes and characters as the teletype. The 64-character printer has the figure and upper case sets, codes 040-137 (again, giving a lower case code prints the upper case character). The "96"-character printer has these plus the lower case set, codes 040-176. The latter printer actually has only ninety-five characters unless a special character is "hidden" under the delete code, 177. A hidden character is printed by sending its code prefixed by the delete code. Hence a character hidden under DEL is printed by sending the printer two 177s in a row.

Besides printing characters, the line printer responds to ten control characters, HT, CR, LF, VT, FF, DLE and DC1-4. The 128-character printer uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control characters that affect the printer and also under null and delete. In all cases, prefixing DEL causes the hidden character to be printed. The extra thirty-three characters that complete the set are ordered special for each installation.

The first page of the table of card codes [*pages*] lists the column punch required to represent any character in the two DEC codes. The octal codes listed are those used by the PDP-10 software. In other words, when reading cards, the Monitor translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. The remaining pages of the table show the relationship between the DEC card codes and several IBM card punches. Each of the column punches is produced by a single key on any punch for which a character is listed; the character being that which is printed at the top of the card.

---

*This appendix reproduced with the permission of Digital Equipment Corporation from the PDP-10 Reference Handbook.

TELETYPE CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 35.
1	001	SOH	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also EOA, end of address. Control B.
0	003	ETX	End of text; also EOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is . . . ") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you . . . ?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 35.
0	011	HT	Horizontal tab. Control I on Model 35.
0	012	LF	Line feed or line space (NEW LINE), advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 35.
1	013	VT	Vertical tab (VTAB). Control K on Model 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DC0).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (~~TAPE~~, AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is generated by control shift K on Model 35, but the Monitor translates it to 175.
1	034	FS	File separator. Control shift L on Model 35.
0	035	GS	Group separator. Control shift M on Model 35.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	036	RS	Record separator. Control shift N on Model 35.
1	037	US	Unit separator. Control shift O on Model 35.
1	040	SP	Space.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052	*	Repeats on Model 37.
0	053	+	
1	054	,	
0	055	-	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	Ø	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	:	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[	Shift K on Model 35.
0	134	\	Shift L on Model 35.
1	135	]	Shift M on Model 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140		Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.
1	171	y	
1	172	z	
0	173	{	▲
1	174	|	
0	175	}	This code generated by ALT MODE on Model 35. ▲
0	176	~	This code generated by ESC key (if present) on Model 35, but the Monitor translates it to 175.
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 35 local line feed.
LOC CR	Model 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 21-character message.

CARD CODES

▲ Character	PDP-10 ASCII	DEC 029	DEC 026		Character	PDP-10 ASCII	DEC 029	DEC 026
Space	040	*None*	*None*		@	100	8 4	8 4
!	041	11 8 2	12 8 7		A	101	12 1	12 1
"	042	8 7	0 8 5		B	102	12 2	12 2
#	043	8 3	0 8 6		C	103	12 3	12 3
$	044	11 8 3	11 8 3		D	104	12 4	12 4
%	045	0 8 4	0 8 7		E	105	12 5	12 5
&	046	12	11 8 7		F	106	12 6	12 6
'	047	8 5	8 6		G	107	12 7	12 7
(	050	12 8 5	0 8 4	▲	H	110	12 8	12 8
)	051	11 8 5	12 8 4	▲	I	111	12 9	12 9
*	052	11 8 4	11 8 4		J	112	11 1	11 1
+	053	12 8 6	12		K	113	11 2	11 2
,	054	0 8 3	0 8 3		L	114	11 3	11 3
−	055	11	11		M	115	11 4	11 4
.	056	12 8 3	12 8 3		N	116	11 5	11 5
/	057	0 1	0 1		O	117	11 6	11 6
0	060	0	0		P	120	11 7	11 7
1	061	1	1		Q	121	11 8	11 8
2	062	2	2		R	122	11 9	11 9
3	063	3	3		S	123	0 2	0 2
4	064	4	4		T	124	0 3	0 3
5	065	5	5		U	125	0 4	0 4
6	066	6	6		V	126	0 5	0 5
7	067	7	7		W	127	0 6	0 6
8	070	8	8		X	130	0 7	0 7
9	071	9	9		Y	131	0 8	0 8
:	072	8 2	11 8 2 *or* 11 0		Z	132	0 9	0 9
;	073	11 8 6	0 8 2		[	133	12 8 2	11 8 5
<	074	12 8 4	12 8 6		\	134	11 8 7	8 7
=	075	8 6	8 3		]	135	0 8 2	12 8 5
>	076	0 8 6	11 8 6		↑	136	12 8 7	8 5
?	077	0 8 7	12 8 2 *or* 12 0		←	137	0 8 5	8 2

Binary	7 9
Mode Switch	12 0 2 4 6 8
End of File	12 11 0 1

The octal codes given above are those generated by the Monitor from the column punches. The card reader interface actually supplies a direct binary equivalent of the column punch, as listed in the following two pages.

Column Punch	Character	Octal	Column Punch	Character	Octal
None	*Space*	0000	12 9	I	4001
0	0	1000	11 1	J	2400
1	1	0400	11 2	K	2200
2	2	0200	11 3	L	2100
3	3	0100	11 4	M	2040
4	4	0040	11 5	N	2020
5	5	0020	11 6	O	2010
6	6	0010	11 7	P	2004
7	7	0004	11 8	Q	2002
8	8	0002	11 9	R	2001
9	9	0001	0 1	/	1400
12 1	A	4400	0 2	S	1200
12 2	B	4200	0 3	T	1100
12 3	C	4100	0 4	U	1040
12 4	D	4040	0 5	V	1020
12 5	E	4020	0 6	W	1010
12 6	F	4010	0 7	X	1004
12 7	G	4004	0 8	Y	1002
12 8	H	4002	0 9	Z	1001

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12	&	+	&	+	&	4000
11	−	−	−	−	−	2000
12 0				?		5000
11 0				:		3000
8 2			:	←	:	0202
8 3	#	=	#	=	#	0102
8 4	@	-	@	@	@	0042
8 5			'	↑	'	0022
8 6			=	'	=	0012
8 7			"	\	"	0006
12 8 2			¢	?	|	4202
12 8 3	.	.	.	.	.	4102
12 8 4	¤	)	<	)	<	4042
12 8 5			(	]	(	4022
12 8 6			+	<	+	4012

INPUT OUTPUT CODES

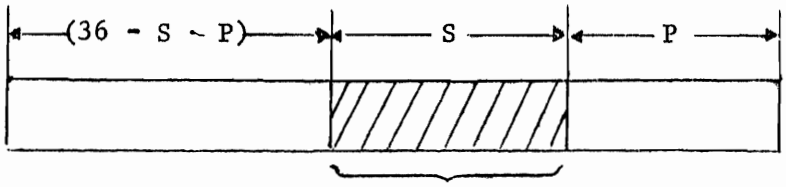
olumn Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12 8 7			|	!	↑	4006
11 8 2			!	:	!	2202
11 8 3	$	$	$	$	$	2102
11 8 4	*	*	*	*	*	2042
11 8 5			)	[	)	2022
11 8 6			;	>	;	2012
11 8 7			¬	&	\	2006
0 8 2			*See note*	;	]	1202
0 8 3	.	,	,	,	,	1102
0 8 4	%	(	%	(	%	1042
0 8 5			←	"	←	1022
0 8 6			>	#	>	1012
0 8 7			?	?	?	1006
12 11 0 1				*End of File*	*End of File*	7400
12 0 2 4 6 8				*Mode Switch*	*Mode Switch*	5252
7 9				*Binary*	*Binary*	*xx*05

NOTE: There is a single key for the 0 8 2 punch on the 029 but printing is suppressed.

The Monitor translates the octal code for the 12 0 punch in DEC 026 to 4202 (which corresponds to a , 12 8 2 punch), and the code for 11 0 to 2202 (11 8 2).

APPENDIX C: WORD FORMATS

<P,S> refers to a field S bits wide and P bits up from the right hand end of the word, thus:



referenced partial word

The format of a pointer is

P = <30,6>	Position
S = <24,6>	Size
I = <22,1>	Indirect address
X = <18,4>	IndeX
Y = <0,18>	

The format of an (non I/O) instruction is

F = <27,9>	Function code
A = <23,4>	Accumulator
I,X,Y as above	

The format of an integer number is

SIGN	= <35,1>
MAGNITUDE	= <0,35>

The format of a floating point number is

SIGN	= <35,1>
EXPONENT	= <27,8>
MANTISSA	= <0,27>

APPENDIX D: BLISS ERROR MESSAGES

NUMBER	MESSAGE
0*	undeclared identifier
1	error in simple expression
2	not the correct matching close bracket
3	expressions must be separated by a delimiter
4	an operator must be followed by a simple expression
5	a relational expression must not be followed by a relational operator
6	a unary (binary) operator must (not) be preceded by a delimiter
7	a control expression must not be used as a subexpression
10	left part of an assignment is incorrect
11	too many ←'s (current implementation allows 8)
12	righthand side of an assignment is incorrect
13	an actual parameter expression should not be empty
14	a simple expression should be followed by a delimiter
15	a subscript expression should not be empty
16	too many subscripts (current implementation allows 8)
20	OF must be followed by SET in CASE stmt
21	incorrect escape expression
22	missing control variable in INCR or DECR
23	the constituent expressions of a complex expression should not be empty
25	declarations are only allowed in a block head
26	
27	
30	current close br does not match marked open bracket (paired with err 31)
31	not the correct close bracket
32	
33	
34	
35	
36	illegal control variable name in INCR or DECR
37	empty condition in WHILE-DO, UNTIL-DO, DO-WHILE, or DO-UNTIL

*Warning message, not fatal

NUMBER	MESSAGE
40	illegal up-level addressing
41	too many parameters in a pointer expression
42	too many close brackets, or not enough opens (compiler exited to highest level before the eof on input file)
43*	as 42, except warning only, recovery attempted
44	FROM-TO-BY-DO out of order in INCR/DECR expression
45	empty DO part, may not be defaulted in INCR/DECR expression
46	empty condition in if-then-else not permitted
47	missing THEN
50	empty FROM,TO, or BY expression in incr/decr
51	number of levels in escape expression is not a literal
52	missing ']' in number levels part of an escape expression
53	empty expression not permitted as pointer-pointer in special function
54	missing ')' in a special function
55	missing 'OF' in select expression
56	missing or misplaced 'NSET' in select expression
57	labeling expression of nset-element may not be empty
60	missing or misplaced ":" in a SELECT expression
61	missing or misplaced TESN in select expression
62	empty elist element in SELECT expression
63	'SET' is not an allowed stmt beginner
64	No '(' after EXCHJ
65	Empty new-base expression in EXCHJ
66	Missing ')' in EXCHJ
67	Missing AT in CREATE
70	Missing AT-expression or LENGTH in CREATE
71	Missing LENGTH-expression or THEN in CREATE
72	Missing '(' after CREATE
73	

NUMBER	MESSAGE
74	symbol to be declared is not an identifier
75	missing "=" on a routine, function, or structure declaration
76	missing formal parameter list right delimiter, i.e., ")"|"]"|","
77	missing right bracket on the size portion of a "namesize"
100	missing delimiter on a list, i.e., "," or ";"
101	missing ")" on a name par.
102	
103	missing "=" in a machop declaration
104	missing "," | ":" | ";" in allocation declaration
105	
106*	structure access not to an identifier, e.g., 1[∈], 'vector' assumed
126*	register is neither reserved or 'system' type, see MODULE declaration
127*	register value out of range (0-15)
130	register number is not a literal
131*	attempted structure access to a variable which has not been mapped, 'vector' assumed
132*	extra incarnation actuals...ignored
133	size expression must not be a block
134	symbol may not be addressed, and hence may not be mapped
135	invalid expression in a FORWARD declaration
136	invalid expression in a MACHOP declaration
137	may not map a symbol of this type
140	attempting to map onto an undeclared structure
141	incarnation actual or resulting size expression is not a literal
142	
143	symbol previously declared in the current context (blocklevel)
144	invalid attempt to escape from routine or function
145	warning: using a temporary register may invalidate code
146	register position of a machine op. must be reg name or literal
147	
150	
170	illegal macro name
171	empty formal list in macro definition
172	more than 31 formals in macro formal list

*Warning message, not fatal

NUMBER	MESSAGE
173*	illegal formal parameter
174*	macro definition during macro expansion suppressed
175*	recursive macro call
176*	macro in use at block purge time
177*	"(" missing on macro call
200*	missing exponent on floating constant: ϕ assumed
201	compile time (floating) division by zero
250	
300	
350	
400	
450	
500	
600	
613	module declaration within module body
614	stack declaration not own, global or external
615	stack syntax error (pointer is accurate)
616	stack length invalid
617	trying to reserve a register in use already
620	special register reg num. not valid
621	trying to declare a special register already in use
622	module errors; compilation starts at pointer
623	reserved+special+declarable exceeds 12--reservation ignored
624	missing equals in special reg.
625	module declaration errors with a trivial program
626	syntax errors in entries switch
627	declared entry point is not defined
630	plit missing right paren
631	compile time expression error
632	load time expression error
633	negative plit duplication factor
634	may not use long string in this context

NUMBER	ESSAGE
760	no temporary register available
761	no declared registers available (INCR/DECR)
762	no declared registers available (declaration)
763	
764	
765	
766	
767	overlay compiler error in attempt at overlay
770	> 100 attempts made to expand space and failed
771	gt savef overflow
772	reg. table use field overflow
773	graph table UCCF overflow
774	literal table capacity exceeded
775	pointer table capacity exceeded
776	operand pair without intervening delimiter
777	compiler error

report these errors to the implementors!

Note: on some errors related to internal consistency checks the compiler may 'punt--that is, print an error message, abort compilation, and return to the user with an "*". These errors should always be reported to the implementors.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213	UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

BLISS REFERENCE MANUAL

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

W. A. Wulf, D. Russell, A. N. Habermann, C. Geschke,
J. Apperson, D. Wile, and R. F. Brender

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
January 15, 1970	64	

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-67-C-0058	
b. PROJECT NO. 9718	
c. 6154501R	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. 681304	

10. DISTRIBUTION STATEMENT

1. This document has been approved for public release and sale;
its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH, OTHER	Air Force Office of Scientific Research 1400 Wilson Boulevard (SRMA) Arlington, Virginia 22209

13. ABSTRACT

This document describes the BLISS implementation language as written for the PDP-10. BLISS is a language specifically designed for use as a tool in implementing large software programs. Special attention is given in the language design to the requirements of the systems programming task, such as: space and time efficiency, the representation of data structures, the lack of run-time support facilities, flexible control structures, modularization, and parameterization of programs.

DD FORM 1 NOV 65 1473

Security Classification